АЛЕКСАНДР ПАНОВ

bhv®

РЕВЕРСИНГ и ЗАЩИТА ПРОГРАММ ОТ ВЗЛОМА

Наиболее распространенные способы защиты программ и способы их преодоления

Примеры работы с отладчиками и дизассемблерами

Подробное описание отладчика OllyDBg

Методика написания средств защиты программ от взлома

PRO
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Александр Панов

РЕВЕРСИНГ и ЗАЩИТА ПРОГРАММ ОТ ВЗЛОМА

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.06
ББК 32.973.26-018.1
П16

Панов А. С.

П16 Реверсинг и защита программ от взлома. — СПб.: БХВ-Петербург, 2006. — 256 с.: ил.

ISBN 5-94157-889-X

Подробно изложены современные основные методики защиты программного обеспечения, начиная с составления программ и заканчивая их отладкой. Рассмотрены примеры взлома стандартных защит и даны рекомендации для предотвращения такого взлома. Приведено большое количество рабочих примеров, которые должны помочь программисту решить возникшие перед ним проблемы в защите его интеллектуальной собственности. Подробно описана работа с отладчиком OllyDBg. На сопроводительном компакт-диске находятся описанные в книге программы.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Татьяна Темкина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.06.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 20,64.
Тираж 2500 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 5-94157-889-X

Оглавление

Глава 1

Введение

Эта книга уникальна. До сих пор ничего подобного не издавалось. Это первая книга, посвященная только крекерам и разнообразным анализам защиты программ, а также способам ее преодоления.

Многие, наверное, задавались вопросом: как же появляются на свет различные креки (crack) и патчи (patch) к программам, откуда они берутся? Для того чтобы ответить на этот вопрос, и была написана эта книга. Прочитав ее, вы поймете разницу между хакером и крекером, узнаете, насколько увлекателен реверсинг (reverse code engineering), и научитесь снимать защиту с программ и создавать ее.

Надеюсь, книга вам понравится.

Для кого предназначена эта книга

Книга предназначена в первую очередь для всех тех, кто сам занимается написанием защиты программного обеспечения, всем программистам и тем, кто хотел бы защитить написанную им программу от изменения.

На страницах книги вы найдете множество рекомендаций по усилению защиты ваших программных продуктов, примеры и способы разработки разнообразных защит и т. д.

Книгу можно использовать как справочник по отладчику OllyDBg. Это первая книга, в которой вы найдете описание данного отладчика.

Книга также пригодится тем, кто интересуется, как именно взламываются программы, какие защиты существуют и что вообще такое "реверсинг".

Благодарности

В первую очередь я говорю огромное СПАСИБО моей жене Пановой Ольге Анатольевне, за терпение, поддержку и внимание, проявленные в процессе работы над материалом книги и ее подготовки к изданию. Если бы не она, книга никогда не была бы закончена.

Большое спасибо моим родителям за то, что они вырастили меня таким, какой я есть.

Большое спасибо и моему брату, Панову Владимиру Сергеевичу, именно с ним мы начинали осваивать компьютер.

Также хочу поблагодарить Масло Татьяну Викторовну за ее помощь и советы по изложению материала.

Я хочу сказать спасибо моему другу nice за его советы и помощь в написании книги. Без них книга не была бы подготовлена в срок.

Также хочу поблагодарить Bad_guy и всех жителей сайта **cracklab.ru** за помощь в написании этой книги.

Особая благодарность — высокопрофессиональному коллективу редакции издательства "БХВ-Петербург", предоставившего возможность издать данную книгу.

Кто такой хакер и как им стать

Для начала поговорим немного о терминах.

Сейчас в мире происходит путаница. Многие СМИ под "хакерами" подразумевают большой пласт компьютерных профессионалов — хакеров, крекеров, программистов, компьютерных пиратов и т. д., что неверно по сути, поскольку у них совершенно разные направления деятельности. Это то же самое, что путать директора предприятия и дворника, инженера и секретаршу. Конечно, ничто не мешает одному человеку выполнять все обязанности, но чаще это разные люди. Упомянутые области программмирования настолько обширны, что досконально знать их все, быть везде профессионалом просто невозможно. Для того чтобы стать профессионалом, необходима постоянная и долгая практика. Человек пока, к сожалению, не может одновременно делать несколько больших дел, например, читать техническую документацию, чертить чертеж и писать письмо любимой.

Итак, кто же такой хакер? *Хакер* — профессионал, который разрабатыват или взламывает сетевую защиту. Главный интерес хакера — это сайты и серверы в Интернете. Хакеру интересен сам процесс исследования защиты, но он никогда

не сделает что-либо, вызывающее крах исследуемой им системы. Закончив исследование системы защиты, хакер составит подробный отчет о проделанной им работе, чтобы администратор данного сайта смог лучше защитить данный сайт от взлома. Этим он отличается от компьютерного пирата. *Компьютерный пират* тоже исследует защиту сайтов, но главная цель пирата — порча или уничтожение информации. Этим обычно занимаются обиженные на весь мир недалекие подростки. Им кажется, что так они обратят на себя внимание, покажут свою "крутость". Они ничего не придумывают сами — просто скачивают из Интернета программы и пытаются ими воспользоваться, не понимая принцип их действия. К сожалению, как показывает практика, они не догадываются, что после того, как на них "обратят внимание", им уже больше ничего не захочется.

Как стать хакером, т. е. профессионалом, занимающимся исключительно защитой сайтов и серверов? Ответить на этот вопрос не так просто, как кажется. В первую очередь, необходимо постоянно читать техническую документацию по серверам, программным защитам сайтов, межсетевым экранам. Это связано с тем, что все в мире постоянно изменяется, и вам необходимо быть в курсе последних разработок. Информация даже годичной давности сегодня уже не принесет большой пользы. Во-вторых, необходимо приобрести знания администратора компьютерной сети. Это позволит понять, как "думают" администраторы, изучить их сильные и слабые стороны. Еще нужно изучить языки программирования, начав с Perl, HTML, Java (к другим языкам сетевого программирования лучше обратиться уже после того). Знать ассемблер хакеру, скорее всего, не нужно, как и Delphi, C++ и т. д.

Главное, что необходимо, — это упорный труд и желание, без этого никакие знания вам не помогут.

Кто такой крекер и как им стать

Теперь поговорим о том, кто такой крекер (cracker).

Cracker переводится как "взломщик программной защиты", т. е. тот, кто исследует защиту программ. Это разнообразные триал-защиты, защиты Shareware и т. д. Конечно, совершенно необязательно взлому иметь негативный характер, это может быть и взлом, заказанный автором программы с целью исследования того, насколько хороша его защита. После исследования программы крекер составляет подробный отчет, который он передает автору с рекомендациями по усилению (если оно необходимо) защиты программы. Крекер никоим образом не занимается сетевой защитой сайтов, серверов и т. д. Его стихия — это программы. Он копирует программу на свой компьютер

(в отличие от хакера) и исследует ее. Его компьютер — это своеобразная лаборатория по исследованию программ. Крекеру в первую очередь необходимо знание ассемблера. Это своего рода специфика работы, т. к. программы доходят до крекера главным, если не единственным, образом — в виде исполняемого модуля. Крекер также должен знать какой-либо язык программирования высокого уровня, в идеале C++ или Delphi. Это необходимо для написания разнообразных утилит, облегчающих жизнь крекера. Конечно, никто не запрещает писать эти утилиты на ассемблере, но крекеры очень ценят свое время, поэтому все, что может быть автоматизировано, будет ими автоматизировано с наикратчайшие сроки.

Из всего сказанного понятно, что крекерство — не то занятие, в котором обиженные на весь мир подростки могут выказывать свою "крутость", применяя скачанные из Интернета программы. Это тяжелый труд. Но от этого он не становиться менее увлекательным. Эта профессия затягивает. В ней очень много интересного. Это как противоборство интеллектов человека и машины.

Как стать крекером? Ответить на этот вопрос тоже не так просто. Для начала необходимо изучить ассемблер. В этом вам может помочь моя книга "Assembler: Экспресс-курс" ("БХВ-Петербург", 2006). Изучив ассемблер и написав на нем пару программ (чтобы убедиться в том, что вы его знаете и понимаете), загружайте с сопроводительного компакт-диска этой книги или из Интернета (например, с моего сайта **www.abyss-soft.narod.ru** или с сайта **cracklab.ru**) разнообразные программы типа crackme, созданные специально для того, чтобы их взломали. Начинать лучше всего с простых программ (например, с моих или с crackme Fantoma). После этого вам придется изучить язык программирования высокого уровня (например C++); это потребуется для автоматизации процесса реверсинга.

На моем сайте вы найдете программу для удаления мусора с дисков, написанную мной по заказу клубов; на нее я поставил защиту, чтобы вы смогли потренироваться. На сопроводительном диске книги тоже есть несколько программ, которые я написал специально для этого. На таких программах я демонстрирую способы защиты программного обеспечения, а на страницах этой книги показываю, как преодолеваются такие защиты, можно сказать, "веду за ручку" начинающего крекера. Прочтя данную книгу, вы "проскочите" тот трудный участок в начале пути, когда, разбираясь в основах, не у кого спросить об элементарных вещах, до обсуждения которых не снисходят более опытные "братья по оружию". Вы справитесь с этим играючи.

Также хочу сказать, что лучший сайт, посвященный реверсингу, какой я когда-либо видел, это **cracklab.ru**. На этом сайте есть много профессионалов (привет вам, nice, Bad_guy и Ara), которые помогут вам в этом нелегком деле.

Что понадобится для работы с книгой

Для работы с этой книгой вам понадобится несколько вещей. В первую очередь, это отладчик. Рекомендую вам очень мощный отладчик OllyDBg, подробнее о котором я расскажу немного позднее. Сейчас скажу только, что для большинства исследований он очень удобен. Найти его можно на просторах Интернета. Но вы можете взять и любой отладчик, который вам больше нравится, например, SoftICE. Еще вам будут нужны разнообразные программные мониторы — RegMon, FileMon, Regsnap. Также вам понадобится дизассемблер IDA, самый мощный из существующих (про него я тоже расскажу позднее).

Глава 2

Особенности ассемблера для исследования работы программ

В этой главе кратко описаны принципы работы персонального компьютера и ассемблера (языка Assembler). Для более подробного изучения языка ассемблер я рекомендую вам прочитать мою книгу "Assembler: Экспресс курс". Это пригодится вам для того, чтобы лучше понять, как именно можно спроектировать защиту программы или как ее можно снять (это зависит от целей, которые вы перед собой ставите).

Желающим побыстрее начать что-то делать рекомендую перейти сразу к четвертой главе; если понадобится, вы сможете вернуться к этой главе позднее. Но если вы ступили на путь реверсинга программ впервые, то я рекомендую вам читать все по порядку. Это поможет вам понять многие принципы, сэкономив много времени.

Приведенные здесь основы ассемблера позволят лучше понять, что же именно мы будем делать, как устроена защита той или иной программы и т. д. Я не буду объяснять весь синтаксис ассемблера. Это нам ни к чему.

Мы рассмотрим принципы работы компьютера и микропроцессора на примере микропроцессора 8086. Сейчас у многих имеются компьютеры с гораздо более совершенной архитектурой (AMD Athlon XP, Intel Pentium и др.) но все, что я расскажу о микропроцессоре 8086, справедливо и для всех остальных моделей процессоров (естественно, речь идет только об архитектуре PC).

Структурная схема компьютера

Структурная схема персональной ЭВМ, работающей на базе микропроцессора, представлена на рис. 2.1.

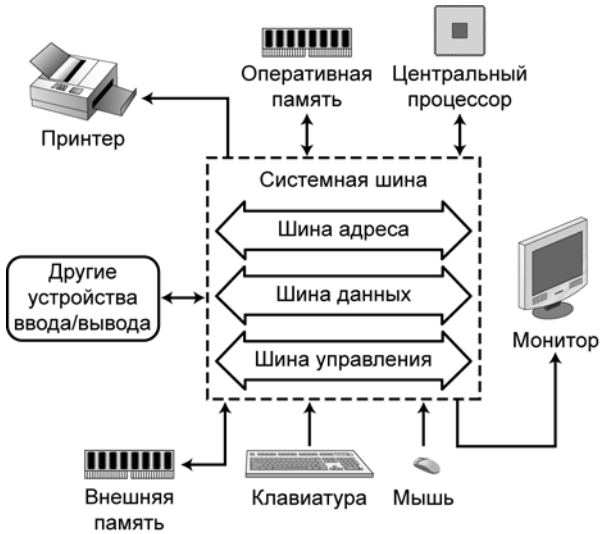

Рис. 2.1. Структурная схема микрокомпьютера

При работе персонального компьютера (также называемого *микрокомпьютером*) выполняются следующие действия:

1. Программа или данные вводятся с помощью устройств ввода из внешней памяти (жесткий диск, дискета и т. д.) в оперативную память.
2. Микропроцессор выбирает команды программы из оперативной памяти в строгом порядке, в котором их следует выполнять. Микропроцессор считывает указанные в этих командах данные и производит указанные командами действия. Адреса и данные передаются через *системную шину*, соединяющую центральный процессор, память и устройства ввода/вывода (рис. 2.1).
3. Данные, полученные в результате обработки команд, микропроцессор пересылает в оперативную память или на устройство вывода (дисплей, принтер, внешняя память и т. д.).

Микропроцессор расположен на *системной плате*, где также расположены микросхемы оперативной памяти и микросхемы *постоянного запоминающего устройства (ПЗУ)*, содержащего программу начальной загрузки памяти ЭВМ и другие вспомогательные программы. Различные компоненты компьютера подсоединяют к системной шине, состоящей из адресной шины, шины данных и шины управления. Связь между процессором, памятью и устройствами ввода/вывода осуществляется тоже через системную шину. Процессор подсоединяется к шине через блок логики управления шиной, устройства — через интерфейсы.

Архитектура микропроцессора

Главное, что должен делать центральный процессор, — упрощать работу со следующими объектами вычислительной системы:

- присваивания и арифметические выражения;
- безусловные/условные переходы;
- логические выражения;
- циклы;
- массивы/структуры данных;
- подпрограммы;
- устройства ввода/вывода.

Стандартная архитектура центрального процессора, обладающего такими возможностями, должна включать:

- набор регистров для адресации данных и выполнения арифметических операций;
- устройство управления для исполнения команд;

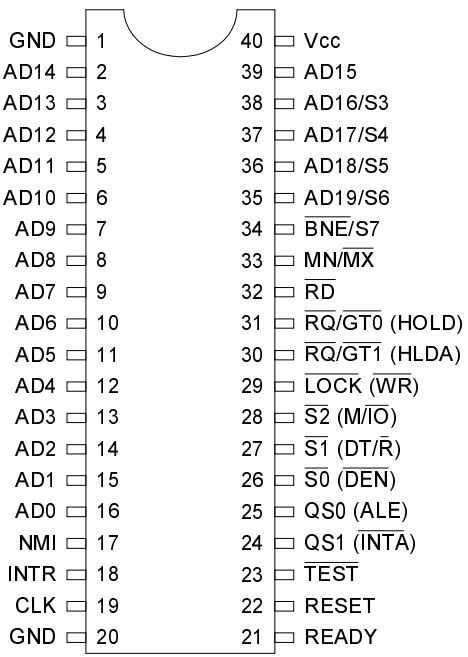

Рис. 2.2. Разводка контактов микропроцессора Intel 8086

- арифметико-логическое устройство (АЛУ) для выполнения арифметических и логических операций;
- секцию управления вводом/выводом.

Под *регистром* мы будем понимать запоминающее устройство для временного хранения целого числа байт, расположенное внутри микропроцессора.

Рассмотрим эти компоненты на примере микропроцессора Intel 8086 (рис. 2.2).

В компьютерах на базе данных процессоров используется системная шина, называемая *мультиплексной*, поскольку по ее линиям передаются как адреса, так и данные.

Сигналы принимаются и передаются через мультиплексную шину микропроцессором, который делится на две части — операционное устройство и шинный интерфейс (рис. 2.3).

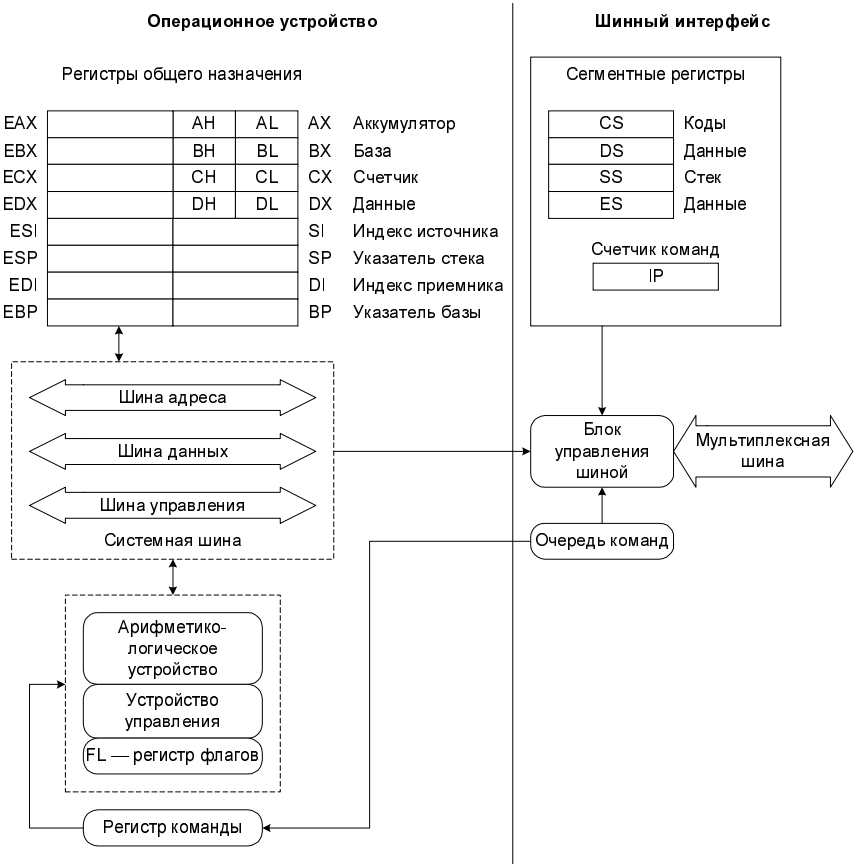

Рис. 2.3. Внутренняя архитектура микропроцессора Intel 8086

Операционное устройство предназначено для выполнения команд. Оно содержит АЛУ, устройство управления и 10 регистров.

Шинный интерфейс подготавливает команды и данные для выполнения и состоит из блока управления шиной, очереди команд и сегментных регистров. Он управляет передачей данных на операционное устройство, в память и на устройства ввода/вывода. В шинном интерфейсе адрес формируется с помощью сегментных регистров. Шинный интерфейс также имеет доступ к командам, находящимся в оперативной памяти, осуществляет выборку этих команд и помещает их в очередь команд. При этом шинный интерфейс должен как бы "заглядывать вперед" и выбирать команды так, чтобы всегда существовала непустая очередь команд, готовых для выполнения.

Согласно рис. 2.3 микропроцессор 8086 имеет 14 16-разрядных регистров, доступных для программиста. Начиная с микропроцессора i80386 пользовательских регистров стало 16 — добавились два сегментных регистра FS и GS. Остальные регистры расширились до 32 бит. Основные регистры микропроцессоров семейства 80x86 показаны на рис. 2.4. Регистры позволяют свести операции к командам, имеющим не более одного операнда, находящегося в памяти.

Рис. 2.4. Регистры микропроцессоров семейства Intel 80x86

Организация памяти

Внутренняя память микрокомпьютера делится на два типа.

Первый тип — это память, доступная только для чтения и называемая *постоянной памятью* (Read Only Memory, ROM). ROM-память представляет собой специальную микросхему, находящуюся на материнской плате, данные в которую записываются на заводе — изготовителе материнской платы. Основное назначение ROM-памяти — поддержка процедуры начальной загрузки. Для программиста наиболее важным элементом ROM-памяти является *базовая система ввода/вывода* (Basic Input/Output System, BIOS).

Память второго типа называется *памятью произвольного доступа* (Random Access Memory, RAM), или *оперативной памятью*. Эта память доступна для чтения и записи и является памятью, с которой работает программист. Содержимое оперативной памяти теряется при выключении питания компьютера. Поэтому для сохранения данных и программ требуются средства *внешней памяти* (обычно жесткий диск или дискета).

Память ЭВМ состоит из последовательности ячеек, каждая из которых имеет *адрес*. Таким образом, *память* — это область хранения информации, доступ к которой осуществляется по шине адреса.

Минимальная компьютерная единица информации называется *битом*. Бит может принимать только одно значение из двух — 0 или 1. Память разбивается на блоки различных размеров — байт, слово, сегмент, страница и др. Рассмотрим их подробнее.

Байтом называется минимальная адресуемая единица информации. Байт состоит из восьми бит. Таким образом, чтобы изменить один бит памяти, надо записать, по крайней мере, восемь бит в байт, содержащий этот бит. Каждому байту соответствует его собственный физический адрес.

Словом называется последовательность бит, число которых равно размеру регистра-аккумулятора. Например, в микропроцессоре 8086 регистр AX имеет длину 16 бит, т. е. слово состоит из 16 бит, или содержит 2 байта. В более современных компьютерах аккумулятором является регистр EAX, а это значит, что слово состоит из 32 бит (4 байта). В этом случае его называют *двойным словом*, оставляя название "слово" за областью памяти, состоящей из 16 бит.

Примечание

В 16-разрядных процессорах: слово — 16 бит, двойное слово — 32 бита, в 32-разрядных процессорах: слово — 32 бита, двойное слово — 64 бита.

Микропроцессор работает в одном из следующих режимов:

- *реальный режим* — режим, в котором работал процессор 8086;
- *защищенный режим* — режим, позволяющий максимально реализовать возможности процессоров 80*x*86 (начиная с процессора 80286 в этом режиме физический адрес формируется не так, как в реальном режиме);
- *режим виртуального 8086* — режим, возможный при работе процессора в защищенном режиме; в этом режиме могут работать несколько программ, разработанных для 8086; процесс формирования физического адреса для этих программ производится как в реальном режиме;
- *режим системного управления* — режим, появившийся в микропроцессоре Pentium; функционирование процессора в этом режиме по общим принципам подобно его работе в реальном режиме.

Мы только начинаем изучать язык ассемблера, поэтому для простоты работы будем рассматривать адресацию памяти в реальном режиме.

Сегментом называется независимый, поддерживаемый на аппаратном уровне блок памяти, имеющий размер не больше 2^{16} байт (поскольку $2^{16} = 2^6 \cdot 2^{10} = 64 \cdot 1024$, а 1024 байт = 1 Кбайт, максимальный размер сегмента будет равен 64 Кбайт).

Параграфом называется минимальный размер сегмента 16 байт.

Адресная шина процессора 8086 состоит из 20 линий, поэтому максимальный адресуемый объем памяти в реальном режиме равен 2^{20} байт = 2^{10} Кбайт = = 1 Мбайт. В защищенном режиме для процессора 80286 он равен 2^{24} байт = 16 Мбайт, а для процессоров 80386 и выше — 2^{32} байт = 4 Гбайт.

Память разделяется на сегменты. Адрес ячейки формируется подобно почтовому адресу (сегментный адрес указывает "улицу", а смещение — "номер дома"). Физический адрес байта, имеющего смещение (т. е. номер байта в сегменте) *x* относительно начала сегмента, указанного с помощью сегментного регистра S, будет равен $16 \times S + x$. Например, регистр CS (сегмент кодов) используется для указания сегмента, в котором находится код программы. Так как содержимое счетчика команд IP равно смещению текущей команды относительно начала сегмента, физический адрес EA текущей команды будет равен $16 \cdot CS + IP$.

Вернемся к рис. 2.3 и рассмотрим процессы выполнения команд и обмена данными между микропроцессором и оперативной памятью.

1. По содержимому регистров CS и IP, хранящих адрес команды в виде базового адреса, т. е. адреса начала сегмента (CS), и смещения (номера байта в сегменте) (IP), определяется местонахождение (EA) команды, выполняемой на следующем шаге.

2. Вычисленный адрес запоминается в стеке.
3. Из оперативной памяти считывается команда с этим адресом и пересылается в процессор.
4. Команды помещаются в очередь команд и выполняются в порядке очередности.
5. Местоположение данных, участвующих в командах, определяется в шинном интерфейсе по информации, содержащейся в адресной части команды, в регистрах сегментов DS, SS и ES, а также в регистрах SP, BP, SI, DI.
6. Данные, прочитанные из памяти, пересылаются в АЛУ или регистр данных (DS) и обрабатываются текущей командой, адрес которой был найден на шаге 2. Результаты обработки данных помещаются в оперативную память или в какой-либо из регистров.

Циклическое повторение описанных процедур составляет выполнение программы. Номер сегмента называется *адресом сегмента (сегментным адресом)*, а номер байта в сегменте — *смещением*. На практике указывают лишь смещение, а также по умолчанию предполагается, что значение регистра DS известно.

Надо помнить, что любой сегмент имеет объем до 64 Кбайт и имеется 4 сегментных регистра; таким образом, доступна память 256 Кбайт. Но в действительности в программе возможно любое количество сегментов. Для того чтобы адресовать любой сегмент, достаточно изменить значение сегментного регистра.

Память RAM включает в себя первые три четверти внутренней памяти, а ROM — последнюю четверть. В памяти объема 1 Мбайт ROM начинается с 768-го килобайта. Распределение памяти при работе в операционной системе MS-DOS показано на рис. 2.5 (слева указаны начальные адреса блоков памяти).

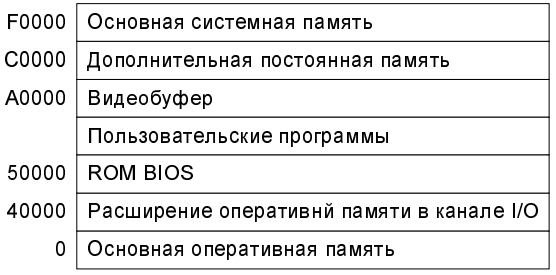

Рис. 2.5. Распределение памяти в MS-DOS

Представление данных

Данные в памяти ЭВМ имеют двоичное представление. В самом общем случае мы можем отвести под целое число любое количество соседних байтов. Программисту приходится работать с числами разных форматов (не только с целыми). Рассмотрим некоторые наиболее часто используемые типы данных:

- целые числа;
- двоично-десятичные числа;
- неупакованный BDC-формат;
- упакованный BDC-формат;
- числа с плавающей точкой;
- символьный тип.

Для быстрого старта этого достаточно, а подробную информацию о любых типах данных вы можете получить из специальных источников.

Целые числа

Система команд ЭВМ поддерживает работу с целыми числами *без знака* и *со знаком*, занимающими байт, слово или двойное слово.

Целое число без знака в двоичной записи состоит из k бит, где $k = 8$, 16 или 32.

Байтом 1010 1100 представляется число $2^7 + 2^5 + 2^3 + 2^2 = 172$, а словом 0111 0111 0000 1001 — число $7 \cdot 16^3 + 7 \cdot 16^2 + 9 = 30\,473$.

Самое большое число, которое можно представить байтом, равно 255, а наибольшее число, представляемое словом, — $2^{16} - 1 = 65\,535$.

Диапазоны значений целых чисел со знаком, имеющих разную длину:

- байт: от –128 до 127;
- слово: от –32 768 до 32 767;
- двойное слово: –2 147 483 648 до 2 147 483 647.

Целые числа со знаком записываются в дополнительном коде: положительное число записывается как беззнаковое (его старший бит равен 0), а отрицательное число $-x$ (при $x > 0$) представляется как $2^k - x$, где k — количество разрядов ячейки памяти, отведенной под число (k = 8,16 или 32). Если в этой ячейке записано некоторое беззнаковое число z, старший бит которого равен 1, то это число представляет число со знаком, равное $-(2^k - z)$.

Например, байт 10101100 является беззнаковым числом 172, которое превращается в $-(256 - 172) = -84$, если рассматривать его как число со знаком.

Итак, отрицательное число $-x$ ($x > 0$) представляется как $2^k - x$; его старший разряд должен содержать 1. Это представление можно получить с помощью следующих действий:

1. Записать двоичное представление числа x.
2. Инвертировать двоичное число, т. е. в разряды, содержащие единицы, записать нули, а в разряды, содержащие нули, — единицы.
3. К полученному числу прибавить 1.

Например, для числа –84 (байт) выполним следующие действия:

1. $84 = 2^6 + 2^4 + 2^2$, следовательно, двоичное представление: 01010100.
2. Инвертируем все разряды: 10101011.
3. Прибавляем 1: 10101100.

Двоично-десятичные числа

Затраты на перевод чисел из десятичной формы записи в двоичную и наоборот могут быть чрезмерно велики. Поэтому предусмотрено специальное представление целых чисел — двоично-десятичный код (binary coded decimal, BCD), строящийся по следующему принципу: берется десятичная запись числа и каждая его цифра заменяется на четыре двоичные цифры. Например, число 326 заменяется на последовательность бит 0011 0010 0110. Такой формат называется *BCD-форматом*. Различают *неупакованный* и *упакованный* BCD-форматы.

Неупакованный BCD-формат. На каждую десятичную цифру числа отводится один байт. Значение байта равно значению его младшей тетрады (четырех бит). Для обозначения знака применяются неиспользованные тетрады (старший байт). Если использовать ASCII-коды для представления цифр и знаков, то число будет состоять из символов:

00110000 = '0'	00110110 = '6'
00110001 = '1'	00110111 = '7'
00110010 = '2'	00111000 = '8'
00110011 = '3'	00111001 = '9'
00110100 = '4'	00101011 = '+'
00110101 = '5'	00101101 = '-'

Например, число –326 будет представлено такой последовательностью байт: '6', '2', '3', '-'. В программе это записывается с помощью директивы `db`:

```
db      '623-'
```

Запись числа в неупакованном BCD-формате может состоять из произвольного фиксированного числа байт. При выполнении арифметических операций отрицательные числа могут быть представлены в дополнительном коде — если число байт формата равно n, то отрицательное число записывается как $10^n - x$. Например, при $n = 4$ число –326 будет представлено символьной строкой '4769'.

Упакованный BCD-формат. В каждом байте записываются две десятичные цифры. Число состоит из 20 цифр и занимает 10 байт, старший байт отводится под знак.

Такое число определяется с помощью директивы `dt`. Например, число $x = 12\,345$ будет определено так:

```
x       dt      12345
```

и состоять из двадцати цифр — 00000000000000012345, каждая из которых записана четырьмя битами. В этом случае цифры располагаются в памяти парами в виде такой последовательности байт: 01000101, 00100011, 00000001, 00000000, 00000000, … (т. е. 45, 23, 01, 00 и т. д.).

Примечание

Байты в памяти компьютера располагаются в обратном порядке. Это связано с некоторыми особенностями строения персональных компьютеров. А для упакованного BCD-формата число условно разбивается на двузначные числа (недостающие разряды дополняются нулями). Например, число 47 281 будет разбито на пары 81, 72, 04.

Первый байт будет иметь смещение (относительный адрес) x, второй — $x + 1$, третий — $x + 2$ и т. д.

Самый старший бит десятого байта используется для хранения знака числа.

Например, запись

```
x       dt      -12345
```

будет транслироваться как строка, значение старшего байта которой — 80, затем идут шесть нулевых байтов, а затем — байты с значениями 01, 23 и 45.

Также надо иметь в виду, что для форматов BCD может применяется противоположный порядок хранения байтов в памяти. Формат хранения целиком зависит от квалификации и предпочтения программиста. Более подробную информацию вы можете найти в справочной литературе.

Числа с плавающей точкой

Как мы знаем, к вещественному типу относятся числа с фиксированной и плавающей точкой. Для ассемблера числа с фиксированной точкой и целые числа являются одним и тем же. Поэтому мы рассмотрим только числа с плавающей точкой, которые будем называть вещественными.

Число с плавающей точкой определяется значениями его знака, порядка и мантиссы. В частности, если под число отводится 4 байта, то эти 32 бита распределяются так:

- 1 бит (31-й) — знак (S);
- 8 бит (с 23-го по 30-й) — порядок (E);
- 23 бита (с 0-го по 22-й) — мантисса (M).

В этом случае 32 бита представляют число с плавающей точкой, значение которого равно $(-1)^S \cdot 2^{E-127} \cdot 1.M$.

Примечание

Тем, кто хочет подробнее разобраться в этом, я рекомендую обратиться к соответствующей литературе, в которой рассматривается программирование сопроцессора (например, [6]).

Следовательно, наибольшее положительное и наименьшее отрицательное значения такого числа равны $\pm 2^{255-127} \cdot (2 - 2^{-23})$. Наименьшее положительное число, записанное в формате с плавающей точкой, равно 2^{-127}.

Например, число 1 в формате с плавающей точкой будет представлено битами 00111111 10000000 00000000 00000000, т. е. $1 = (-1)^0 \cdot 2^{127-127} \cdot 1.0$.

Дробное число переводится в двоичную форму записи следующим образом: сначала целая часть переводится в двоичную форму, затем ставится точка и вычисляются разряды дробной части. С этой целью дробная часть последовательно сравнивается с числом $1/2^i$, начиная с $i = 1$, т. е. сначала с 1/2, затем с 1/4, 1/8 и т. д. (пока полученное число удовлетворяет необходимой нам разрядности). Если дробная часть меньше, записывается 0, а если больше или равна, то записывается 1 и из дробной части вычитается это число. Затем остаток дробной части сравнивается со следующим числом $1/2^i$.

Например, пусть $x = 1/3$:

1. x меньше 1/2, поэтому записываем 0.
2. x больше 1/4, поэтому записываем 1 и вычисляем новое значение:
3. $x = 1/3 - 1/2 = 1/12$.
4. x меньше 1/8, записываем 0.
5. x больше 1/16, поэтому записываем 1 и вычисляем новое значение $x = 1/12 - 1/16$.

и т. д.

В результате получаем: 1/3 = 0,0101… (двоичное число).

Символьные данные

Символьные данные используются, например, для работы с текстовыми строками.

Символьные данные заслуживают отдельного описания, они как бы стоят в стороне. На них надо обратить особое внимание при реверсинге. Дело в том, что при вводе пароля или серийного номера в программу программа что-то делает с введенными данными, а потом сравнивает с данными, находящимися в самой программе. Из этого следует, что в программе уже есть эти данные и в большинстве своем они представлены именно как "символьные".

Обычно в ЭВМ используются восьмиразрядные коды символов ASCII. Отметим следующие особенности.

- Коды латинских букв упорядочены согласно алфавиту и идут без пропусков. Это очень важно! В частности, если к коду буквы `'a'` прибавить 1, то получится код буквы `'b'`, `'a' + 2` будет равно `'c'` и т. д.
- Коды цифр упорядочены по возрастанию и идут без пропусков, значит, все x в пределах `'0'` $\le x \le$ `'9'` будут цифрами.

Представление данных сопроцессора

Математический сопроцессор серии 80x87 может обрабатывать 7 типов данных:

- целое число со знаком;
- короткое целое со знаком;
- длинное целое со знаком;
- целое число в упакованном BCD-формате;
- короткое вещественное число;
- длинное вещественное число;
- расширенное вещественное число.

Диапазоны изменения чисел первых трех типов соответственно:

$-2^{15} \le x \le 2^{15} - 1$;

$-2^{31} \le x \le 2^{31} - 1$;

$-2^{63} \le x \le 2^{63} - 1$.

Поскольку в упакованном BCD-формате старший байт отводится под знак, в этом формате число состоит из 18 десятичных цифр и, значит, лежит в диапазоне $-10^{18} \le x \le 10^{18} - 1$.

Короткое вещественное число занимает 32 бита и было описано ранее.

В длинном формате порядок E занимает 11 бит, мантисса M — 53 бита. Значение числа, представленного в длинном формате, равно $(-1)^S \cdot 2^{E-1023} \cdot 1.M$.[1]

В расширенном формате порядок E занимает 15 бит, мантисса M — 64 бита. Значение такого числа: $(-1)^S \cdot 2^{E-16\,383} \cdot 1.M$.

Регистры микропроцессора

Регистром процессора называется ячейка процессора, отведенная для хранения чисел или предназначенная для специальных операций.

Другими словами, регистры — это ячейки, расположенные в центральном процессоре и доступные из программы. В программах на языке ассемблера регистры используются постоянно для различных целей — временного хранения данных, аргументов или результатов различных операций. В регистре может, например, храниться, флаг, сигнализирующий о том, зарегистрирована программа или нет, и т. д. Большинство регистров имеют определенное функциональное назначение. Но есть и регистры, которые можно без ограничения использовать для любых операций.

Поскольку для доступа к регистру адрес не нужен, есть возможность организовать операции между регистром и памятью с помощью команд, имеющих лишь один операнд, находящийся в памяти. Другое назначение регистров — адресация данных с помощью сегментных, базовых и индексных регистров. Наконец, информация о состоянии процессора также находится в одном из регистров.

Регистры общего назначения

При обработке данных компьютером значительная часть времени микропроцессора тратится на передачу данных между микропроцессором и памятью. Время доступа к данным значительно уменьшается, если часто используемые операнды и результаты вычислений хранить в самом микропроцессоре. Четыре регистра — EAX (AX), EBX (BX), ECX (CX) и EDX (DX) — предназначены специально для этих целей. Тут надо немного сказать о самих регистрах. Когда на заре компьютерной индустрии был изобретен процессор 8086—80286, все регистры были 16-битные и назывались они AX, BX, CX, DX. Каждый из них состоял из 8-битных старшего и младшего байтов. Так как программистам довольно часто приходилось использовать эти регистры,

[1] Бит знака хранится не в самом числе — для этого используется флаг переноса.

они получили свои имена — старшие байты этих регистров стали называть AH, BX, CX, DX, а младшие — AL, BL, CL, DL (рис. 2.6). После появления на свет процессора 80386, уже 32-битного, программистам стало довольно сложно и неудобно работать с этими регистрами — вместо одной команды пересылки 32-битных данных приходилось использовать две и больше. Вот для того чтобы облегчить труд программистов, и появились в процессоре 80386 32-битные регистры — EAX, EBX, ECX, EDX.[1] Таким образом, сейчас получается (см. рис. 2.6), что младшие 16 бит каждого 32-битного регистра могут использоваться как самостоятельные регистры. Все эти регистры называются *регистрами общего назначения*.

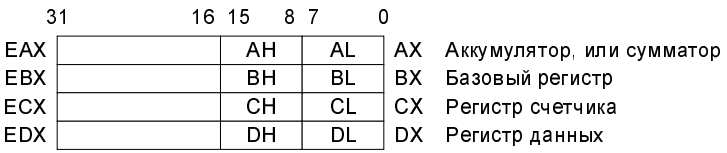

Рис. 2.6. Регистры общего назначения

Регистр AX (EAX). Является основным сумматором и применяется для арифметических операций, строковых операций, операций ввода/вывода. Команды умножения (`mul` и `imul`), деления (`div` и `idiv`), преобразования (`xlat`), коррекции (`aaa`, `aad`, `aam`, `aas`, `daa`, `das`) используют регистр AL — младший байт регистра AX. Старший байт регистра AX — регистр AH используется для задания функций обслуживающих подпрограмм (обработки прерываний).

Регистр BX (EBX). Применяется для вычислительных операций и в различных методах адресации. Это единственный регистр общего назначения, который можно использовать в качестве индекса элементов массива (для косвенно-регистровой адресации). Применяется в команде преобразования (`xlat`) для указания начала таблицы.

Регистр CX (ECX). Необходим для управления числом повторений в командах цикла (счетчик). Применяется в командах сдвига для указания числа бит, на которое сдвигается содержимое операнда. Содержит число повторений строковых операций при наличии префикса повторения.

Регистр DX (EDX). Применяется в командах ввода/вывода (`in` и `out`), умножения (`mul` и `imul`), деления (`div` и `idiv`), использующих регистровую пару DX:AX.

[1] E в названии регистра означает "32-битный".

Регистровые указатели и индексные регистры

Регистровые указатели SP и BP обеспечивают доступ к данным в сегменте стека. Регистр BP используется также в различных методах адресации.

Индексные регистры SI и DI применяются в строковых операциях и в различных методах адресации (рис. 2.7).

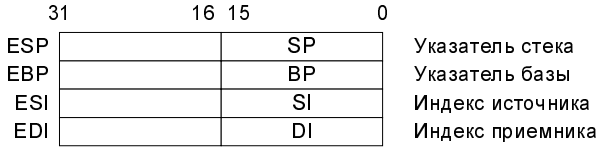

Рис. 2.7. Регистровые указатели и индексные регистры

Регистр SP (ESP). Указатель стека обеспечивает использование стека в памяти.

Стеком называется свободная область памяти, предназначенная для временного сохранения данных с целью их дальнейшего восстановления.

Адрес начала стека равен 16 × SS + SP, где SS — содержимое регистра сегмента стека.

Регистр BP (EBP). Облегчает доступ к параметрам, значения которых передаются с помощью сохранения данных в стеке.

Регистр SI (ESI). Является указателем источника. Применяется в строковых операциях. В строковых операциях связан с регистром сегмента данных DS. Адрес источника равен 16 · DS + SI.

Регистр DI (EDI). Является указателем приемника. В строковых операциях связан с дополнительным регистром сегмента данных ES.

Адрес приемника равен 16 · ES + DI.

Сегментные регистры

Как уже говорилось, физический адрес любой ячейки памяти состоит из 16-битного значения адреса сегмента и 16-битного смещения внутри сегмента. Каждый сегментный регистр используется для адресации определенного типа. Имеется четыре сегментных регистра: CS, DS, ES, SS. Микропроцессоры фирмы Intel, начиная с 80486, имеют два новых дополнительных сегментных регистра — FS и GS.

Регистр CS. Определяет сегмент кода — сегмент программы, содержащий выполняемые команды. Устанавливается при загрузке программы автоматически.

Регистр DS. Определяет сегмент данных или некоторую область памяти. В строковых операциях связан с регистром источника SI. Команды обработки данных по умолчанию, как правило, используют для адресации регистр DS. При загрузке программы значение регистра DS устанавливается программистом.

Регистр ES. Определяет дополнительный сегмент данных и применяется в тех случаях, когда требуется обратиться к произвольному сегменту памяти. В строковых операциях связан с регистром приемника DI. Инициализируется программистом.

Регистр SS. Регистр сегмента стека, связан с регистром SP. Содержит начальный адрес сегмента стека (в то время как SP устанавливается на конец стека). Регистры SS и SP при загрузке программы устанавливаются автоматически.

Регистры состояния и управления

В микропроцессоре присутствует несколько регистров, содержащих информацию о состоянии микропроцессора и программы, команды которой находятся в очереди команд ("загружены на конвейер"). К этим регистрам относятся счетчик команд IP (EIP), регистр флагов FL (EFL) и системные регистры.

Счетчик команд IP (EIP). Называется также *командным указателем*. Он связан с регистром сегмента кодов CS и содержит смещение команды, которая должна быть выполнена. При выполнении команды значение счетчика увеличивается за исключением тех случаев, когда команда относится к командам управления — перехода, цикла, вызова подпрограммы и возвращения из подпрограммы (включая программные прерывания).

Регистр флагов FL (EFL). Содержит 16 (32) бит. Отдельные биты имеют определенное функциональное назначение и называются *флагами*. Различают три типа флагов:

- системные флаги — отражают текущее состояние компьютера в целом и чаще используются операционной системой, а не программами пользователя;
- флаги состояния — изменяются после выполнения каждой команды;
- флаги управления.

Формат регистра флагов показан на рис. 2.8, назначение флагов приведено в табл. 2.1—2.3. Отметим, что бит 1 регистра флагов всегда равен единице. Остальные неиспользованные для флагов биты равны нулю (это относится, в частности, к битам 22—31).

Рис. 2.8. Регистр флагов EFL

Как мы знаем, микропроцессор может работать в одном из трех режимов — в реальном, защищенном и в режиме виртуального 8086 (V86). В разных режимах процессора флаги используются по-разному.

Таблица 2.1. *Системные флаги*

Флаг	Название	Бит	Назначение
ID	Флаг идентификации	21	Служит для проверки того, выполняет ли процессор команду `cpuid`. Если в программе можно установить и сбросить флаг ID, то данный процессор поддерживает команду `cpuid`, предоставляющую программисту информацию о продавце, модели и поколении данного процессора

Таблица 2.1 *(окончание)*

Флаг	Название	Бит	Назначение
VIP	Ожидание виртуального прерывания	20	Как и флаг VIF, позволяет каждой прикладной программе в многозадачном режиме иметь виртуальную версию флага IF
VIF	Флаг виртуального прерывания	19	Является виртуальным подобием флага IF и используется совместно с флагом VIP
AC	Контроль выравнивания	18	Предназначен для контроля выравнивания при обращениях к памяти. Используется совместно с битом AM в системном регистре CR0. К примеру, микропроцессор Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию
VM	Виртуальный режим	17	Признак работы микропроцессора в режиме виртуального 8086 (V86) — процессор работает в режиме V86 либо в реальном, или защищенном режиме (режим V86 является специальным подмножеством защищенного режима)
RF	Флаг возобновления	16	Служит для временного выключения обработки исключительных ситуаций отладки для того, чтобы команда, вызвавшая такую ситуацию, могла быть перезапущена и не стала бы причиной новой исключительной ситуации. Отладчик устанавливает этот флаг с помощью команды `iretd` при возврате в прерванную программу
IF	Разрешение прерываний	9	1 — микропроцессор воспринимает и соответственно реагирует на запрос прерывания по входу `intr`; 0 — прерывания по этому входу запрещаются. Значение флага IF не влияет на восприятие немаскируемых прерываний, выполняемых по команде `int`. Устанавливается командой `sti`, сбрасывается с помощью команды `cli`
TF	Флаг трассировки	8	1 — микропроцессор переходит в пошаговый режим работы, при котором генерируется прерывание 1 после выполнения каждой команды. Флаг можно установить с помощью команд `popf`, `popfd` или `iret`

Следующий пример показывает, как можно запретить или разрешить прерывания, что отражается на регистре флагов в разных режимах:

```
sti     ; Разрешить прерывание
cli     ; Запретить прерывания
```

Существует один *флаг управления*, воздействующий на циклические команды (табл. 2.2).

Таблица 2.2. *Флаг управления*

Флаг	Название	Бит	Назначение
DF	Направление	10	Управляет направлением операций: 0 — циклы обрабатываются от младших адресов к старшим; 1 — циклы обрабатываются от старших к младшим. Очищается с помощью команды `cld`, устанавливается с помощью команды `std`

Например:

```
; Установить обратное направление обработки байт
std
```

Таблица 2.3. *Флаги состояния*

Флаг	Название	Бит	Назначение
NT	Вложенная задача	14	Используется в защищенном режиме для фиксации того факта, что одна задача вложена в другую
IOPL	Уровень привилегий ввода/вывода	12, 13	Используется в защищенном режиме для управления доступом к командам ввода/вывода в зависимости от привилегированности задачи
OF	Переполнение	11	Изменение старшего и знакового битов результата при переносе или заеме. Номера знакового и старшего битов: 7 и 6 — при байтовых операциях, 15 и 14 — при 16-битных операциях, 31 и 30 — при 32-битных операциях
SF	Знак	7	Знаковый бит результата
ZF	Ноль	6	Устанавливается, если результат операции равен 0. Очищается, если результат ненулевой
AF	Вспомогательный перенос	4	Применяется для работы с числами в BCD-формате. Устанавливается в 1, если в результате операции сложения был произведен перенос из бита 3 в бит 4, или при операции вычитания — заем из бита 4 в бит 3. Иначе при выполнении этих операций устанавливается в 0

Таблица 2.3 *(окончание)*

Флаг	Название	Бит	Назначение
PF	Флаг четности (паритет)	2	Принимает значение 0, если сумма значений битов младшего байта результата нечетная, и 1 — если четная
CF	Перенос	0	Устанавливается при переносе из знакового бита результата или при заеме в знаковый бит

Установку флагов состояния лучше всего рассматривать на примерах команд сложения и вычитания. Будем рассматривать операции над байтами. Данные примеры удобно анализировать в любом отладчике, например, TD.

Флаг PF устанавливается, если младший байт результата имеет четное число ненулевых разрядов. В этом случае, при работе с отладчиком DEBUG в режиме трассировки, этот флаг будет выведен как PE (parity even), в противном случае PF = 0 будет выведен как `PO` (parity odd). Например, после выполнения команд

```
mov AL, 1
mov BL, 1
add AL, BL
```

значение PF будет равно 0 (PO). А после выполнения команд

```
mov AL, 2
mov BL, 1
add AL, BL
```

значение PF будет равно 1 (PE).

Флаг AF устанавливается при переносе из бита 3 в бит 4 и заеме из бита 4 в бит 3. Давайте рассмотрим команды (`xxxxb` обозначает число в двоичном представлении):

```
mov AL, 1000b
mov BL, 1000b
add Al, BL
```

После выполнения этих команд флаг AF будет равен 1. А, например, при сложении AL = 4 и BL = 4 флаг AF очистится.

Примечание

При сложении чисел в столбик, справа налево, в случае, когда сумма цифр больше девяти, возникает ситуация, которая называется *переносом* из разряда, в котором находятся эти цифры. При вычитании чисел, если цифра первого числа больше цифры второго числа, возникает *заем* в разряд, к которому принадлежат эти цифры.

Согласно описанию, флаг CF устанавливается при переносе знакового бита и заеме в знаковый бит. Флаг CF будет установлен в следующем случае:

```
mov AL, 110000000b
mov BL, 110000000b
add AL, BL
```

поскольку был произведен перенос из знакового бита (и полученная сумма один бит потеряла).

Для команды вычитания

```
sub операнд1, операнд2
```

флаг CF устанавливается в случае заема.

После выполнения команд

```
mov AL, 0
sub AL, 1
```

будут установлены флаги CF, AF, PF и SF.

После выполнения команд

```
mov AL, 10000000b
add AL, 10000000b
```

установятся два флага — CF и OF.

После выполнения команд

```
mov AL, 01000000b
add AL, 11000000b
```

установится флаг CF.

После выполнения команд

```
mov AL, 10000000b
sub Al, 01000000b
```

установится флаг OF.

Таким образом, флаг CF устанавливается при выходе результата за пределы слова (байта).

Операции сложения и вычитания могут устанавливать также флаги SF и ZF. Например, после выполнения команд

```
mov AL, 10101010b
add AL, 10000000b
```

флаги состояния будут иметь следующие значения: CF = 1, PF = 0, AF = 0, ZF = 0, SF = 0, OF = 1.

После выполнения команд

```
mov AL, 10101010b
add AL, 00111111b
```

флаги состояния будут иметь следующие значения: CF = 0, PF = 0, AF = 1, SF = 1, OF = 0.

В заключение отметим, что команда сравнения `cmp` устанавливает флаги, как команда вычитания `sub`, но не записывает результат в первый операнд. Например, после выполнения команд

```
mov AL, 01000000b
cmp AL, 11000000b
```

будут установлены флаги OF и CF, а после выполнения команд

```
mov AL, 0
cmp AL, 1
```

будут установлены флаги AF, PF и SF.

Заметим, что флаг CF можно установить с помощью команды `stc` или очистить с помощью команды `clc`:

```
stc    ; Установить флаг CF
clc    ; Очистить флаг CF
```

Способы адресации

Зачем нужны способы адресации? Представьте себе, что вы хотите написать небольшую программу, т. е. программу с кодом очень маленького размера. Это может быть, например, тренер для игры, дающий бесконечные жизни или ресурсы, драйвер, загрузчик операционной системы и др. — главное, чтобы программа занимала как можно меньше места. Для этого вы создаете односегментную COM-программу. Но тут у вас возникает трудность — ведь в программе вам нужны не только коды, но и данные для работы! А процессор не делает различия между данными и кодом — ему все равно, что исполнять (если он в сегменте кодов встретит данные, он исполнит и их), — что у вас в результате получится и как это будет работать, заранее не знает никто. О том, чтобы этого не происходило, программист должен позаботиться сам. Адресация — один из видов такой "заботы". Немного погодя мы рассмотрим пример, на котором я покажу, как можно в одном сегменте разместить и код программы, и данные, необходимые для ее работы.

Максимальная длина машинной команды — 15 байт. Команда может состоять даже из одного байта. Коду операции предшествуют однобайтные пре-

фиксы, модифицирующие операцию. После префиксов идет код операции. Последующие поля машинной команды определяют местонахождение операндов.

Операнды могут адресоваться следующими способами (если не считать того, что некоторые операнды могут задаваться неявно, как, например, регистры AX и DX в операциях умножения и деления, или CX в командах цикла).

- *Непосредственная адресация.* Операнд находится в поле команды. Например, в командах

```
mov AX, 512
mov DX, offset mes
```

  вторые операнды (`512` и `offset mes`) задаются непосредственно.

- *Регистровая адресация*. Операнд находится в одном из регистров. Например, оба операнда в команде

```
mov DS, AX
```

  задаются с помощью регистрового способа адресации.

- *Косвенно-регистровая адресация*. Адрес операнда находится в одном из регистров — SI, DI, BX. Например, команда

```
mov AX, [SI]
```

  переписывает содержимое слова, смещение которого указано в регистре SI, в регистр AX. Второй операнд задан с помощью косвенно-регистрового метода адресации.

- *Прямая адресация*. Относительный адрес операнда содержится в команде в виде смещения. В частности, команды пересылки в аккумулятор транслируются следующим образом:

```
mov AL, db   ; db - смещение байта
mov AX, dw   ; dw - смещение слова
mov EAX, dd  ; dd - смещение двойного слова
```

- *Базовая адресация.* Относительный адрес операнда равен сумме содержимого базового регистра (BP или BX) и смещения (8- или 16-разрядного). Например:

```
mov AX, ddd [BX]
mov AX, ddd [BP]
```

- *Индексная адресация.* Относительный адрес операнда формируется путем сложения содержимого индексного регистра (SI или DI) и смещения (8- или 16-разрядного). В частности, если второй операнд команды `mov`

является индексным, то команда, как и в случае базовой адресации, транслируется как `mov AX, xxxx [r]`. Например:

```
mov AX, ddd [SI]
mov AX, ddd [DI]
```

- *Базово-индексная адресация.* Относительный адрес операнда равен сумме значений базового регистра (BX или BP) и индексного регистра (SI или DI). Этот способ адресации реализуется при значениях поля адреса 0, 1, 2 или 3. Например:

```
mov AX,[BX+SI]
mov AX,[BX+DI]
mov AX,[BP+SI]
mov AX,[BP+DI]
```

- *Базово-индексная относительная адресация.* К сумме значений базового и индексного регистра прибавляется смещение (8- или 16-разрядное). Например:

```
mov AX, add[BX+SI]
mov AX, ddd[BX+DI]
mov AX, ddd[BP+DI]
```

Тут необходимо сделать несколько замечаний.

- Если *x* — смещение, то операнды `[x]` и `x` транслируются одинаково. Например, команды

```
mov AX, [ddd]
mov AX, ddd
```

имеют одинаковые коды, и, значит, действуют одинаково.

- Недопустима команда

```
mov AX, [BX+x]
```

Вместо этой команды можно применить команду

```
mov AX, x[BX]
```

или

```
mov AX, [BX+offset x]
```

- При непосредственном методе адресации не всегда ясно, сколько байт содержит операнд. В этом случае применяется модификатор `ptr`. Например:

```
mov byte ptr x, 255
mov byte ptr [DI+515],4
mov word ptr [DI+515],4
```

Я обещал вам пример, в котором мы рассмотрим способы адресации. Это позволит вам понять, каким образом можно использовать адресацию в программах. Давайте его напишем (листинг 2.1). В данной программе мы в сегменте кодов разместим две переменные `a` и `в`. Выведем на экран цифру 1, заданную в переменной `a`, затем перепишем значение 2 в `a` методом непосредственной адресации. Область данных, расположенную в сегменте кодов, мы обойдем с помощью команды `jmp` методом базовой адресации.

Листинг 2.1. Использование непосредственной и базовой адресации

```
code segment
assume CS:code, DS:code
start:
mov AX, code
; Устанавливаем регистр DS
mov DS,AX
; Выводим на экран '1'
mov AH, 9
mov DX, offset a
int 21h
; Копируем байт (непосредственная адресация)
mov AL, '2'
mov a, AL

; Обходим область переменных (базовая адресация)
lea BX, b+2
jmp BX
;---------- Переменные-------
a db '1', '$'
b db '2', '$'
; Выводим новое значение
lab1:
mov AH, 9
lea DX, a
int 21h
; Возврат в ОС
mov AX, 4C00h
int 21h
;---------------------------
code ends
end start
```

Данная программа должна быть понятна вам, т. к. все основные блоки мы уже рассмотрели ранее, но для удобства я разделил программу на блоки.

Для лучшей демонстрации методов я дам и другой вариант программы (листинг 2.2), использующий регистровую адресацию и для вывода на экран переменной `a` с значением `'1'`, и для обхода области данных.

Листинг 2.2. Использование регистровой адресации

```
code segment
assume CS:code, DS:code
start:
mov AX, code
; Устанавливаем регистр DS
mov DS, AX
; Выводим на экран '1'
mov AH, 9
mov DX, offset a
int 21h
; Копируем байт (регистровая адресация)
mov AL, [b]
mov a, AL
; Обходим данные (регистровая адресация)
mov BX, offset lab1
jmp BX
;---------- Переменные-------
a db '1', '$'
b db '2', '$'
c dw $
; Выводим новое значение
lab1:
mov AH, 9
lea DX, a
int 21h
; Возврат в ОС
mov AX, 4C00h
int 21h
;---------------------------
code ends
end start
```

Создавая исполняемый файл, вы должны помнить, что создаете COM-программу. Для этого наберите текст этих программ в любом текстовом редакторе. Сохраните текст в своем рабочем каталоге в файле с именем MY_COM.ASM. Далее запускаем трансляцию и сборку:

```
tasm my_com.asm
tlink /t my_com.obj
```

После этого, если нет ошибок, мы получаем исполняемый файл MY_COM.COM, который можно запускать на выполнение.

Работа со стеком

Вспомним, что стеком называется область памяти, предназначенная для временного хранения данных. Для этой области в программе выделяется сегмент объемом до 64 Кбайт, который называется *сегментом стека*. Для работы со стеком предназначены следующие регистры:

- SS — сегментный регистр стека;
- SP — регистр указателя стека.

Доступ к содержимому стека осуществляется также с помощью регистра BP.

Команды работы со стеком организованы в соответствии с принципом LIFO ("последним пришел — первым ушел"):

```
push операнд  ; Запись операнда в стек
pop операнд   ; Чтение операнда из стека
```

Команда `push` сначала вычитает из значения регистра SP число 2, уменьшая тем самым адрес начала стека на 2, а затем записывает операнд в начало стека. Команда `pop`, напротив, сначала считывает слово из начала стека и записывает его в *операнд*, а затем увеличивает значение регистра SP на 2.

Если операндом является двойное слово, то команда `push` вычитает из значения SP число 4, а команда `pop` увеличивает значение SP на 4.

Например, команды

```
push 100 ; Записать число 100 в стек
pop BX   ; Извлечь из стека
```

установят BX = 100.

При разработке подпрограмм рекомендуется в начале подпрограммы сохранять значения всех используемых в ней регистров в стеке, а в конце — восстанавливать.

Команда сохранения значений регистров AX, CX, DX, BX, SP, BP, SI, DI в стеке (значения сохраняются в указанном порядке):

```
pusha
```

Команда восстановления значений регистров AX, CX, DX, BX, SP, BP, SI, DI из стека:

```
popa
```

Слова, выбираемые из стека, размещаются в регистрах DI, SI, BP, *, BX, DX, CX, AX (звездочкой обозначено слово, которое игнорируется вместо того, чтобы быть помещенным в регистр SP).

Команды работы со стеком можно использовать для установки флагов:

```
; Запомнить значение регистра FL в стеке
pushf
; Извлечь слово из стека и записать его в регистр FL
popf
```

Команда `pushf` не изменяет разряды регистра FL, а `popf`, напротив, записывает в регистр FL новое слово и, значит, воздействует на все флаги.

Логика и организация программы

Вот мы и подошли к главному. Как мы знаем, программа на любом языке программирования, в том числе и на ассемблере, кроме команд процессора содержит специальные инструкции. Эти инструкции указывают самому ассемблеру, как организовать различные секции программы, как располагать данные и т. д. Здесь мы рассмотрим эти команды ассемблера, механизм передачи управления в программе и др.

Безусловный переход

Для начала давайте рассмотрим команду безусловного перехода:

```
jmp адрес
```

Эта команда передает управление команде, которая находится по указанному адресу. Адрес может быть указан с помощью смещения, состоящего:

- из одного байта (`short ptr`);
- из слова (`near ptr`);
- из двойного слова (`far ptr`).

Команда прибавляет к счетчику команд IP указанное смещение или устанавливает IP равным указанному двойному слову.

При *внутрисегментном переходе* предполагается, что содержимое регистра CS не изменяется, а изменяется регистр IP.

Для команды

```
jmp short ptr prog1
```

метка `prog1` отстоит от текущей команды на расстояние от –128 до 127 байт. Такой переход (в пределах 127 байт) называется *ближним (коротким) переходом*.

При переходе

```
jmp near ptr prog1
```

смещение `prog1` занимает *два байта* и метка `prog1` находится на расстоянии от –32 768 до 32 765 байт от текущей команды.

Для *межсегментного перехода* адрес (значение, на которое осуществляется переход) состоит из *четырех байт*, два из которых составляют смещение, и два — значение сегментной составляющей адреса. Переход между сегментами или переход более чем на 127 байт называется *дальним (длинным) переходом*. Для команды

```
jmp far ptr prog1
```

метка `prog1` может находиться *в другом сегменте*. В этом случае эта метка должна быть определена как

```
prog1 label far
```

Рассмотрим это на примере (листинг 2.3).

Листинг 2.3. Использование межсегментного перехода

```
code segment
assume CS:code
start:
       jmp far ptr lab2
lab1 label far
       mov AX, 4C00h
       int 21h
code ends
code2 segment
assume CS:code2
lab2 label far
       mov AH, 10h
```

```
        int 16h
        jmp far ptr lab1
code2 ends
end start
```

В этой программе выполняется переход во второй сегмент, где программа ожидает ввода символа с клавиатуры, после чего возвращается в первый сегмент.

Команды условного перехода

Передача управления указанной команде может осуществляться в том случае, когда выполняется какое-нибудь условие. Например, если у нас в программе есть запрос пароля или серийного номера, то после его ввода пользователем программа проверяет пароль: если он верный, то выполняется одна ветка программы, если неверный, то другая. Выполнение условия зависит от того, установлены или нет определяющие это условие биты регистра флагов FL. Такие команды передачи управления называются *командами условного перехода*. Например, после команды

```
jne lab1
```

управление передается на метку `lab1` в том и только том случае, когда флаг ZF равен 0. Если же ZF = 1, то выполняется команда, следующая за командой перехода.

Мнемоническое обозначение `jne` происходит от слов "jump not equal" ("перейти, если не равно"). Имеется в виду, что перед командой перехода выполнялась команда сравнения двух операндов, и переход надо осуществлять, если эти два операнда не равны.

Таким образом, обычно флаги устанавливаются с помощью команды сравнения:

`cmp` *операнд1*`,` *операнд2*

действие которой аналогично действию команды вычитания

`sub` *операнд1*`,` *операнд2*

за исключением того, что команда `cmp` не записывает результат вычитания в первый операнд.

Например, команды

```
cmp AX, BX
jne lab1
```

осуществляют передачу управления на метку `lab1` в случае, если значения регистров AX и BX не равны.

В следующем примере вместо команды

```
mov DX, offset mes
```

мы применяем команду записи адреса строки с именем `mes` в регистр DX:

```
lea DX, mes
```

Эти команды действуют аналогично, и программист уже сам отдает предпочтение тому или иному способу записи.

Команды сравнения

В действительности микропроцессор имеет 18 команд условного перехода. Переходы могут осуществляться в зависимости от состояния флагов ZF, SF, CF, OF, PF (но не AF).

Для того чтобы делать переходы на основании выполнения отношений равно, меньше, не меньше, больше, не больше и других, применяется команда `cmp`.

В левом столбце табл. 2.4 записаны мнемоники команд перехода. В среднем столбце — условия, которые были определены с помощью команды `cmp` или другой команды, предшествующей команде перехода. В последнем столбце — значения флагов, при которых команда перехода передаст управление на смещение, указанное операндом.

В командах условного перехода операнды состоят из одного или двух байт. Управление между сегментами не передается.

Флаги CF и ZF очищаются командой сравнения, когда первый операнд больше второго, если рассматривать операнды как беззнаковые целые числа. В этом случае говорят, что первый операнд *выше* второго. В случае, когда первый операнд меньше второго, если их рассматривать как беззнаковые целые числа, говорят, что первый операнд *ниже* второго. Например: число –1 будет меньше нуля, но это число будет выше нуля, ибо рассматриваемое как беззнаковое (0FFFFh), оно равно 65 535.

При операциях сравнения не стоит забывать и о таких операциях, как `add`, `sub`, которые мы рассмотрим немного позднее.

Таблица 2.4. *Команды перехода*

Команда	Отношение между операндами команды `cmp` или свойства результата операции	Значения флагов
`ja/jnbe`	Выше	CF = 0 и ZF = 0
`jae/jnb`	Выше или равно/не ниже/не перенос	CF = 0
`jb/jnae/jc`	Ниже/перенос	CF = 1

Таблица 2.4 *(окончание)*

Команда	Отношение между операндами команды `cmp` или свойства результата операции	Значения флагов
`jbe/jna`	Ниже или равно/не выше	CF = 1 или ZF = 1
`jcxz/jecxz`	CX = 0/ECX = 0	Не влияет на флаги
`je/jz`	Равно/ноль	ZF = 1
`jg/jnle`	Больше	ZF = 0 и SF = OF
`jge/jnl`	Больше или равно/не меньше	SF = OF
`jl/jnge`	Меньше	SF ≠ OF
`jle/jng`	Меньше или равно/не больше	ZF = 1 или SF ≠ OF
`jne/jnz`	Не равно/не ноль	ZF = 0
`jno`	Не переполнение	OF = 0
`jnp/jpo`	Не паритет/паритет нечетный	PF = 0
`jns`	Не знак	SF = 0
`jp/jpe`	Паритет/паритет четный	PF = 1
`jo`	Переполнение	OF = 1
`js`	Знак	SF = 1

Команды цикла

Для организации циклов применяется команда цикла

`loop` *`метка`*

и ее разновидности.

Эта команда равносильна последовательности, состоящей из двух команд:

```
; CX = CX-1
dec CX
jnz short ptr метка
```

Стало быть, ее операнд имеет размер не более одного байта; *`метка`* лежит в пределах от –128 до +127 байт от команды `loop`.

Команда `loop` работает следующим образом:

1. Значение регистра CX уменьшается на 1.
2. Если CX ≠ 0, то управление передается на метку, иначе выполняется следующий за командой `loop` оператор.

Например, в случае команд

```
  mov CX, 0
lab1: add AX, 1
  loop lab1
```

при первом выполнении команды `loop` сначала значение регистра CX будет уменьшено на 1, а потом его содержимое будет сравниваться с нулем. Поскольку значение CX станет равным 65 535, произойдет переход на `lab1`. Затем значение регистра AX будет увеличено на 1, а значение CX станет равным 65 534. Цикл будет повторяться до тех пор, пока значение CX не станет равным 0. Следовательно, команда `add AX, 1` будет выполнена 65 536 раз.

Давайте напишем программу, выводящую на экран все символы, коды ASCII которых лежат в диапазоне от 21h = 33 до 33 + 93 = 126 (листинг 2.4).

Листинг 2.4. Вывод ASCII-символов с кодами от 33 до 126

```
code segment
assume CS:code
; Число символов
start: mov CX, 94
; Код первого символа
       mov DL, 21h
; Вывод символа на экран
next: mov AH, 2
       int 21h
; Следующий код
       add DL, 1
       loop next
; Выход
       mov AX, 4c00h
       int 21h
code ends
end start
```

Есть несколько разновидностей команды `loop`.

Следующие команды имеют одинаковые коды:

`loope` *метка*

`loopz` *метка*

Переход на метку осуществляется, если значение регистра CX $\neq 0$ и значение флага ZF = 1.

Команды

```
loopne метка
loopnz метка
```

тоже имеют одинаковые коды и, значит, действуют одинаково. В них переход на метку осуществляется, если CX ≠ 0 и ZF = 0.

Во всех этих случаях метка имеет атрибут `short ptr`.

Логические операции и команды сдвига

Логические операции используются для сброса и установки битов и для арифметических операций с кодами ASCII.

Команды логических операций `and`, `or`, `xor`, `not`, `test` обрабатывают байт или слово, в регистре или в памяти, и воздействуют на флаги CF, OF, а остальные флаги зависят от результата операции.

В командах:

```
and операнд1, операнд2  ; Поразрядное логическое умножение (И)
or операнд1, операнд2   ; Поразрядное логическое сложение (ИЛИ)
cor операнд1, операнд2  ; Поразрядное сложение по модулю 2
not операнд             ; Поразрядное логическое НЕ
```

результат выполнения операции записывается в первый операнд.

Команда

```
test
```

действует подобно команде `and` кроме того, что результат выполнения поразрядного логического умножения не записывается на место первого операнда.

Разберем это на примерах.

Пусть значение регистра AL равно 10101010b. Тогда команда

```
and AL, 00101111b
```

запишет в AL число 00101010b и даст значения флагов CF = 0, OF = 0, PF = 0, SF = 0, ZF = 0.

Команда

```
or AL, 00101111b
```

запишет в AL число 10101111b и даст значения флагов CF = 0, OF = 0, PF = 1, SF = 1, ZF = 0.

Команда

```
xor AL, 00101111b
```

запишет в AL число 10000101b и даст значения флагов CF = 0, OF = 0, PF = 1, SF = 1, ZF = 0.

Команда

```
xor AL, 00101111b
```

запишет в AL число 10000101b и даст значения флагов CF = 0, OF = 0, PF = 0, SF = 1, ZF = 0.

Команда

```
not AL
```

изменит значение AL на 01010101b и даст значения флагов CF = 0, OF = 0, PF = 1, SF = 0, ZF = 0.

А команда

```
test AL, 00101111b
```

не изменит значение AL, но установит флаги так же, как команда `and`.

Команды сдвига и циклического сдвига имеют следующий формат:

код *операнд1*, *счетчик*

где *счетчик* равен 1 или CL. Действие команд сдвига показано на рис. 2.9. Операндом команды сдвига является байт или слово.

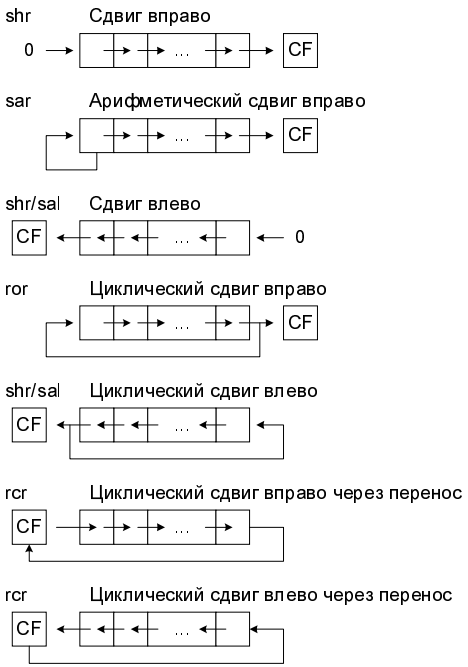

Рис. 2.9. Команды сдвига

При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита. Например, пусть AL = BL = 10001000b. Тогда команды

```
shr AL, 1
sur BL, 1
```

запишут в CF значение 0, преобразуют AL в число 01000100b, а BL — в число 11000100b.

При тех же условиях команды

```
mov CL, 5
rol AL, CL
```

сдвигают содержимое AL влево 5 раз, перенося каждый раз старший бит в младший. Флаг CF последовательно принимает значения 1, 0, 0, 0, 1. В результате AL = 00010001b.

Команда сдвига влево используется для умножения операнда на 2. Рассмотрим следующие команды для умножения на 2 числа, состоящего из 32 бит и находящегося в двух регистрах.

Пусть число находится в регистровой паре DX:AX так, что старшие 16 бит находятся в DX, а младшие — в AX. Следующие команды

```
shl AX, 1
rcl DX, 1
```

умножают это число на 2. Здесь вторая команда переносит бит CF в младший бит регистра DX, после того как бит CF был извлечен из старшего бита регистра AX.

Применим команду сдвига влево для вывода слова в двоичном виде. Напишем программу, содержащую команды, выводящие на экран двоичное представление регистра BX (листинг 2.5).

Листинг 2.5. Вывод значения BX в двоичном представлении

```
code segment
assume CS:code, DS:code
; Произвольное число
       mov BX, 101001001b
; Вывести 16 бит
       mov CX, 16
; ASCII-код нуля
lp1: mov DL, '0'
; Сдвиг влево
```

```
        shl BX, 1
; Если старший бит был равен 0
        jnc lp2
; иначе DL = '1'
        add DL, 1
; Вывод символа из DL на экран
lp2: mov AH, 2
        int 21h
        loop lp1
; Выход
        mov AX, 4c00h
        int 21h
code ends
end
```

В результате на экран будет выведено число 0000000101001001. В данной программе для вывода символа из DL была применена функция 2 прерывания 21h.

Организация циклов

Команда `loop` позволяет организовывать циклы, состоящие из небольшого количества команд. В общем случае в начале или внутри цикла проверяется некоторое условие, и если оно не выполнено, производится выход из цикла. В конце цикла ставится оператор перехода `jmp` на начало цикла.

Давайте разберем это на примере. В следующей программе с клавиатуры вводится шестнадцатеричное число и записывается в регистр BX (листинг 2.6).

Листинг 2.6. Запись шестнадцатеричного числа в регистр BX

```
code segment
        assume  CS:code
start:
; Очистить BX
        mov BX, 0
input:
; Ввод символа с клавиатуры в AL
        mov AH, 1
        int 21h
```

```
; Переход, если AL ниже, чем '0'
       cmp AL, '0'
       jb atof
; Переход, если AL выше, чем '9'
       cmp AL, '9'
       ja atof
; Превратить код цифры в значение
       sub AL, '0'
digit:
       shl BX, 1
; Умножение BX на 16
       shl BX, 1
       shl BX, 1
       shl BX, 1
       xor AH, AH
; BX = 16 BX + AL
       add BX, AX
       jmp input
; Если введенный символ не является
; цифрой или символом a, b, c, d, e, f, то выход из цикла
atof: cmp AL, 'a'
       jb exit
       cmp AL, 'f'
       ja exit
; AL = (AL - 'a') + 10
       sub AL, 'a'-10
       jmp digit
; Выход
exit: mov AX, 4c00h
       int 21h
code ends
end start
```

Здесь символ с клавиатуры принимается в AL с помощью функции 1 прерывания 21h.

Вызов подпрограмм

Любую сложную или большую программу необходимо как-то упростить. Самое простое, что можно придумать, это разбить программу на маленькие

программки или блоки. Эти блоки программисты называют *подпрограммами*. Например, программу, которую мы рассматривали, можно разбить на блоки (листинги 2.7—2.10).

Листинг 2.7. Работа с подпрограммами

```
code segment
assume CS:code, DS:data
mov AX, data
mov DS, AX
;-----
call pod1
;-----
call pod2
;-----
call pod3
code ends
;-----
; Описываем сегмент данных
data segment
; Строка, которую будем выводить на экран
mes db 'Hello, world!', 13, 10, '$'
data ends
end
```

Листинг 2.8. Подпрограмма вывода сообщения на экран

```
pod1 proc
; Вывод сообщения на экран
mov DX, offset mes
mov AH, 9
int 21h
ret
pod1 endp
```

Листинг 2.9. Подпрограмма нажатия любой клавиши

```
pod2 proc
; Ожидание нажатия любой клавиши
mov AH, 10h
```

```
int 16h
ret
pod2 endp
```

Листинг 2.10. Подпрограмма, организующая выход в ОС

```
pod3 proc
; Выход из программы
mov AX, 4c00h
int 21h
ret
pod3 endp
```

Таким образом, любую программу можно организовать как набор подпрограмм. Среди этих подпрограмм выделяется главная подпрограмма, на которую передает управление загрузчик. Главная подпрограмма в процессе своей работы вызывает с помощью команды `call` подпрограммы, которые, в свою очередь, возвращают ей управление с помощью команды `ret` (это стоит помнить всегда). Неважно, на каком языке написана программа, насколько она сложна, — в ней всегда есть подпрограммы. Конечно, если программа написана на языке высокого уровня (Delphi, VB, Builder), то там можно использовать такие вещи, как вызываемые внешние функции (плагины — plug-ins), которые можно оформить в виде отдельных библиотек. Но все равно в ассемблерном коде это будет выглядеть как вызов `call ... ret`.

Команды *call* и *ret*

Команда `call` выполняет вызов процедуры, адрес которой может задаваться смещением, либо сегментом и смещением. В последнем случае вызов называется *межсегментным*.

Алгоритм работы команды `call`:

1. Если вызов межсегментный, то значение CS запоминается в стеке и регистр CS устанавливается в значение, равное адресу сегмента подпрограммы.
2. Значение счетчика команд IP запоминается в стеке и регистр IP устанавливается в значение, равное смещению операнда.

Примечание

Помните, что при записи в стек значение SP предварительно уменьшается на 2.

Команда возврата `ret` просто извлекает из стека значение регистра IP. Если возврат (переход) длинный, то применяется команда `retf`. В этом случае после извлечения IP производится извлечение CS. Команда может иметь операнд:

```
ret cnt ; cnt - произвольная константа
```

После возврата из процедуры значение SP увеличивается на `cnt`.

Подпрограммы

Программа разбивается на небольшие модули, состоящие из групп команд, которые называются *процедурами* или *подпрограммами* и записываются следующим образом:

```
имя1 proc
... ; группа команд
ret
имя1 endp
имя2 proc far
... ; группа команд
retf
имя2 endp
```

Здесь `имя1` и `имя2` — произвольные имена, `proc` — директива, обозначающая начало процедуры, `endp` — директива, обозначающая конец процедуры. Если к директиве `proc` добавляется слово `far`, то подпрограмма имеет дальний адрес. Если указано слово `near`, то процедура должна находиться в одном сегменте с вызывающей процедурой. Слово `near` может быть опущено, ибо оно подразумевается по умолчанию.

Напишем программу, состоящую из главного модуля и подпрограммы, выводящей символ из регистра DL на экран (листинг 2.11).

Листинг 2.11. Вывод на экран символа из регистра DL

```
mcode segment
    assume CS:mcode
main  proc far
; Для возврата в операционную систему
    push ES
    mov AX, 0
    push AX
    mov DL, '1'
```

```
; Вывод '1'
    call display
    mov DL, '2'
; Вывод '2'
    call display
    mov DL, '3'
; Вывод '3'
    call display
; Выход в MS-DOS
    ret
main endp
display proc
    mov AH, 2
    int 21h
    ret
display endp
mcode ends
; На главную подпрограмму
    end main
```

Здесь выход в MS-DOS осуществляется с помощью команды `ret`, которая передает управление на команду с адресом ES:0 (по этому адресу находится команда `int 20h` — вызов прерывания, производящего выход).

Внешние подпрограммы

Подпрограммы называются *внешними*, если их объектные модули находятся в отдельных файлах или библиотеках. Для вызова внешней подпрограммы необходимо описать эту подпрограмму с помощью одной из директив:

```
extrn имя:far
extrn имя:near
```

Доступ к подпрограмме или к другим данным вызываемого модуля обеспечивается директивой:

```
public имя
```

Рассмотрим пример, на котором я покажу, как можно организовать компиляцию и сборку программы из нескольких файлов.

Главный модуль и вызываемую подпрограмму, рассмотренные ранее, мы запишем в два файла. Добавим вызываемую подпрограмму `keybin` ввода символа с клавиатуры. Получим текст листинга 2.12.

Листинг 2.12. Первый файл — DISPL.ASM

```
mcode segment
    assume CS:mcode
extrn display:near
extrn keybin:near
; Модуль mylib.obj
go: mov DL, 'a'
    call display
    mov DL, 'b'
    call display
    mov DL, 'c'
    call display
    call keybin
    ret
mcode ends
    end go
```

Вызываемые подпрограммы запишем во второй файл — MYLIB.ASM (листинг 2.13).

Листинг 2.13. Второй файл — MYLIB.ASM

```
subr segment
public display, keybin
    assume CS:subr
; Подпрограммы
display proc
    mov AH, 2
    int 21h
    ret
display endp
keybin proc near
    mov AH, 1
    int 21h
    ret
keybin endp
subr ends
    end
```

Компиляция и сборка выполняются после ввода команд

```
tasm displ
tasm mylib
tlink mylib+displ
```

В результате будет создан загрузочный модуль DISPL.EXE, который после загрузки выведет символы `abc` и после нажатия клавиш <Ctrl>+<C> закончит работу.

Структура COM- и EXE-программ. Размещение программы в памяти

Надо понимать, что готовый для выполнения файл называется *загрузочным модулем*. Он может быть построен несколькими способами и в зависимости от способа построения имеет расширение exe или com (формат COM использовался в MS-DOS).

Для выполнения программы пользователя операционная система (точнее, загрузчик COMMAND.COM) размещает коды программы в свободной оперативной памяти и добавляет перед программой состоящий из 256 байт блок, который называется *префиксом программного сегмента (PSP))*. После его размещения операционная система передает управление в точку входа этой программы (определенную как операнд директивы `end`).

Операционная система при этом выполняет следующие действия:

1. Определяет сегментный адрес префикса программного сегмента (PSP).
2. Создает два блока памяти — блок памяти для переменных среды и блок памяти для PSP и программы.
3. Помещает в блок памяти переменных среды путь, откуда была загружена программа.
4. Заполняет поля PSP.

Загрузочный модуль размещается в памяти следующим образом:

1. PSP (256 байт).
2. Код программы.
3. Данные программы.
4. Неинициализируемые данные.

Стек размещается в конце сегмента стека (который в случае COM-программы совпадает с сегментом кодов).

Регистры DS и ES после размещения программы пользователя указывают на PSP.

Загрузочный модуль как файл может существовать в одном из двух форматов — EXE и COM. Формат EXE отличается от COM тем, что он содержит специальный начальный блок, который называется *заголовком* и занимает не менее 512 байт (пока речь идет об ОС MS-DOS).

При загрузке программы значения регистров DS и ES указывают на PSP и запоминаются поля PSP, но дальнейшие действия зависят от формата файла.

EXE-файл загружается, начиная с адреса PSP:100h. В момент загрузки EXE-файла регистры CS, IP, SS и SP инициализируются значениями, указанными в заголовке.

Регистры DS и ES указывают на PSP. CS устанавливается на начало сегмента кодов.

COM-файл состоит из единственного сегмента. Образ COM-файла размещается начиная с адреса PSP:100h. Регистры CS и SS указывают на PSP, регистр SP — на конец сегмента PSP (т. е. SP = 0FFFEh), значение регистра IP равно 100h.

В первых байтах PSP записывается команда `int 20h` (выход в операционную систему). Стало быть, если выполнить в начале EXE-программы команды

```
push ES
mov AX, 0
push AX
```

то в случае, если встретится команда `ret`, произойдет переход по адресу ES:0 на команду `int 20h`. Если же вместо `ret` использовать для возврата в операционную систему команду `int 20h` непосредственно в программе, то, поскольку сегмент вызывающей программы будет отличаться от PSP, адрес возврата будет вычислен некорректно.

В COM-программе эти шаги инициализации регистра DS и записи нулевого слова в стек не нужны. Эти действия производит загрузчик. Выход можно осуществлять с помощью команды `ret`. Так как адресация начинается со смещения 100h от начала PSP, COM-программа должна содержать директиву `org 100h` (после оператора `segment`).

Основное отличие формата COM от EXE заключается в том, что размер COM-программы не превышает 64 Кбайт и все данные, коды команд и область стека в ней размещаются в одном сегменте.

Сейчас в мире компьютерных форматов произошли сильные изменения. Например, для ОС Windows используется другой формат файлов (но это уже выходит за рамки данной книги).

Ввод и вывод данных

Ввод и вывод выполняются с помощью команд программных прерываний (MS-DOS) и системных функций (Windows), вызывающих необходимые подпрограммы.

Определение данных

Директивы ассемблера, позволяющие выделить память для числовых переменных и констант, приведены в табл. 2.5. Как и в случае других директив распределения памяти, ассемблер связывает тип с любой переменной, определенной с помощью этих директив. Размер переменной каждого типа определяется в байтах.

Директивы распределения памяти могут иметь соответствующие типу переменных преобразователи — `word ptr`, `qword ptr` и `tbyte ptr`.

Начальное значение для числовых констант можно получить несколькими способами. Двоичные целые константы можно определить как битовые строки, десятичные целые, восьмеричные целые или шестнадцатеричные строки. Упакованные десятичные величины обычно записываются как десятичные целые, хотя ассемблер будет принимать и обрабатывать и другие представления целых. Вещественные значения можно записывать как обычные вещественные десятичные числа (с десятичной точкой), как десятичные числа в научной нотации или как шестнадцатеричные строки. В табл. 2.5 дан пример, показывающий несколько способов присваивания данным различных числовых типов одной и той же начальной величины.

Таблица 2.5. *Директивы распределения памяти*

Директива	Интерпретация	Тип данных
`db`	Определить байт	Целое (байт)
`dw`	Определить слово	Целое (слово)
`dd`	Определить двойное слово	Короткое целое, короткое вещественное
`dq`	Определить слово длиной 8 байт	Длинное целое, длинное вещественное
`dt`	Определить слово длиной 10 байт	Упакованное десятичное, временное вещественное

Директивы определения данных

Общий формат директивы определения данных (см. табл. 2.5):

```
[имя] dn выражение
```

где

- задавать имя элемента данных необязательно;
- `dn` принимает одно из следующих символьных значений: `db` (байт), `dw` (слово), `dd` (двойное слово), `dq` (учетверенное слово), `dt` (десять байт);
- *выражение* может быть константой или знаком вопроса для неопределенного значения; оно может содержать несколько констант, разделенных запятыми или заданных с числом повторений.

Примеры:

```
; Массив x из 4 байт
x db 10, 12, 30, 40
; Массив y из 10 слов
y dw 10 dup(?)
; Массив z из 10 нулей
z dw 10 dup(0)
; Массив text из 4 байт
text db 'abcd'
```

Обращение к элементам массива осуществляется с помощью смещения относительно начального адреса массива. Например, команда

```
mov AL, x+2
```

запишет в предыдущем примере байт со значением 30 в регистр AL.

Знак `$` обозначает текущий адрес (его еще называют *счетчик адреса*). Особенность его в том, что когда транслятор встречает в программе на ассемблере этот символ, то подставляет вместо него текущее значение счетчика адреса. Постарайтесь запомнить это. Этот операнд очень часто используется в ассемблере. Например:

```
; По адресу adr записано смещение adr
adr dw $
```

или

```
; Эта команда вызывает переход на саму себя, т. е. бесконечный цикл
jmp $
```

Константы

Символьная константа задается в апострофах. Например: 'a', 'и', '1', '9'.

Рассмотрим числовые константы. Как вы знаете, в программах на любом языке программирования (и на ассемблере в том числе) могут использоваться числовые константы в разнообразных системах счисления. Наиболее распространены двоичная, восьмеричная, шестнадцатеричная и десятичная системы счисления. В самой программе они определяются следующим образом:

- двоичная константа — в конец выражения добавляется буква b, например: 10b, 10010b, 10000110b, 101110111000b;
- восьмеричная константа — в конец выражения добавляется буква o, например: 12o, 177o, 3650o;
- десятичная константа — в конец выражения добавляется буква d или ничего, например: 10, 14, 134, 143;
- шестнадцатеричная константа — начинается с цифры (иначе она будет принята за идентификатор), справа добавляется буква h, например: 0FFh, 123h.

Для перевода чисел из одной системы в другую есть специальные правила. Когда-то без этих правил не мог обойтись ни один программист. Сегодня ситуация не сильно изменилась. Конечно, любой профессиональный программист должен уметь переводить числа между этими системами, но у начинающих обычно возникает вопрос — а зачем это вообще нужно? Как вы знаете, компьютер при всех своих многообразных умениях "понимает" только машинные команды, т. е. команды самого процессора. Мы привыкли считать в десятичной системе счисления. Для этого мы используем цифры 0—9. Если компьютеру написать "Посчитай мне 2+2", он этого не поймет. Ему надо сказать это на его языке, т. е. на языке двоичных данных, или в двоичной системе счисления. Двоичная система счисления использует всего две цифры — 0 и 1. Например, десятичному числу 2 в двоичной системе соответствует число 10 (читается "один-ноль"). Программистам приходится оперировать и большими числами. Представим, например, число 40 000 в двоичной системе счисления: 1001110001000000b. Удобно ли работать с такими числами? Уверен, что не очень. Для удобства придумали шестнадцатеричную систему счисления. Десятичное число 40 000 соответствует шестнадцатеричному числу 9C40h. Как видите, эта запись намного короче, чем двоичная. Аналогично, в восьмеричной системе число 40 000 записывается как 116100o.

Что касается правил перевода чисел из одной системы в другую, то вам, я думаю, они не понадобятся. Есть специальные средства, позволяющие

переводить числовые константы из одной системы в другую значительно быстрее и с гарантированным отсутствием ошибок. Самое простое из них — любой инженерный канкулятор, например, встроенный в Windows.

Директива *equ*

Директива `equ` определяет имя для константы. Например, команда:

```
cnt equ 10
```

определяет константу `cnt` со значением 10. Теперь вместо `cnt` повсюду в программе будет подставляться число 10.

Команды пересылки и строковые операции

Рассмотрим команды пересылки данных. С их помощью можно выполнять операции пересылки, загрузки, записи, сравнения, сканирования строк. Также мы рассмотрим применение этих операций и команд коррекции для арифметических действий над числами, представленными в виде строк ASCII и в формате BCD.

Для обработки строковых данных имеются пять команд:

- `movs` — переписать один байт или одно слово (двойное слово) из одной области памяти в другую;
- `lods` — загрузить из памяти один байт или одно слово (двойное слово) в регистр-аккумулятор;
- `stos` — записать содержимое аккумулятора в память;
- `cmps` — сравнить содержимое двух байт или двух слов (двойных слов), расположенных в памяти;
- `scas` — сравнить содержимое аккумулятора с одним байтом или с одним словом (двойным словом), расположенным в памяти.

Примечание

Для обработки строк любой длины применяется префикс `rep`. Благодаря этому префиксу мы можем одной командой переслать до 64 Кбайт данных (если размер адреса в сегменте указан как `use16`, т. е. 16 бит) или до 4 Гбайт данных (если размер адреса в сегменте указан как `use32`, т. е. 32 бита). Подробнее мы рассмотрим его в *разд. "Команда `rep`" далее*.

Для этих команд операнды задаются неявно. В этих командах участвуют:

- регистры DS, ES, SI и DI;
- флаг DF.

Флаг DF задает направление, в котором мы будем обрабатывать элементы строки — по возрастанию или по убыванию адресов. Установить нулевое значение флага можно с помощью команды сброса флага направления:

```
cld
```

Выражением [DS:SI] мы обозначим содержимое байта (или слова), адрес которого задается сегментным регистром DS и индексным регистром SI, — отсюда мы будем переписывать данные. Аналогично, [ES:DI] обозначает содержимое байта, адрес которого определяется регистрами ES и DI — сюда мы будем переписывать данные.

Команда *movs*

Назначение этой команды — пересылка элементов одной строки в другую строку. Разновидности команды — `movsb` (пересылка байтов), `movsw` (пересылка слов), `movsd` (пересылка двойных слов).

Команда `movsb` работает так:

1. Байт [DS:SI] переписывается в [ES:DI].
2. Если флаг DF равен 0, то значения SI и DI увеличиваются на 1, в противном случае значения обоих регистров SI и DI уменьшаются на 1.

Команды `movsw` и `movsd` отличаются от `movsb` тем, что на шаге 2 значения обоих регистров SI и DI изменяются соответственно на 2 и 4.

Команда *lods*

Команда `lods` загружает элемент строки в регистр-аккумулятор. Разновидности команды — `lodsb` (загрузка байта в регистр AL), `lodsw` (загрузка слова в регистр AX), `lodsd` (загрузка двойного слова в регистр EAX).

Команда `lodsb` работает так:

1. Значение ячейки [DS:SI] переписывается в регистр AL.
2. Если флаг DF равен 0, то значение регистра SI увеличивается на 1, а если DF = 1, то значение регистра SI уменьшается на 1.

Команда `lodsw` отличается тем, что на шаге 1 в регистр AX переписывается содержимое слова [DS:SI], а на шаге 2 значение регистра SI увеличивается или уменьшается на 2.

Команда `lodsd` отличается тем, что на шаге 1 в регистр EAX переписывается содержимое двойного слова [DS:SI], а на шаге 2 значение регистра SI увеличивается или уменьшается на 4.

Команда *stos*

Команда `stos` сохраняет элемент из регистра-аккумулятора в строке. Разновидности команды — `stosb` (сохранение байта из регистра AL), `stows` (сохранение слова из регистра AX), `stosd` (сохранение двойного слова из регистра EAX).

Команда `stosb` работает так:

1. Значение регистра AL переписывается в ячейку [ES:DI].
2. Если флаг DF равен 0, то значение регистра DI увеличивается на 1, в случае DF = 1 значение регистра DI уменьшается на 1.

Команда `stosw` отличается тем, что на шаге 1 в [ES:DI] переписывается значение регистра AX, а на шаге 2 значение регистра DI увеличивается или уменьшается на 2.

Команда `stosd` отличается тем, что на шаге 1 в [ES:DI] переписывается значение регистра EAX, а на шаге 2 значение регистра DI увеличивается или уменьшается на 4.

Команда *cmps*

Команда `cmps` предназначена для сравнения элементов двух строк, расположенных в памяти. Разновидности команды — `cmpsb` (сравнение строк байтов), `cmpsw` (сравнение строк слов), `cmpsd` (сравнение строк двойных слов).

Команда `cmpsb` работает так:

1. Сравнивает байты [DS:SI] и [ES:DI] и устанавливает флаги AF, CF, OF, PF, SF и ZF, как команда вычитания [DS:SI] – [ES:DI].
2. Если DF = 0, то значения регистров SI и DI увеличиваются на 1, иначе уменьшаются на 1.

Команда `cmpsw` отличается тем, что сравниваются слова, и на шаге 2 значения регистров SI и DI увеличиваются или уменьшаются на 2.

Команда `cmpsd` отличается тем, что сравниваются двойные слова, и на шаге 2 значения регистров SI и DI увеличиваются или уменьшаются на 4.

Команда *scas*

Команда `scas` предназначена для выполнения поиска значения (символа) в строке, расположенной в памяти, путем сравнения ее элементов с содержимым регистра (сканирование). Разновидности команды — `scasb` (сравнение байта), `scasw` (поиск слова), `scasd` (поиск двойного слова).

Команда `scasb` работает так:

1. Сравнивает содержимое регистра AL с байтом [ES:DI]; действует на флаги AF, CF, OF, PF, SF, ZF.
2. Если DF = 0, то значение регистра DI увеличивается на 1, иначе DI уменьшается на 1.

Команда `scasw` сравнивает значение регистра AX со словом [ES:DI] и увеличивает или уменьшает значение DI на 2.

Команда `scasd` сравнивает значение регистра EAX с двойным словом [ES:DI] и увеличивает или уменьшает значение DI на 4.

Команда *rep*

Теперь рассмотрим обещанную команду, без которой мы не сможем эффективно работать со строками, — это префикс `rep`, или как его еще называют *префикс повторения*. Если поставить перед строковой операцией префикс `rep`, то строковая операция будет повторяться, пока значение регистра ECX/CX не станет равным 0. Значение регистра CX уменьшается на 1 после выполнения операции. Например, после выполнения команд

```
      lea      SI, text1
      lea      DI, text2
      cld
      mov      CX, 30
rep   movsb
```

30 байт строки `text1` копируются в строку `text2` (предполагается, что строка `text1` находится в сегменте, на который указывает регистр DS, а строка `text2` — в сегменте, на который указывает регистр ES).

Команда `rep` предписывает повторять циклическую операцию до тех пор, пока значение регистра CX не равно нулю. Разновидности команды:

- `repe` — повторять, пока равно;
- `repz` — повторять, пока ноль;
- `repne` — повторять, пока не равно;
- `repnz` — повторять, пока не ноль.

Вот это очень важно! Лучше всего мы это рассмотрим на примере сравнения двух строк (листинг 2.14).

Листинг 2.14. Сравнение строк

```
; Это находится в сегменте данных:
; Первая строка
text1 db 'aBCDefg'
; Определение длины строки для сравнения
len_text1 db  $-text1
; Вторая строка
text2 db 'abcefgh'
; Здесь расположены другие команды и данные
...
; Это находится в сегменте кодов:
; Сами команды сравнения
cld
; Устанавливаем SI на начало строки text1
lea SI, text1
; Устанавливаем DI на начало строки text2
lea DI, text2
; Помещаем в CX длину строки (число символов для сравнения)
mov CX, len_text1
; Сравнивать, пока элементы равны
repe cmpsb
je LabeL1
LabeL2:
```

Вот что у нас получается:

Регистр SI

text1 =	'a'	'b'	'c'	'd'	...

text2 =	'a'	'b'	'c'	'e'	...

Регистр DI

Как видите, первые три символа равны, поэтому значение флага ZF = 1. Четвертые символы строк различаются, поэтому ZF = 0, и выполнение цикла прекращается. Поскольку после цикла значение ZF не равно 1, мы переходим на метку `LabeL2`. Если бы все элементы были равны, то значение флага ZF было бы равно 1, и мы перешли бы на метку `LabeL1`.

Рассмотрим применение строковой операции `stosb` для отображения на экране российского флага. В следующей программе (листинг 2.15) в ES запи-

сывается адрес видеопамяти, а в регистр DI — начальное смещение 0. Затем с помощью команд

```
      mov     CX, 19200
      mov     AL, 15
rep   stosb
```

в видеопамять по адресам A0000h, A0001h, …, A0000h + 19199 записываются байты, значения которых равны 15. Эти байты соответствуют 60 строкам экрана, окрашенным в белый цвет.

Затем, команды

```
      mov     CX, 19200
      mov     AL, 1
rep   stosb
```

окрашивают следующие 60 строк в синий цвет, а команды

```
      mov     CX, 19200
      mov     AL, 4
rep   stosb
```

окрашивают последние 60 строк в красный.

Как видите, все это не очень сложно.

Листинг 2.15. Рисование флага на экране

```
codesg      segment
assume      CS:codesg, DS:codesg, SS:codesg
org         100h
main        proc      near
mov         AH, 0fh
int         10h
push        AX
mov         AX, 0013h
int         10h
mov         AX, 0A000h
mov         ES, AX
mov         DI, 0
mov         CX, 19200
mov         AL, 15
rep         stosb
mov         CX, 19200
```

```
mov         AL, 1
rep         stosb
mov         CX, 19200
mov         AL, 4
rep         stosb
mov         AX, 0
int         16h
pop         AX
mov         AH, 0
int         10h
ret
main        endp
codesg      ends
end         main
```

Давайте напишем подпрограмму поиска символа `'D'` в строке, изменяющейся в процессе работы нашей программы (листинг 2.16), которая определена в сегменте данных следующим образом:

```
text1      db      10 dup(' ');
```

Надо только запомнить, что при вызове подпрограммы регистр ES установлен на сегмент данных.

Листинг 2.16. Подпрограмма поиска символа `'D'` в строке

```
find      proc
          cld
          mov      CX, 10
          lea      DI, text1
; Символ 'D' не найден
          mov      AH, 0
          mov      AL, 'D'
repne     scasb
          jne      loop1
; Символ 'D' найден
          mov      AH, 1
loop1:
          ret
find      endp
```

Если символ 'D' найден в строке, то подпрограмма возвращает AH = 1, а ES:DI будет адресом позиции, содержащей символ 'D'. В противном случае подпрограмма возвращает AH = 0.

Прерывания

Многие современные программисты довольно часто сталкиваются с такой ситуацией: есть старая программа, написанная несколько лет назад, разработчика уже и след простыл, а работать компания продолжает. Многие знают, что такие программы раньше защищали как только могли (точнее, как только мог придумать программист), — ключевые дискеты, всевозможные метки и т. д. Данный раздел написан специально для того, чтобы, попав в такую ситуацию, вы понимали, как работает такая программа и что следует делать.

В архитектуре процессоров 8086 предусмотрены особые случаи, когда процессор прекращает (прерывает) выполнение текущей программы и немедленно передает управление программе-обработчику, специально написанной для обработки подобной ситуации. Такая программа-обработчик называется *процедурой обработки прерывания*. Прерывание можно рассматривать как некоторое событие, требующее моментальной реакции. Практически все системы ввода/вывода в компьютере работают с использованием прерываний. Например, если в программе нужно организовать ожидание компьютером нажатия любой клавиши, используются такие команды:

```
mov  AH, 10h
int  16h
```

Команда `int 16h` и дает знать процессору, что надо вызвать прерывание с номером 16h для чтения клавиатуры.

Прерывания вырабатываются и контроллером диска, и видеоадаптером, и клавиатурой, и многими другими устройствами. Для того чтобы процессор различал все эти прерывания и не путал устройства, от которых они приходят, с каждым прерыванием связывают число из диапазона 00h—0FFh (от 0 до 255) — *номер прерывания*, однозначно характеризующий то или иное событие.

Программа может сама вызвать прерывание с заданным номером. Для этого используется команда:

```
int номер
```

где *номер* — номер прерывания в шестнадцатеричной записи.

Система прерываний

Прерывание может быть инициировано внешними по отношению к микропроцессору устройствами или вызвано командой, выполняемой микропроцессором. В первом случае прерывание называется *внешним*, во втором — *внутренним*.

Внутренние прерывания возникают внутри микропроцессора во время вычислительного процесса. Их можно разделить на два класса:

- *программные прерывания* — это прерывания, инициируемые командой

  `int XXh`

- *исключительные ситуации*, возникающие при выполнении команд, — деление на ноль, некорректная команда и т. д.

Внешние прерывания называются *маскируемыми*. Эти прерывания можно запретить с помощью команды `cli`, обнуляющей флаг IF регистра флагов. В этом случае подпрограммы обработки внешних прерываний не вызываются. Чтобы вновь разрешить эти прерывания, следует установить IF = 1, например, с помощью команды `sti`.

Прерывания обрабатываются микропроцессором независимо от значения флага IF. Тут есть один очень интересный момент: если установить флаг TF равным 1, то после выполнения каждой команды вызывается подпрограмма обработки прерывания номер 1. Этот момент довольно часто (практически всегда) используют отладчики уровня приложений, например Debug, Turbo Debugger и т. д. Данный режим работы называется *пошаговым*, а флаг TF — *флагом трассировки*. Прерывание номер 1, генерируемое в этом случае микропроцессором после каждой команды, является внутренним. Если TF = 0, то прерывание номер 1 не генерируется, и команды выполняются обычным образом.

Обработка прерывания

Обработка прерывания в реальном режиме производится в три этапа:

1. Прекращается выполнение текущей программы.
2. В стеке запоминаются значения регистров FL, затем CS и IP, очищается флаг IF и управление передается по адресу из таблицы прерываний.
3. Выполняется процедура обработки прерывания и управление возвращается прерванной программе с помощью команды `iret`. Команда `iret` восстанавливает содержимое регистров IP, CS и FL.

Как вы уже знаете, любая программа, загруженная в память, занимает свое, отдельное от других место. Ресурсами, которые операционная система разделяет между программами, являются также регистры микропроцессора, в том числе регистр флагов, поэтому их содержимое надо сохранять в своей программе обработки прерываний.

Таким образом, получается, что команда `int n` производит следующие действия:

1. Записывает в стек значение регистра FL.
2. Записывает в стек значение регистра CS.
3. Записывает в стек значение регистра IP.
4. Очищает бит IF.
5. Передает в регистр IP содержимое слова, имеющего адрес `4n`.
6. Передает в регистр CS содержимое слова, имеющего адрес `4n + 2`.

Команда `into` генерирует прерывание 4, если значение флага OF = 1, в противном случае никакие действия не выполняются.

Собственные программы обработки прерываний

Довольно часто может возникнуть ситуация, когда программисту приходится писать свою собственную программу-обработчик прерывания. На этом основаны некоторые защиты.

Обработчик прерывания, или *процедура обработки прерывания* пишется как обычная подпрограмма, например, как показано в листинге 2.17.

Листинг 2.17. Процедура обработки прерывания

```
obr_inter  proc  far
;     ...
; (тело программы)
     ...
      iret
obr_inter  endp
```

Этот обработчик, или программа, имеет адрес, состоящий из четырех байт. После того как мы написали сам обработчик прерывания, надо привязать его к номеру прерывания. Это можно сделать, например, записав его четырехбайтовый адрес прямо в таблицу векторов прерываний. Это самый надежный способ.

Примечание

Работая в ОС Windows, не забудьте запустить какую-либо DOS-оболочку (например, Dos Navigator, а лучше Norton Commander for DOS) в полноэкранном режиме. Свои программы запускайте только в этом окне, потому что Windows автоматически заменяет все прерывания и символы графики на свои и не позволяет менять их в окне, не развернутом на полный экран.

Если вы работаете в режиме DOS или виртуальной машины, то делать это не нужно, хотя и желательно.

Допустим, мы хотим связать наш обработчик с прерыванием 63h (это прерывание программист может использовать для своих нужд). Тогда нужно сделать следующее (листинг 2.18).

Листинг 2.18. Замена обработчика прерывания

```
; Запишем сегментный адрес таблицы векторов прерываний в ES
    push 0
    pop  ES
; Поместим регистр флагов в стек
    pushf
; Запретим прерывания.
; Это необходимо сделать для того, чтобы никакая программа не могла
; вызвать обработчик прерывания в момент изменения его адреса
; (когда смещение уже будет записано, а сегментный
; адрес - еще нет), поскольку это приведет к тому,
; что управление будет передано в область памяти,
; которая может содержать что угодно.
; Если это произойдет, возможен крах системы.
    cli
; Поместим дальний адрес обработчика obr_inter в элемент
; с номером 63h таблицы векторов прерываний
    mov      word ptr ES:[63h*4], offset obr_inter
    mov      word ptr ES:[63h*4+2], seg obr_inter
; Восстановим исходное значение флага IF
    popf
```

Таким образом, мы заполнили соответствующий вектор прерывания, который можно вызывать с помощью команды `int 63h`. Наш обработчик прерывания (см. листинг 2.17) состоит из одной команды `iret`, т. е. не делает ничего.

В нашей программе (см. листинг 2.18) мы не заботились о старом обработчике прерывания, что само по себе неверно и может нарушить нормальную работу системы. Чтобы это не произошло, перед завершением работы нашей программы мы должны восстановить все старые обработчики прерываний, даже если это были неиспользуемые прерывания.

Таким образом, мы получаем листинг 2.19.

Листинг 2.19. Корректная замена обработчика прерывания

```
; Запишем сегментный адрес таблицы векторов прерываний в ES
    push  0
    pop   ES
; Скопируем адрес предыдущего обработчика в переменную old_INT
    mov   EAX, dword ptr ES:[63h*4]
    mov   dword ptr old_INT, EAX
; Установим наш обработчик прерывания
; Поместим регистр флагов в стек
    pushf
; Запретим прерывания
    cli
; Поместим дальний адрес обработчика obr_inter в таблицу
; векторов прерываний, в элемент с номером 63h
    mov   word ptr ES:[63h*4], offset obr_inter
    mov   word ptr ES:[63h*4+2], seg obr_inter
; Восстановим исходное значение флага IF
    popf
; Собственно обработка прерывания
    ...
    ...
    ...
; Восстановим предыдущий обработчик
    push  0
    pop   ES
    pushf
    cli
    mov   EAX, word ptr old_INT
    mov   word ptr ES:[63h*4], EAX
    popf
```

Установка и чтение адреса обработчика прерывания с помощью функций DOS

Есть еще один способ для установки и чтения адреса обработчика прерывания в операционной системе DOS — с помощью функций 25h и 35h прерывания 21h: первая отвечает за установку, а вторая — за чтение адреса обработчика прерывания.

35h

Функция 35h прерывания 21h отвечает за чтение адреса обработчика прерывания. Входные данные:

- AH = 35h;
- AL = номер прерывания (от 00h до 0FFh).

Возвращаемые значения:

- ES:BX = адрес обработчика прерывания.

Функция возвращает значение адреса обработчика из таблицы векторов прерываний для `int AL`, т. е. загружает в регистр BX слово с адресом 0000:[AL*4], а в ES — слово с адресом 0000:[(AL*4) + 2].

25h

Функция 25h прерывания 21h отвечает за установку адреса обработчика прерывания. Входные данные:

- AH = 25h;
- AL = номер прерывания (00h до 0FFh);
- DS:DX = вектор прерывания (адрес подпрограммы).

Функция устанавливает значение элемента таблицы векторов прерываний для прерывания с номером, который содержится в регистре AL, равным DS:DX. Это равносильно записи 4-байтового адреса в 0000:[AL*4], но, в отличие от прямой записи, в момент записи прерывания будут заблокированы.

Установим обработчик прерывания с помощью функций DOS (листинг 2.20).

Листинг 2.20. Замена обработчика прерывания с помощью функций DOS

```
; Скопируем адрес предыдущего обработчика в переменную old_INT,
; а в AL = номер прерывания, которое мы хотим изменить.
; Вместо этих двух команд можно выполнить одну команду:
;                    mov AX, 3563h
    mov     AH, 35h
```

```
    mov     AL, 63h
; Функция DOS: считать адрес обработчика прерывания
    int     21h
; Поместим смещение в BX
    mov     word ptr old_INT, BX
; Поместим сегментный адрес в ES
    mov     word ptr old_INT+2, ES
; Установим наш обработчик прерывания
; для AL = номер прерывания, которое хотим заменить
; (можно использовать одну команду: mov  AX, 2563h
    mov     AH, 25h
    mov     AL, 63h
; Поместим сегментный адрес в DS
    mov     DX, seg obr_inter
    mov     DS, DX
; а смещение в DX
    mov     DX, offset obr_inter
; вызовем функцию DOS - установить обработчик прерывания
    int     21h
; Сама программа.
; НО: помните,что содержимое регистра ES
; после вызова функции 35h изменилось!!!
    ...
    ...
    ...
; Восстановим предыдущий обработчик:
; сегментный адрес в DS и смещение в DX
    lds     DX, old_INT
    mov     AH, 25h
    mov     AL, 63h
; Установим обработчик
    int     21h
```

Прерывания для ввода/вывода символов

Как вы уже успели заметить, операционная система тоже предоставляет набор функций для работы (например, для работы с графикой).

Рассмотрим функции, которые операционная система предоставляет для ввода/вывода. Это несколько первых функций прерывания 21h. Тут надо заметить, что эти функции работают со STDIN (стандартное устройство ввода), а раз так,

то вывод можно перенаправить в файл. Это очень удобно, если вам надо будет позже посмотреть, что же выводит программа.

01h

Функция 01h позволяет считать строку символов из стандартного устройства ввода (STDIN) с отображением на экран, ожиданием, проверкой на нажатие клавиш <Ctrl>+<Break>.

Входные данные:

- AH = 01h.

Возвращаемое значение:

- AL = ASCII-код символа или 0 (тогда во второй раз функция возвратит в AL расширенный ASCII-код символа).

Эта функция очень удобна тем, что если пользователь нажал какую-либо функциональную клавишу, например, стрелку, <Home>, <End>, <F1> и т. д., то функция вернет в AL = 0 и для получения расширенного ASCII-кода функцию надо вызвать еще раз.

Примечание

Это утверждение справедливо и для функций 06h, 07h, 08h.

02h

Функция 02h посылает символ из DL на стандартное устройство вывода. Входные данные:

- AH = 02h;
- DL = символ, выводимый на стандартное устройство вывода.

Данная функция обрабатывает и символ BS (т. е. нажатие клавиши <Backspace>, ASCII-код — 8), перемещая курсор влево на одну позицию и оставляя его в новой позиции.

03h

Функция 03h считывает (ожидает) символ со стандартного вспомогательного устройства ввода — COM1 (по умолчанию) или AUX — и возвращает этот символ в регистре AL.

Входные данные:

- AH = 03h.

Возвращаемое значение:

- AL = символ, полученный со стандартного вспомогательного устройства ввода.

Тут надо помнить, что ввод не буферизуется и должен опрашиваться (не управляется прерываниями). При запуске DOS порт AUX (COM1) инициализируется так:

- 2400 бод;
- без проверки на четность;
- 1 стоп-бит;
- 8-битовые слова.

04h

Функция 04h посылает символ (регистр DL) на стандартное вспомогательное устройство вывода (COM1 или AUX). Входные данные:

- AH = 04h;
- DL = символ, записываемый на стандартное вспомогательное устройство вывода.

05h

Функция 05h выводит символ (регистр DL) на стандартное устройство печати PRN (обычно LPT1). Входные данные:

- AH = 05h;
- DL = символ, записываемый на стандартный принтер.

06h

Функция 06h осуществляет непосредственный ввод и вывод с консоли (без ожидания). Функция не ожидает нажатия клавиши, не отображает введенный символ на экране, а при нажатии клавиш <Ctrl>+<C> не завершает работу программы, а записывает в AL число 3h — ASCII-код символа Ctrl/C.

Если при вызове этой функции регистр DL не содержит 0FFh, то содержимое DL выводится на экран.

Если же DL = 0FFh, то при нажатии клавиши ее код записывается в регистр AL и сбрасывается флаг ZF (при отсутствии нажатой клавиши флаг ZF будет установлен в 1).

Входные данные:

- AH = 06h;
- DL = 0FFh.

Возвращаемое значение:

- при ZF = 1 — если была нажата клавиша, то AL = 00h;
- при ZF = 0 — если клавиша не была нажата, то AL = код символа;
- если DL не содержит 0FFh, то значение DL направляется на стандартный вывод.

07h

Функция 07h считывает (ожидает) символ со стандартного устройства ввода (клавиатуры) и возвращает этот символ в регистре AL. Функция не фильтрует, не проверяет нажатие клавиш <Ctrl>+<Break>, <Backspace> и т. п.

Входные данные:

- AH = 07h.

Возвращаемое значение:

- AL = символ, полученный от стандартного устройства ввода.

08h

Функция 08h считывает (ожидает) символ со стандартного устройства ввода и возвращает этот символ в регистре AL. В отличие от функции 07h, она проверяет нажатие <Ctrl>+<Break>.

Входные данные:

- AH = 08h.

Возвращаемое значение:

- AL = символ, полученный от стандартного устройства ввода.

Примечание

Напомню, что для того чтобы получить расширенный ASCII-код, функцию надо вызвать еще раз.

09h

Функция 09h осуществляет вывод строки символов, находящейся по адресу DS:DX. Строка символов должна оканчиваться символом `'$'`. Входные данные:

- AH = 09h;
- DS:DX = адрес строки, заканчивающейся символом `'$'` (ASCII-код — 24h).

Обычно с этой функцией используют управляющие символы CR/LF (возврат каретки/перевод строки, ASCII-коды 0Dh и 0Ah), применяемые для перехода на новую строку.

Примечание

Строки, содержащие символ `'$'`, можно вывести на экран с помощью функции 40h (BX = 0).

0Ah

Функция 0Ah выполняет буферизованный ввод с клавиатуры. Вводится последовательность символов, последний из которых — возврат каретки, CR (нажатие клавиши <Enter>).

Входные данные:

- AH = 0Ah;
- DS:DX = адрес входного буфера.

Входной буфер это символьная строка, в первом байте которой указано максимальное число вводимых символов (включая возврат каретки, CR). Регистры DS:DX определяют адрес этой символьной строки:

max	?	?	?	?	?

`max` — максимальная допустимая длина ввода (от 1 до 254).

На выходе буфер содержит символьную строку со следующими данными:

- первый байт — максимальное число символов (значение `max`, определенное перед вызовом функции);
- второй байт — число символов, введенных с клавиатуры (значение `len` — действительная длина данных без завершающего нажатия клавиши <Enter>);
- начиная с третьего байта — символы, введенные с клавиатуры, последним из которых будет код 13 (символ возврата каретки), т. е. 0Dh:

max	len	len введенных символов	0Dh

Символы считываются со стандартного устройства ввода вплоть до нажатия клавиши <Enter> или по достижении числа символов, равного `max - 1`. Если считано `max - 1` символов, для каждого очередного символа включается звонок, пока не будет нажата клавиша <Enter> (введен символ возврата каретки, CR).

Во второй байт буфера записывается действительная длина введенной строки без учета завершающего символа CR. Последний символ в буфере — всегда CR (не засчитаный в `len`).

Символы, находящиеся в буфере (включая `len`), при последующих вызовах функции используются как "шаблон".

В процессе ввода можно использовать обычные клавиши редактирования: <Esc> — записывает в буфер символ `'\'` и ввод символов начинается заново; <F3> — выводит на экран все, что есть в буфере; <F5> — выводит на экран символ `'@'` и сохраняет текущую строку как шаблон и т. д. Большинство расширенных кодов ASCII игнорируются.

Функция распознает нажатие клавиш <Ctrl>+<Break>.

0Bh

Функция 0Bh проверяет состояние стандартного устройства ввода.

Входные данные:

- AH = 0Bh.

Возвращаемое значение:

- AL = 0FFh — символ доступен со стандартного устройства ввода;
- AL = 0 — нет символа.

Эту функцию очень удобно использовать перед вызовом функций 01h, 07h и 08h, чтобы избежать ожидания нажатия клавиши.

0Ch

Функция 0Ch очищает буфер стандартного устройства ввода, а затем вызывает функцию ввода, заданную в регистре AL. Это заставляет систему ожидать ввода очередного символа. Входные данные:

- AH = 0Ch;
- допустимые значения регистра AL:
  - 01h — ввод с клавиатуры;
  - 06h — ввод с консоли;
  - 07h — нефильтрующий ввод без эхо-вывода;
  - 08h — ввод без эхо-вывода;
  - 0Ah — буферизованный ввод.

Именно эту функцию используют в программах, которые могут уничтожить данные, для того, чтобы спросить согласия у пользователя. Например, прежде

чем форматировать диск, программа должна вывести на экран вопрос "Отформатировать диск Y/N?" и дождаться, пока пользователь нажмет клавишу <Y>.

Арифметические операции

Что такое арифметические операции, думаю, объяснять не надо. Давайте посмотрим, как реализуются в ассемблере основные арифметические операции (не будем затрагивать перенос и т. д., поскольку для анализа защиты нам это не пригодится; те, кому это понадобится, смогут найти все в справочной литературе (например, в моей книге "Assembler: Экспресс-курс").

Сложение и вычитание

В ассемблере используются следующие инструкции: `add` (сложение) и `sub` (вычитание). Их действие соответствует названию. Инструкция `add` выполняет сложение операнда-источника (правого операнда) с содержимым операнда-приемника и записывает результат в операнд-приемник. Инструкция `sub` делает то же самое, только она вычитает операнд-источник из операнда-приемника.

Например:

```
val_b    dw     99
end_v    dw     10

mov      dx, [val_b]
add      dx, 11
sub      dx, [end_v]
```

Вы видите, что сначала значение, записанное в `val_b`, загружается в регистр DX, затем к этому значению прибавляется константа 11 (в результате в DX получается значение 110) и, наконец, из значения DX вычитается значение 10, записанное в переменной `end_v`. Полученное в результате значение 100 сохраняется в регистре DX. Все очень просто.

Увеличение и уменьшение

Иногда в программе на ассемблере требуется выполнить сложение или вычитание, которое состоит просто в прибавлении к операнду или вычитании из операнда значения 1. Эти операции называют увеличением (инкрементацией, или инкрементом) и уменьшением (декрементацией, или декрементом). Для таких действий, как изменение содержимого счетчика или продвижение

регистров-указателей по памяти, вместо операций сложения и вычитания можно использовать операции увеличения и уменьшения.

Для выполнения таких операций предусмотрены две инструкции — `inc` (от increase — увеличить) и `dec` (от decrease — уменьшить). Инструкция `inc` прибавляет 1 к регистру или переменной, а инструкция `DEC` вычитает 1 из регистра или переменной.

Почему предпочтительнее использовать инструкцию `inc BX`, а не `add BX, 1`?

Тут все дело в быстродействии. Инструкция `ADD` занимает 3 байта, инструкция `inc` занимает только 1 байт и выполняется быстрее. Фактически экономнее выполнить две операции `inc`, чем прибавить к регистру размером в слово значение 2 (инструкция увеличения или уменьшения значения регистра или переменной размером в 1 байт занимает 2 байта, т. е. она тоже короче, чем инструкция сложения или вычитания). Это необходимо учитывать, если вы захотите изменить готовую программу, а листинга уже нет.

В общем, инструкции `inc` и `dec` — это наиболее эффективные инструкции, с помощью которых можно увеличивать и уменьшать значения переменных и регистров. Обычно их используют в различных защитах или чтобы задать компилятору оптимизацию программы по скорости выполнения.

Умножение и деление

Для операций умножения и деления также есть специальные инструкции.

Инструкция `mul` перемножает беззнаковые сомножители, всегда сохраняя 16-битовое произведение в регистре AX. Например, в следующем фрагменте программы:

```
mov     AL, 25
mov     DH, 40
mul     DH
```

значение регистра AL умножается на значение регистра DH, а результат (1000) помещается в регистр AX. Заметим, что в инструкции `mul` требуется указывать только один операнд, — другой сомножитель всегда хранится в регистре AL (или в регистре AX в случае перемножения 16-битовых сомножителей).

Перемножение 16-битовых сомножителей работает аналогично. Один из сомножителей должен храниться в регистре AX, а другой может находиться в 16-разрядном общем регистре или в переменной. 32-битовое произведение инструкция `mul` помещает в этом случае в регистры DX:AX, при этом млад-

шие (менее значащие) 16 бит произведения записываются в регистр AX, а старшие (более значащие) 16 бит — в регистр DX. Например, инструкции

```
mov    AX, 1000
mul    AX
```

загружают в регистр AX значение 1000, а затем возводят его в квадрат, помещая результат (значение 1 000 000) в регистры DX:AX.

В отличие от сложения и вычитания, в операции умножения не учитывается, являются ли сомножители операндами со знаком или без знака, поэтому имеется вторая инструкция умножения `imul` для умножения сомножителей со знаком. Если не принимать во внимание, что перемножаются значения со знаком, инструкция `imul` работает аналогично инструкции `mul`. Например, после выполнения инструкций

```
mov    AL, -2
mov    AH, 10
imul   AH
```

в регистре AX будет записано значение –20.

Инструкция `div` предназначена для деления.

При беззнаковом делении делимое должно быть записано в регистре AX, а делитель может храниться в любом общем регистре или переменной соответствующего размера. Инструкция `div` всегда записывает частное в регистр AL, а остаток — в AH. Например:

```
mov    AX, 51
mov    DL, 10
div    DL
```

Результат 5 (51/10) будет записан в регистр AL, а 1 (остаток от деления 51/10) — в регистр AH.

При делении 32-битового операнда на 16-битовый операнд делимое должно записываться в регистрах DX:AX. 16-битовый делитель может находиться в любом из 16-битовых регистров общего назначения или в переменной соответствующего размера. Например:

```
mov    AX, 2
mov    DX, 1        ; Загрузим в DX:AX 10002h
mov    BX, 10h
div    BX
```

В результате частное 1000h (результат деления 10002h на 10h) будет записано в регистр AX, а 2 (остаток от деления) — в регистр DX.

Как и при умножении, при делении имеет значение, используются операнды со знаком или без знака. Для деления беззнаковых операндов предназначена инструкция `div`, а для деления операндов со знаком — инструкция `idiv`.

Команды *lea* и *xlat*

Это очень полезные команды. Они довольно часто используются при занесении в переменную текстовой строки (пароля) с целью дальнейшей проверки на правильность. Поэтому крекеру очень важно понимать их функции.

Загрузка исполнительного адреса (команда *lea*)

Синтаксис:

```
lea    приемник, источник
```

Или, по-другому, можно представить это так:

```
приемник = Addr(источник)
```

Команда `lea` присваивает значение смещения (offset) операнда *источник* (а не его значение!) операнду *приемник*. Операнд *источник* должен быть ссылкой на ячейку памяти, а в качестве операнда *приемник* может выступать любой 16-битный регистр, кроме сегментных.

Вы должны помнить, что эта команда имеет то преимущество по сравнению с использованием оператора `offset` в команде `mov`, что операнд *источник* может иметь индексы. Например, следующая строка не содержит ошибок

```
lea BX, table[SI]
```

в то время как строка

```
mov BX, offset table[SI]
```

ошибочна, т. к. оператор `offset` вычисляется во время ассемблирования, а указанный адрес не будет известен до тех пор, пока программа не будет запущена на исполнение.

Кодирование по таблице (команда *xlat*)

Синтаксис:

```
xlat операнд
```

Или, по-другому:

```
AL = (BX + AL)
```

Команда `xlat` переводит байт согласно таблице преобразований. Указатель 256-байтовой таблицы преобразований находится в регистре BX. Байт, который нужно перевести, находится в регистре AL. После выполнения команды `xlat` байт в AL заменяется на байт, смещенный на AL байтов от начала таблицы преобразований.

Примечание

Помните, что таблица преобразований может содержать менее 256 байтов.

Эта команда довольно часто используется при шифровании методом подстановки.

Операнд *операнд* является необязательным, поскольку указатель таблицы должен быть загружен в регистр BX еще до начала выполнения команды.

Давайте рассмотрим это на примере. Например, нам надо перевести десятичное число (от 0 до 15) в соответствующую цифру шестнадцатеричной системы счисления:

```
lea           BX, HEX_TABLE       ; Указатель таблицы засылаем в BX,
mov           AL, DECIMAL_DIGIT   ; а переводимую цифру - в AL
xlat          HEX_TABLE           ; Перевод
; Теперь в AL находится ASCII-код соответствующей цифры
;шестнадцатеричной системы
HEX_TABLE     DB      '0123456789ABCDEF'
```

Глава 3

Основные способы ввода/вывода

Ввод данных с клавиатуры

Ввод данных в компьютер — важная часть любой программы, будь то игры, бухгалтерская программа или текстовый процессор. В разных ОС ввод данных реализуется по-разному, например: в Windows — с помощью API, в DOS — с помощью прерываний и т. д. Поскольку мы рассматриваем только основы, я покажу, как осуществляется ввод данных в MS-DOS.

Мы будем учиться принимать символы с клавиатуры.

В ассемблере символы с клавиатуры принимаются с помощью функции 1 прерывания 21h. Можно также вводить символ с клавиатуры с помощью прерывания BIOS. Символ вводится в регистр AL с помощью вызова функции 0 прерывания 16h. Строка символов с клавиатуры вводится с помощью функции 0Ah прерывания 21h. Ввод строки заканчивается нажатием клавиши <Enter>.

Я представлю вашему вниманию несколько наглядных подпрограмм, благодаря которым вы увидите, как именно вводятся данные (листинги 3.1—3.3).

Листинг 3.1. Ввод числа в системе счисления `syst`; результат помещается в регистр EAX

```
Input   proc
        arg     syst:word
; Ввод в буфер цифр числа
        mov     DX, offset Buff
        mov     AH, 10
        int     21h
        mov     SI, offset Buff+1
```

```
        xor     CX, CX
        mov     CL, [SI]
        xor     EAX, EAX
        xor     EDI, EDI
        mov     DI, syst
; Цикл перевода числа в машинный формат
Inp01:  inc     SI
; путем поразрядного умножения
        xor     EBX, EBX
; числа на основание системы счисления
        mov     BL, [SI]
        sub     BL, '0'
        mul     EDI
        add     EAX, EBX
        loop    Inp01
        ret
        endp
```

Листинг 3.2. Вывод числа `num` в системе счисления `syst`

```
Output  proc
        arg     num:dword, syst:dword
        pusha
        mov     DI, offset Buff+33
        mov     EAX, num
; Цикл получения цифр числа
Pr01:   xor     EDX, EDX
; путем вычисления остатков
        div     syst
; от деления на основание системы
        mov     SI, offset Tabl
        add     SI, DX
        mov     DL, [SI]
        mov     [DI], DL
        dec     DI
; Проверка на достижение конца
        cmp     EAX, 0
        jnz     Pr01
        mov     AH, 9
```

```
        mov     DX, DI
        dec     DX
; Печать результата
        int     21h
        popa
        ret
        endp
Buff    db 32, 33 dup(0), '$'
Tabl    db '0123456789ABCDEF'
```

Листинг 3.3. Печать текстовой строки

```
Print   proc
        arg     text:word
        pusha
        mov     DX, text
        mov     AH, 9
        int     21h
        popa
        ret
        endp
```

Преобразование и вывод данных на экран

Преобразованию данных в ассемблере уделяется особое место. Ввиду того, что в этом языке нет операторов для работы со строками, программистам приходится самим искать пути и способы для преобразования данных как таковых (помните, что речь идет о чистом ассемблере).

Я покажу один из способов, основываясь на котором, вы всегда сможете создать подпрограмму, реализующую то преобразование, которое вам необходимо.

Подпрограмма будет выводить содержимое байта (регистр DL) в шестнадцатеричном виде.

Для начала определим таблицу соответствия:

```
table1    db    '0123456789ABCDEF'
```

в которой значение шестнадцатеричной цифры будет равно смещению относительно начала таблицы символа, служащего ASCII-кодом этого значения.

Например код '5' находится по адресу table1 + 5, код 'E' находится по адресу table1 + 0Eh и т. д.

Воспользуемся следующей командой без операндов:

```
xlat
```

Операнды команды xlat задаются неявно:

- BX — является указателем на начало массива символов;
- AL — содержит число, которое служит смещением (индексом) байта, возвращаемого в регистр AL.

Для нашей таблицы table1 после выполнения команд

```
lea BX, table1
mov AL, 5
xlat
```

в регистре AL будет находиться символ '5'.

Подпрограмма (листинг 3.4) выводит на экран содержимое регистра DL следующим образом: сначала выводится символ, соответствующий старшей тетраде байта, а потом — символ, соответствующий его младшей тетраде.

Листинг 3.4. Вывод содержимого байта в шестнадцатеричном виде

```
HexByte proc
; Вывод содержимого регистра DL в шестнадцатеричном виде
push    DX
lea     BX, table1
mov     AL, DL
shr     AL, 4
xlat
; Вывод старшей цифры
mov     DL, AL
mov     AH, 2
int     21h
pop     AX
and     AL, 00001111b
xlat
; Вывод младшей цифры
mov     DL, AL
mov     AH, 2
int     21h
```

```
ret
table   db '0123456789ABCDEF'
HexByte endp
```

Теперь напишем подпрограммы вывода на экран содержимого слова в шестнадцатеричном виде (листинг 3.5). Для этого мы воспользуемся подпрограммой `HexByte`, с помощью которой выведем сначала старший, а затем младший байт.

Листинг 3.5. Подпрограммы вывода содержимого слова в шестнадцатеричном виде

```
subr   segment
public KeyIn, HexByte, HexWord, PressEnr
assume CS:subr, DS:subr
; Ввод символа в регистр AL
KeyIn  proc
mov    AH, 1
int    21h
ret
KeyIn  endp
; Вывод DL в шестнадцатеричном виде
HexByte proc
; Инициализация DS адресом сегмента, содержащего table1
mov     AX, subr
mov     DS, AX
push    DX
lea     BX, table1
mov     AX, 0
mov     AL, DL
shr     AL, 4
; Преобразование ASCII-кода цифры в числовое значение
xlat
mov     Dl, Al
mov     AH, 2
int     21h
; Бывшее содержимое регистра DX
pop     AX
; Младший полубайт
and     AL, 00001111b
xlat
```

```
mov     DL, AL
mov     AH, 2
int     21h
ret
; Вывод управляющих символов
PressEnr proc
mov     DL, 10
mov     AH, 2
int     21h
mov     DL, 13
mov     AH, 2
int     21h
ret
; Вывод в шестнадцатеричном виде
HexWord proc
; Сохранение DX
mov     DI, DX
; Вывод старшего байта
mov     DL, DH
call    HexByte
; Восстановление DX
mov     DX, DI
; Вывод DL
call    HexByte
mov     DL, ' '
mov     AH, 2
int     21h
ret
HexWord endp

; Таблица соответствия
table1   db '0123456789ABCDEF'
HexByte  endp

PressEnr endp
subr     ends
end
```

Команда `xlat` обращается к таблице, имеющей адрес DS:BX, поэтому в подпрограмме `HexByte` инициализируется регистр DS.

Глава 4

Отладчики

В этой главе я расскажу вам об отладчиках и дизассемблерах, описав их достоинства и недостатки. Вы научитесь использовать их для изучения программ. Итак, приступаем.

Отладчик DEBUG

Здесь мы рассмотрим отладчик DEBUG фирмы Microsoft. Этот отладчик уже давно не используется и пригодится только в том случае, когда надо что-то сделать по-быстрому, а другого отладчика под рукой нет. Поэтому мы рассмотрим только основные команды данного отладчика.

Запуск отладчика производится с помощью системной команды `DEBUG`. После ввода команды **`DEBUG`** на экране появляется приглашение в виде дефиса. Если ввести символ `?`, то появится список допустимых команд. Рассмотрим некоторые из них.

Дамп памяти. Для просмотра содержимого ячейки памяти следует ввести команду

`D` ***`S`*****`:`*****`M`**

где *`S`* — адрес сегмента, деленный на 16, а *`M`* — смещение. Например, серийный номер компьютера находится в памяти ROM, по адресу FE000h. Значит, чтобы его увидеть, следует ввести

```
D FE00:0
```

Выход. Для окончания работы с отладчиком следует ввести команду

```
Q
```

Ввод данных. Данные вводятся в оперативную память с помощью команды

`E` ***`S`*****`:`*****`M XX XX ... XX`**

где *S* — адрес сегмента, деленный на 16 (*S* может быть именем сегментного регистра), а *M* — смещение, *XX XX ... XX* — данные в шестнадцатеричной форме. Например, команда

```
E CS:100 CB 20
```

вводит инструкцию, состоящую из единственного оператора `int 20h` (возврат в DOS).

Ввод команд. Ввод команд ассемблера производится с помощью нажатия клавиши <A>, например:

```
A CS:100 int 20
```

вводит команду `int 20h`.

Трассировка. Трассировка производится нажатием клавиши <T>. Выполняется одна команда, которая находится по адресу CS:IP. Выводится содержимое регистров. Вывод содержимого регистров можно сделать также нажатием клавиши <R>.

Дизассемблирование. Если нажать клавишу <U> или ввести команду

```
U диапазон
```

то на экран будут выведены машинные команды на ассемблере, соответствующие кодам, находящимся в данном диапазоне.

Отладка программы. Производится вводом команды

```
DEBUG имя_программы
```

Например, чтобы отладить программу TEST.EXE, надо ввести команду

```
DEBUG TEST.EXE
```

Отладчик Turbo Debugger

Отладчик Turbo Debugger (TD) фирмы Borland является, на мой взгляд, самым удобным для начинающих при изучении программ для ОС MS-DOS. Вот только некоторые его возможности:

- выполнение отлаживаемой программы по шагам или с точками останова (интерактивный отладчик);
- трассировка программы в прямом и обратном направлении;
- вывод на экран содержимого регистров и областей памяти;
- изменение программы, загруженной в память;
- принудительное изменение содержимого регистров;

- другие действия, позволяющие в наглядной и удобной форме контролировать выполнение программы.

Сразу после старта вы видите окно процессора (рис. 4.1). Основные подокна этого окна:

- *окно кода программы* (левая верхняя часть окна процессора) — первые два столбца в этом окне, разделенные двоеточиями, указывают адрес того байта, на котором находится команда, в формате *сегмент:смещение*; остальные столбцы — код программы на ассемблере;
- *окно данных* (левая нижняя часть окна процессора) — позволяет просматривать содержимое любой области памяти как "данные". Первые два столбца — адрес первого байта строки. Затем сами данные — строка из восьми последовательно расположенных байт (числовой вид). Последний столбец отображает тот же набор данных в виде символов;
- окно регистров процессора (правая верхняя часть окна процессора);
- окно регистров флагов (правая верхняя часть окна процессора, крайний правый столбец);
- окно состояния стека (правая нижняя часть окна процессора).

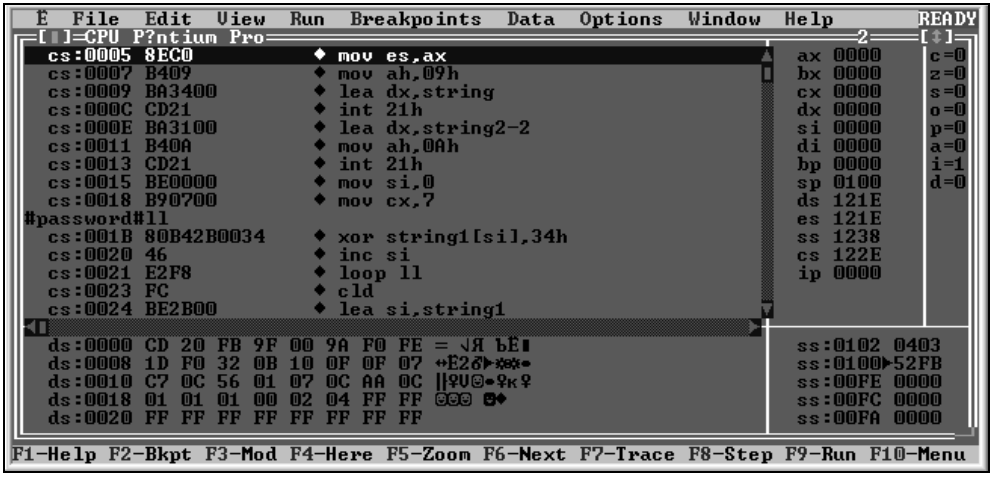

Рис. 4.1. Окно процессора

В процессе отладки программы на экран приходится выводить много дополнительных окон (это и окно кода, и окно переменных, и т. д.). Окон бывает слишком много, они перекрываются и часто скрывают друг друга. Чтобы увидеть их все одновременно, размер окон приходится изменять, а сами окна перемещать по экрану.

Режим изменения размеров и положения окна включается нажатием комбинации клавиш <Ctrl>+<F5>, после чего окно перемещают по экрану с помощью клавиш со стрелками; те же клавиши при нажатой клавише <Shift> позволяют изменять его размер. Выход из режима настройки окна осуществляется нажатием клавиши <Enter>.

Примечание

Все это можно сделать и с помощью мыши, например, чтобы изменить размер окна, захватите мышью и перемещайте его правый нижний угол.

Перемещение между окнами осуществляется с помощью клавиши <F6>.

Чтобы развернуть окно на весь экран, нажмите клавишу <F5>.

В отладчике можно выполнять программу несколькими способами:

- по частям до местоположения курсора (клавиша <F4>);
- команда за командой с заходом в процедуру (клавиша <F7>);
- команда за командой без захода в процедуру (клавиша <F8>).

Можно также наблюдать изменение переменных с помощью окна **Watches**, открыв его с помощью соответствующей команды меню. Однако для отладки программы на уровне языка ассемблера необходимо контролировать все регистры процессора, включая регистр флагов. Самым информативным является окно процессора, вызываемое с помощью пункта **View > CPU** главного меню или комбинации клавиш <Alt>+<V>+<C>. В большинстве случаев его подокон бывает достаточно для отладки программы.

Для того чтобы можно было работать с конкретным окном, надо сделать его активным, щелкнув на нем мышью. Переходить от окна к окну можно также с помощью клавиатуры, нажимая клавишу <Tab>.

Это все теория. Лучше всего изучать отладчик, пробуя отладить какую-либо программу. Поскольку эта книга посвящена защите программного обеспечения, давайте попробуем в качестве тренировки найти с помощью отладчика Turbo Debugger пароль, который запрашивает программа (листинг 4.1).

Листинг 4.1. Запрос пароля

```
; Эта программа создана для демонстрации принципов работы
; с отладчиком Turbo Debugger.
; Программа запрашивает пароль и производит его проверку.
; При правильном пароле программа выдает сообщение "PASSWORD IS OK",
```

```
; при неправильном - сообщение "PASSWORD IS NOT CORRECT"
    model small
    stack 256
    .data
    match   db 0ah,0dh,'PASSWORD IS OK','$'
    failed  db 0ah,0dh,'PASSWORD IS NOT CORRECT','$'
    string1 db 07Fh,071h,06Dh,067h,060h,07Bh,064h,'$'
    string2 db 10
    string  db 'Input PASSWORD:','$'
    .code
assume DS:@data, ES:@data
main:
    mov AX,@data
    mov DS,AX
    mov ES,AX
; Выводим приглашение ко вводу пароля
    mov AH,09h
    lea DX,string
    int 21h
; Запрашиваем символы с клавиатуры
    lea DX,string2-2
    mov AH,0Ah
    int 21h
; Производим расшифровку пароля
    mov SI,0
    mov CX,7
ll:
    xor string1[SI],34h
    inc SI
    loop ll
; Процедура проверки правильности пароля
    cld
    lea SI,string1
    lea DI,string2
    mov CX,7
cycl:
    repe cmps string1,string2
    jcxz equal      ; Если совпадение, переход
    jne not_match
```

```
equal:
; Пароль верный
    mov AH,09h
    lea DX,match
    int 21h
    jmp exit
not_match:
; Пароль неверный
    mov AH,09h
    lea DX,failed
    int 21h
    jmp exit
    jmp cycl
exit:
    mov AX,4c00h
    int 21h
    end main
```

Примечание

Откомпилированный для TD модуль программы вы найдете на сопроводительном компакт-диске книги.

Для начала мы должны загрузить нашу программу в отладчик. Это делается так:

```
td password.exe
```

Введя эту команду, мы увидим окно отладчика (рис. 4.2).

Это окно показывает, что в отладчике сохранены все символьные имена. Это произошло потому, что наша программа была откомпилирована специальным образом. На рис. 4.2 видно, что в левом столбце Turbo Debugger подсказывает нам, какая команда сейчас будет выполняться процессором. Вы можете попробовать выполнить трассировку программы и посмотреть, что изменится в главном окне.

Примечание

Для того чтобы вернуть отладчик в первоначальное состояние и перезагрузить программу, нажмите клавиши <Ctrl>+<F2>.

Это окно хорошо подходит для отладки программы, в которой есть ошибки. Но мы-то должны найти пароль, в чем это окно нам не поможет. Нужно вызвать окно процессора (см. рис. 4.1). Для этого выбираем команду **View > CPU** глав-

ного меню и разворачиваем окно процессора на полный экран нажатием клавиши <F5>.

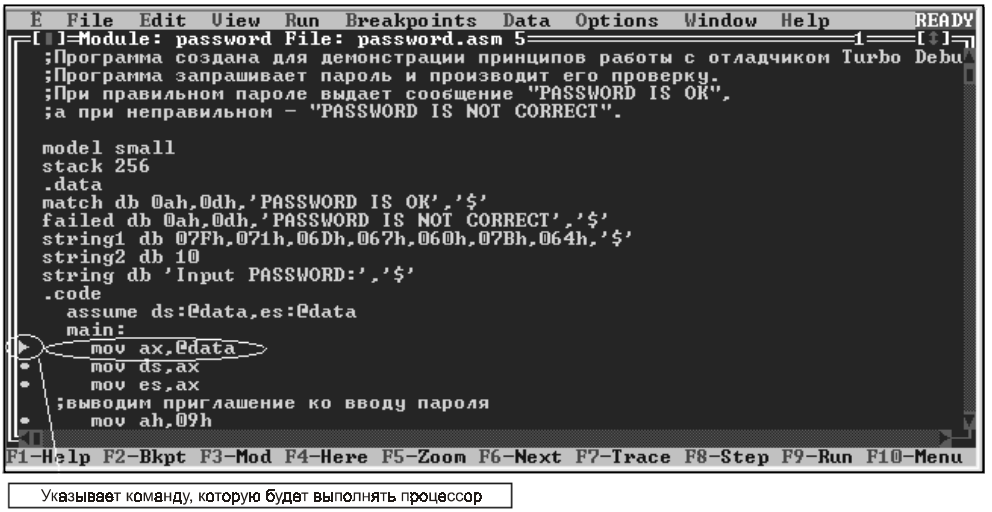

Рис. 4.2. Окно отладки программы

Вы можете опять попробовать выполнить трассировку программы и посмотреть, что изменится.

Примечание

Чтобы увидеть то, что выводит на экран программа, нажмите клавиши <Alt>+<F5>. Чтобы вернуть отладчик в первоначальное состояние, нажмите клавиши <Ctrl>+<F2>.

Выполняем трассировку программы (клавиша <F8>) до строки, показанной на рис. 4.3.

Команды

```
mov AH,09h
lea DX,string
int 21h
```

выводят сообщение на экран. И действительно, если нажать клавиши <Alt>+<F5>, мы увидим, что программа вывела сообщение о вводе пароля (рис. 4.4).

Продолжаем дальнейшую трассировку программы (клавиша <F8>) до строки, показанной на рис. 4.5.

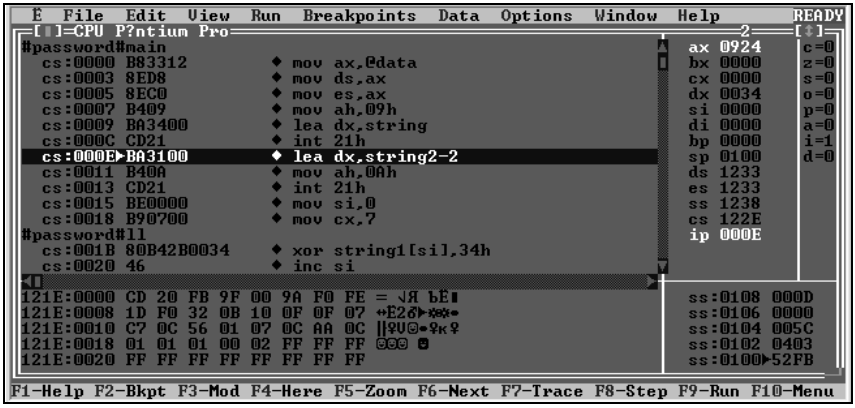

Рис. 4.3. Трассировка программы

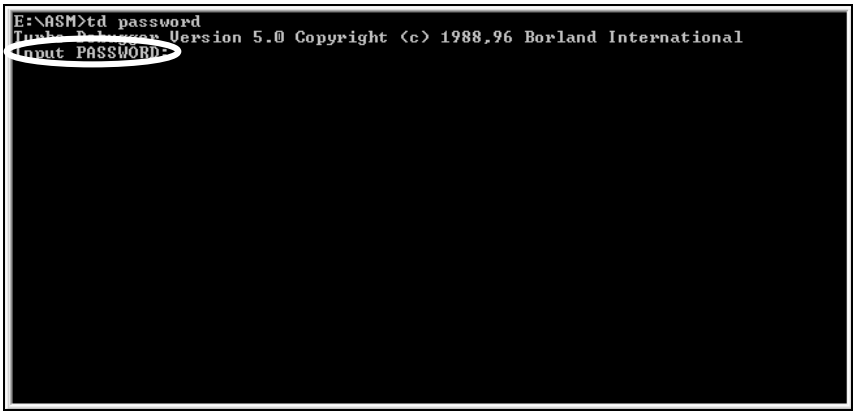

Рис. 4.4. Вывод сообщения

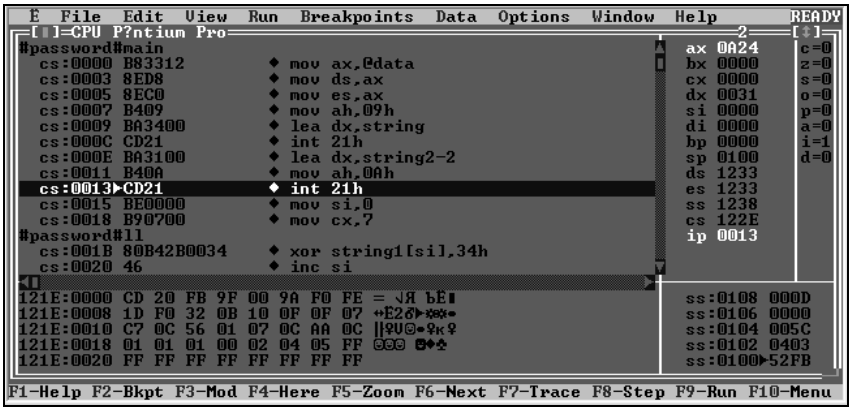

Рис. 4.5. Запрос пароля

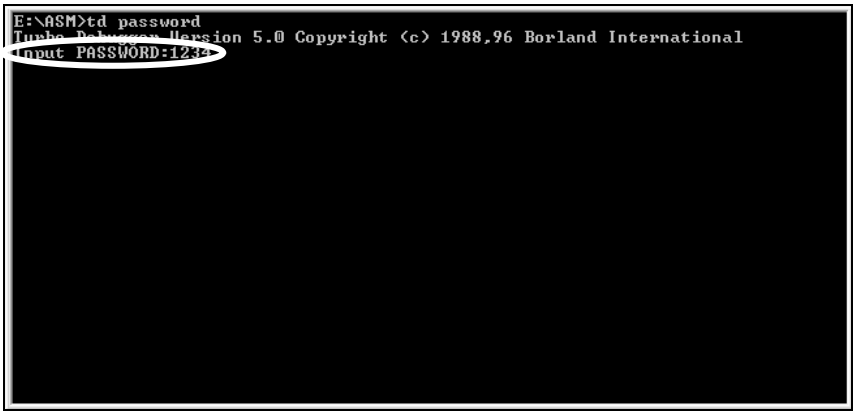

Рис. 4.6. Ввод пароля

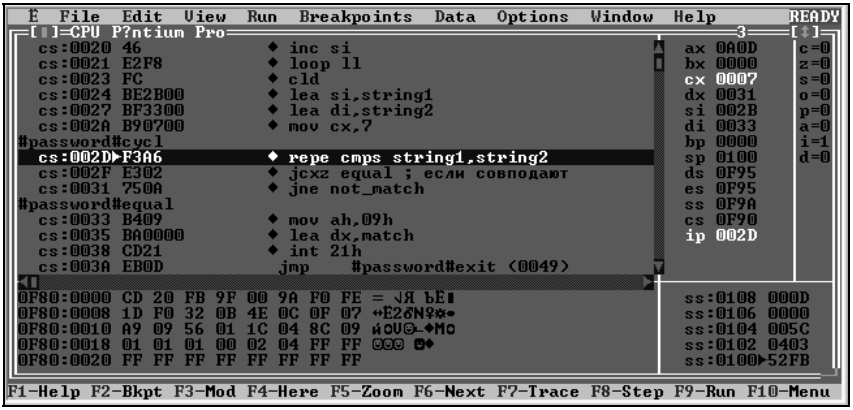

Рис. 4.7. Проверка пароля

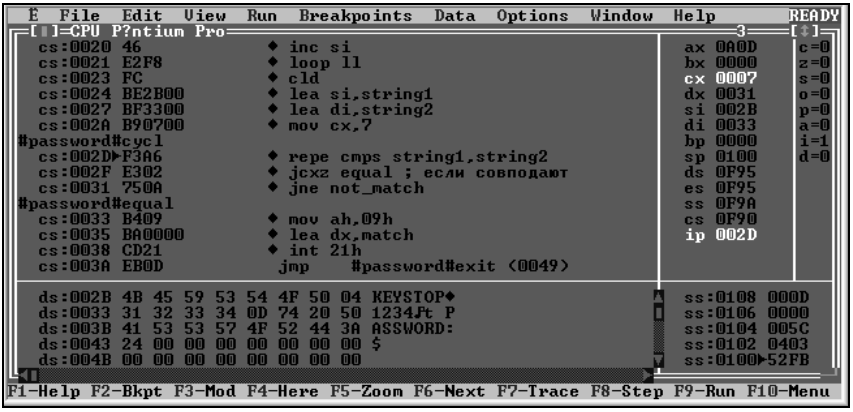

Рис. 4.8. Просмотр данных

Запрос пароля организован в трех строках программы:

```
lea DX,string2-2
mov AH,0Ah
int 21h
```

Если теперь нажать клавишу <F8>, то программа будет запрашивать пароль (см. рис. 4.4). Так как мы не знаем правильного пароля, введем любые символы, например, **1234** и нажмем клавишу <Enter> (рис. 4.6).

После этого нажимаем клавишу трассировки <F8> и доходим до места в программе, показанного на рис. 4.7.

Как видим, тут происходит сравнение введенной строки со строкой пароля. Чуть выше в тексте программы видно, что в регистры SI и DI помещаются две строки. Как бы их посмотреть? Давайте теперь переместимся в окно данных и нажмем комбинацию клавиш <Ctrl>+<G>, а на запрос адреса введем **string1** (рис. 4.8). Что же мы видим? А видим мы строку, которая находится в переменной `string1`, а следующая строка — это введенный нами пароль.

А зачем программе сравнивать две строки, одна из которых — введенный пароль? Значит, вторая строка — это искомый пароль. Проверим. Для этого перезапустим программу, нажав комбинацию клавиш <Ctrl>+<F2>, и на запрос пароля введем **KEYSTOP**.

Правда, ничего сложного? Вот таким образом исследовались почти все защиты для ОС MS-DOS. Вы можете сами найти в листинге программы место, где записан пароль, и, заменив его и откомпилировав программу заново, попробовать найти его еще раз.

Далее мы будем рассматривать программы только для ОС Windows, т. к. это наиболее актуально. Начнем с описания отладчика SoftICE.

Отладчик SoftICE

Теперь пора рассказать вам и об отладчике SoftICE. Из всех существующих для платформы Windows отладчиков наибольшей популярностью вполне заслуженно пользуется отладчик SoftICE фирмы NuMega, на мой взгляд, самый мощный. Правда, сейчас его несколько потеснил OllyDbg.

Особенности

Чем же SoftICE отличается от остальных отладчиков? Наиболее важным его отличием является то, что SoftICE работает в нулевом кольце защиты.

В архитектуре 32-разрядных процессоров фирмы Intel есть три кольца защиты (ring-1, 2, 3). Они предназначены для ограничения взаимодействия выпол-

няющихся программ между собой и с операционной системой. Обычно ОС имеет права полного доступа ко всем остальным выполняемым программам, т. к. она работает в нулевом кольце защиты (наиболее привилегированном), системные задачи выполняются в первом и втором кольцах, а приложения — в третьем (наименее привилегированном). Частично управлять запущенными приложениями можно из первого и второго колец защиты; именно так работает большинство отладчиков для разработки программ.

Разработчики из NuMega подошли к делу еще более профессионально и создали SoftICE — отладчик, который работает в нулевом кольце защиты и позволяет полностью контролировать не только все выполняемые системой задачи, но и саму ОС Windows. Таким образом, с его помощью можно отлаживать все, что вы захотите, даже Windows. Конечно, для большинства задач SoftICE является слишком мощным или неудобным, поэтому я рекомендую использовать отладчик OllyDbg, который намного проще в освоении (но не слабее), и с которым, на мой взгляд, гораздо приятнее работать. Но про него я расскажу позднее.

Итак, начинаем.

Установка

Сейчас у большинства на компьютере установлена ОС Windows XP, поэтому и мы будем рассматривать установку SoftICE в ней. Для тех, у кого установлена ОС Windows 9*x*, установка отладчика пройдет еще проще. Во избежание проблем с установкой я рекомендую брать самую последнюю версию программы на сайте **www.compuware.com** (во время подготовки материала этой книги я работал с версией SoftICE 4.2.7 из Compuware DriverStudio 2.7).

Запускаем установку SoftICE. Нам предлагают выбрать язык, на котором будут выводиться сообщения инсталлятора (выбираем русский). Далее нам показывают лицензионное соглашение, с которым придется согласиться, если мы хотим продолжить. Затем выбираем папку, в которую будет устанавливаться SoftICE, и начинаем установку. После того как программа установится, на экране появится окно конфигурации (рис. 4.9).

По умолчанию выбран режим запуска отладчика **Manual**, т. е. ручной. Так и оставляем. Переключаемся в раздел **Mouse** (рис. 4.10).

В этом разделе устанавливаются настройки мыши. Поскольку у вас, скорее всего, мышь нового образца, выбираем вариант настройки, показанный на рис. 4.10.

Примечание

Если у вашей мыши больше двух кнопок, необходимо установить флажок **Enhanced Mouse**.

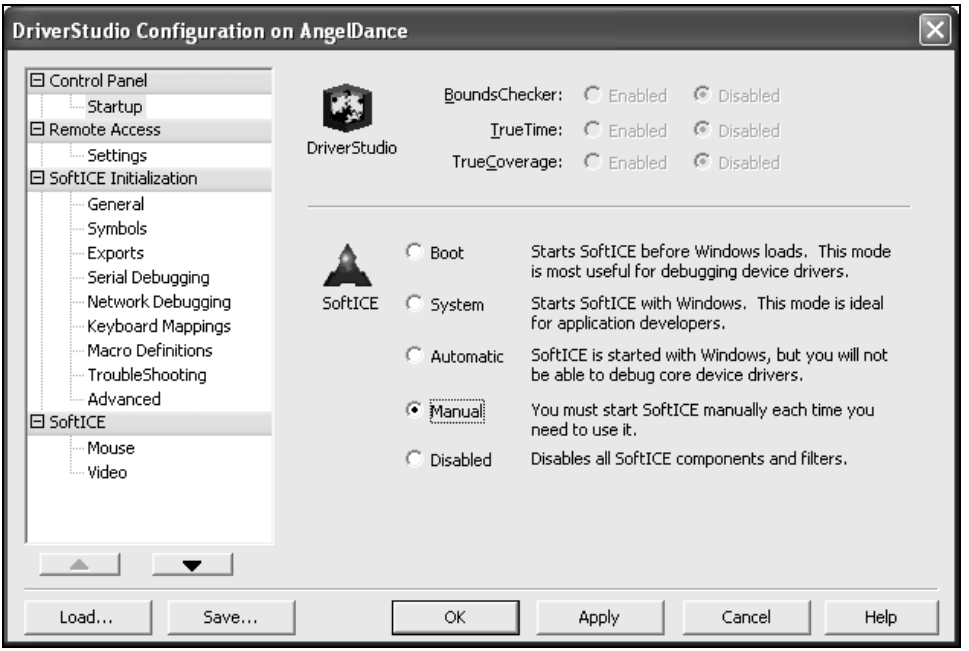

Рис. 4.9. Окно конфигурации SoftICE

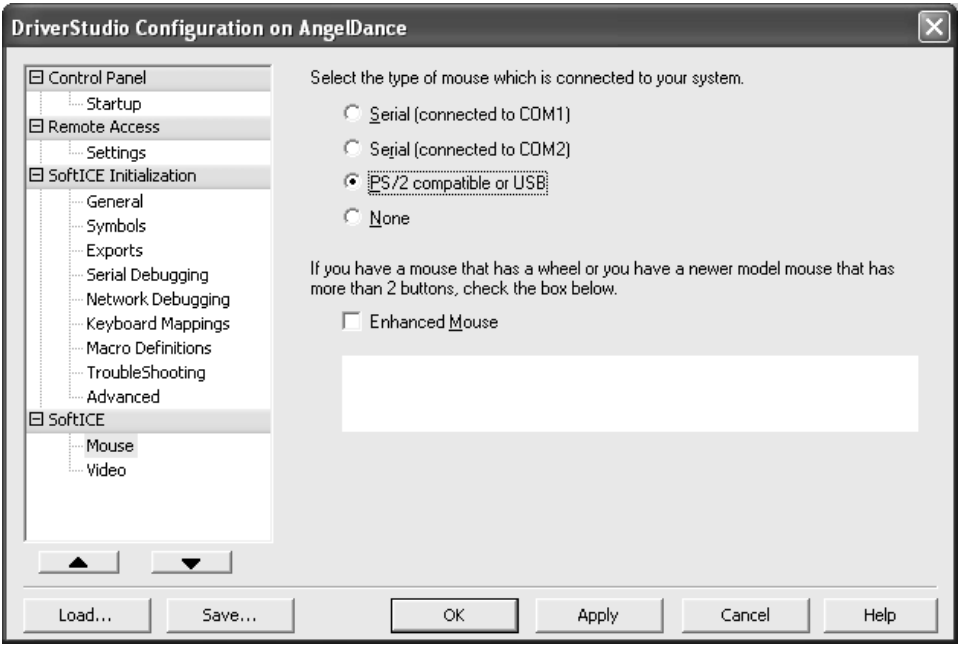

Рис. 4.10. Раздел **Mouse**

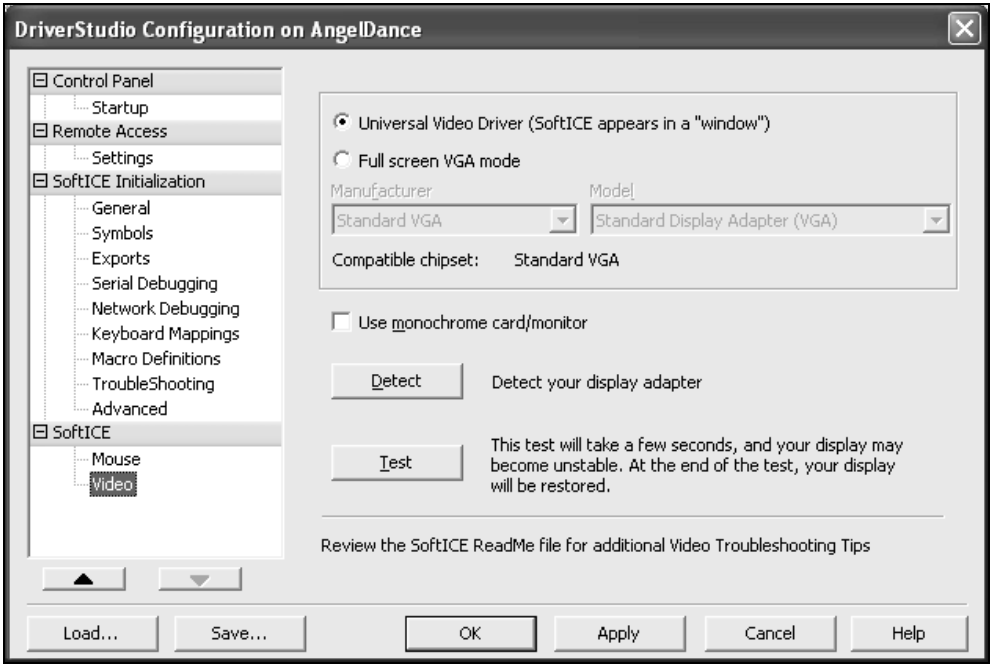

Рис. 4.11. Раздел **Video**

Далее переключаемся в раздел **Video** (рис. 4.11).

По умолчанию SoftICE выбирает режим **Universal Video Driver** (универсальный видеодрайвер) — это позволит выбирать размер окна SoftICE. Я рекомендую оставить так, как есть. Чтобы проверить работоспособность данного режима, нужно нажать кнопку **Test**. Если все в порядке, нажимаем кнопки **Apply** и **OK**.

Все, установка завершена.

Настройка

Теперь нам необходимо настроить SoftICE для корректной работы. Для этого ищем файл WINICE.DAT — это файл настроек. В нем есть строки вида:

```
; EXP=\SystemRoot\System32\hal.dll
; EXP=\SystemRoot\System32\kernel32.dll
; EXP=\SystemRoot\System32\basesrv.dll
; EXP=\SystemRoot\System32\ntdll.dll
; EXP=\SystemRoot\System32\ntoskrnl.exe
```

```
; EXP=\SystemRoot\System32\winsrv.dll
; EXP=\SystemRoot\System32\csrsrv.dll
; EXP=\SystemRoot\System32\user32.dll
```

С символа точки с запятой (;) в программах на ассемблере начинается строка комментария. В этих строках указываются библиотеки экспорта символов для SoftICE. Это значит, что при дизассемблировании файла SoftICE анализирует вызов процедур из этих библиотек. То есть мы будем видеть имена вызываемых процедур, а не просто их адреса. Чтобы все это работало, нам надо убрать точка с запятой у всех строк.

Далее необходимо перезагрузить компьютер. Для запуска SoftICE необходимо запустить файл NTICE.BAT из папки установки SoftICE (по умолчанию это папка C:\program files\SoftICENT\SoftICE).

Если теперь нажать комбинацию клавиш <Ctrl>+<D>, то появится рабочее окно, показанное на рис. 4.12; скрывается окно нажатием этой же комбинации клавиш.

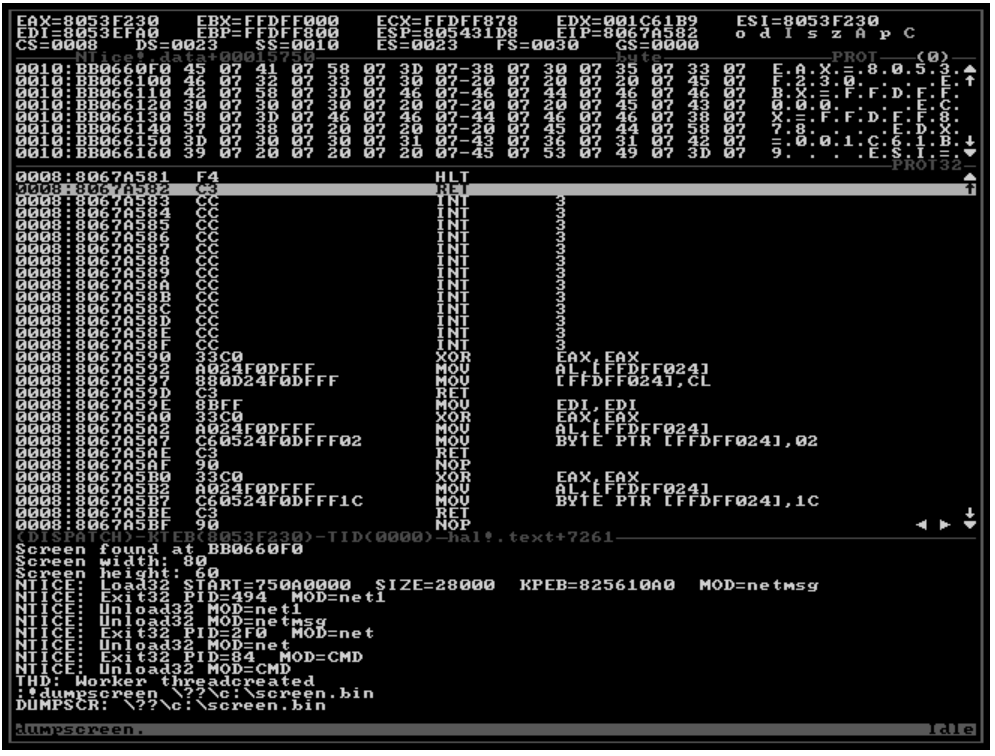

Рис. 4.12. Рабочее окно отладчика

К сожалению, с SoftICE не все бывает гладко, часто возникают проблемы с запуском. Конкретные рекомендации здесь дать сложно, т. к. они зависят от операционной системы, в которой осуществляется установка, а также от версии DriverStudio; также может понадобиться установить дополнительные патчи. Поэтому рекомендую статью "Установка отладчика SoftICE на Windows XP с SP1/SP2" (автор Bad_guy), которую можно найти на сайте **www.cracklab.ru**. В ней автор рассмотрел большинство проблем, возможных после установки SoftICE, и способы их решения.

Итак, мы считаем, что SoftICE установлен правильно, и мы видим окно, представленное на рис. 4.12. Это главное окно программы. Нам видно, что оно поделено на более мелкие, но не менее значимые подокна.

Рассмотрим наиболее важные из них подробнее.

Окно регистров

Главным свойством окна регистров, расположенного в верхней части главного окна отладчика, является то, что содержимое всех регистров, а также состояние флагов, можно изменять "на лету".

Чтобы изменить содержимое регистра, нужно щелкнуть левой кнопкой мыши на значении соответствующего регистра и внести с клавиатуры необходимые изменения. Значения флагов изменяются несколько иначе: надо щелкнуть на букве, обозначающей флаг, а затем изменить значение флага при помощи клавиши <Insert>. Смена регистра буквы означает, что флаг успешно изменен.

Окно данных

Прежде всего, обратим внимание на заголовок окна данных, расположенного под окном регистров. Первым в заголовке обычно указывается имя модуля, которому принадлежат данные (это важно, т. к. нам необходимо знать, в каком модуле мы сейчас находимся), а затем формат отображения данных — byte, word, dword и т. д. Если потребуется изменить формат отображения данных, для этого надо всего лишь щелкнуть на нем мышкой. Таким образом, мы можем видеть одни и те же данные в любом предусмотренном SoftICE формате. К сожалению, в SoftICE отсутствует поддержка просмотра в формате Unicode, а это значит, что мы не увидим русский шрифт.

Добавлю, что в SoftICE имеется 4 окна данных, пронумерованных от 0 до 3. И хотя перед глазами находится только одно из них, мы можем легко переключаться между ними; для этого надо сделать щелчок на номере окна (справа,

в скобках) — он должен измениться. Это позволяет следить за несколькими участками памяти. Можно также переключаться между окнами данных с помощью вводимой с клавиатуры команды:

```
DATA [номер_окна]
```

Например, команда `DATA 0` сделает активным первое по порядку окно данных. Для того чтобы в окне данных появился нужный участок памяти, надо ввести команду

```
D начальный_адрес_участка
```

или, щелкнув на экране на каком-либо адресе правой кнопкой мыши, выбрать в появившемся контекстном меню пункт **Display**.

В этом окне можно и редактировать данные, как в любом из числовых, так и в символьном формате, для этого достаточно щелкнуть в нужной точке и исправить значение.

Окно кода

Это самое интересное окно. Расположено оно под окном данных. В нем отображаются команды исследуемой программы. Для того чтобы установить точку останова на команду или отключить ее, достаточно выполнить на команде двойной щелчок мыши.

В заголовке этого окна указываются имя программного модуля, которому принадлежит отображаемый код, и текущий режим процессора (V86, Prot16 или Prot32).

Окно команд

Это не менее значимое окно расположено в нижней части главного окна. Все команды, которые можно ввести, мы будем вводить именно здесь.

Обратите внимание на строку подсказки, которая находится в нижней строке окна команд. Если вы начали набирать какую-нибудь команду, то в этой строке тут же появится подсказка, перечисляющая все похожие на набранную команды (если команда набрана не до конца) или список и назначение параметров, требуемых для этой команды.

В заголовке окна команд отображается имя текущего исполняемого модуля (исследуемой программы или DLL, OCX и т. д.). Кроме того, в этом окне ведется основная работа.

Точки останова

Теперь, когда SoftICE загружен, Windows и его приложения свободно выполняются и SoftICE реагирует только на особые события, которые еще называют *точками останова* (breakpoint). Точка останова (брекпойнт) это то место, в котором программа прервется и передаст управление отладчику. Точки останова широко используются при отладке программ. Это своего рода рычаги, благодаря которым мы воздействуем на программу. В отличие от большинства других отладчиков, работающих с машинным кодом, точки останова в SoftICE могут содержать условные операторы с вычисляемыми значениями и команды, автоматизирующие некоторые действия.

Задание точек останова

На запуск

Синтаксис:

`bpx <имя_функции_API>`

Эта команда предписывает SoftICE установить точку останова программы в месте вызова программой функции API. Рассмотрим это на примере.

Запустим SoftICE и установим точку останова на API-функции `GetLocalTime`. Для этого введем команду

`bpx GetLocalTime`

Теперь закроем SoftICE нажатием комбинации клавиш <Ctrl>+<D> и сделаем двойной щелчок на значке часов. Вот! Всплыло главное окно SoftICE.

На доступ к памяти (чтение/запись)

Синтаксис:

`bpm <адрес> R/W`

`bpr <начальный_адрес> <конечный_адрес> R/W`

Параметр `r/w` предназначен для того, чтобы установить брекпойнт в том месте, когда выполняется запись (`w`) или чтение (`r`) по адресу *<адрес>* (или в диапазоне адресов от *<начальный_адрес>* до *<конечный_адрес>*).

Если у вас есть опыт, или вы знаете, в какой зоне памяти содержится исследуемое значение, то применяют второй вариант этого брекпойнта. Обычно это используется при подборе паролей.

На поиск в памяти

Синтаксис:

`s <старт> l <размер_маска> '<строка>'`

Скорее всего, вы не знаете нужную величину, вот для этого и используется поиск. Для него обычно задают параметр, чтобы искать значение везде в памяти. Например:

```
s 0 l ffffff 'строка'
```

После того как поиск завершится и будет что-то найдено, окно данных переместится к этому адресу. Чтобы начать следующий поиск, просто введите команду **s**.

Просмотр списка точек останова

Для просмотра списка точек останова введите команду

```
bl
```

Эта команда выводит список всех точек останова в формате

Номер) Тип_точки_останова Параметры Счетчик

Например:

```
00) bpx #0043:07836143 C=02
```

Здесь показано, что установлен один брекпойнт и, как видно из примера (C=02), брекпойнт сработает при втором обращении.

Удаление точек останова

Синтаксис:

bc *<номер>*

Команда **bc** удаляет брекпойнты по номерам, например:

bc 0 — удаляется брекпойнт с номером 0.

bc 1,3,4 — удаляются брекпойнты с номерами 1, 3 и 4.

bc * — удаляются все брекпойнты.

Выключение точек останова

Синтаксис:

bd *<номер>*

Команда **bd** выключает брекпойнты по номерам, например:

bd 1,3,4 — выключаются брекпойнты с номерами 1, 3, 4.

bd 1 — выключается брекпойнт с номером 1.

bd * — выключаются все брекпойнты.

Эта команда бывает полезна при сложных операциях. Например, мы нашли адрес, поставили на него брекпойнт и занимаемся дальнейшей отладкой. Если мы теперь удалим брекпойнт, то для того, чтобы его поставить вновь, придется опять искать этот адрес.

Раз мы уже умеем отключать брекпойнт, давайте научимся его включать.

Включение точек останова

Синтаксис:

`be <номер>`

Полезные команды.

В SoftICE есть много разных команд, с помощью которых можно узнать состояние и получить иную информацию об операционной системе и о запущенных в ней приложениях. Я покажу вам несколько команд.

Команда `H` (help) позволяет получить помощь по всем командам SoftICE или более подробную информацию о конкретной команде, если ввести ее имя в качестве аргумента. Например:

```
H CLASS
Display window class information
CLASS [-x] [task-name]
ex: CLASS USER
```

Первая из выводимых строк содержит описание команды, вторая — информацию о синтаксисе и аргументах, которые могут использоваться в команде, третья содержит пример использования команды.

Далее я покажу все на примере программы GDIDEMO. Данная программа поставляется вместе с SoftICE для того, чтобы пользователь мог проводить эксперименты с SoftICE.

С помощью команды `CLASS` можно посмотреть зарегистрированные классы, например:

```
CLASS GDIDEMO
```

Результатом выполнения данной команды является информация о каждом зарегистрированном классе окна. Информация включает: адрес внутренней структуры данных `WNDCLASS`, имя класса, модуль, зарегистрировавший данный класс, адрес процедуры, обслуживающей данный класс, и состояние флагов стиля класса. Для получения подробной информации воспользуйтесь ключом `-X`.

```
Handle      Class Name   Owner        WndwProc         Styles

5110        BOUNCEDEMO   GDIDEMO      2E9F:00000114    03000003

40AC        DRAWDEMO     GDIDEMO      2E9F:000000FE    03000003

409C        MAZEDEMO     GDIDEMO      2E9F:000000E8    03000003

3BC4        XFORMDEMO    GDIDEMO      2E9F:000000D2    03000003

3BB4        POLYDEMO     GDIDEMO      2E9F:000000BC    03000003

3A00        GDIDEMO      GDIDEMO      2E9F:000000A6    03000003
```

Иногда для отладки приложения мы создаем его с отладочной информацией. Затем, когда вы загружаете такое приложение в SoftICE, он автоматически создает таблицу символьных имен, которая содержит все имена, определенные в данном приложении. С помощью команды

TABLE

можно посмотреть таблицы символьных имен, загруженные в настоящий момент. Например:

```
TABLE
GDIDEMO [NM32]
0001044741 Bytes Of Symbol Memory Available
```

Используемая в данный момент таблица символьных имен выделена цветом. Если текущая таблица символьных имен не соответствует той, на которую ссылается ваше приложение, то, введя команду **TABLE** с именем вашего приложения в качестве аргумента, вы подключите нужную таблицу (если для вашего приложения создана таблица):

```
TABLE GDIDEMO
```

С помощью команды

SYM

вы можете посмотреть все символьные имена, определенные в текущей таблице (они выводятся на экран по сегментам; внутри них — в алфавитном порядке). Если вас интересуют какие-то определенные имена, то используйте шаблоны, например, команда

```
SYM w*
```

выведет на экран список всех символьных имен, начинающихся с буквы "W", расположенных в сегменте `.text` (исполняемый сегмент длиной 0145C1h

байта, начинается с адреса 0137:00401000), т. е. имена функций и процедур, входящих в приложение GDIDEMO:

```
.text     (0137:00401000,  000145C1 bytes)

          0137:004012E0    WinMain

          0137:00405700    WinMainCRTStartup

          0137:004013AD    WndProc

          0137:0040AF50    wcslen

          0137:0040C160    wcsncnt

          0137:004107A0    wctomb

          0137:0040FA50    write_char

          0137:0040FAD0    write_multi_char

          0137:0040FB20    write_string
```

Горячие клавиши

<F1> — помощь;

<F2> — окно регистров;

<F3> — переключение текущего режима работы с исходным текстом;

<F4> — отображение рабочего стола без возможности ввода/вывода;

<F5> — выход в Windows;

<F6> — переключение курсора между окном команд и окном кода;

<F7> — прогон программы до текущей позиции курсора;

<F8> — трассировка к следующей инструкции (переходит на все обращения). Обычно используется для "погружения" в подпрограмму, т. е. мы можем переходить из основного кода в код процедуры и изучать ее изнутри;

<F9> — задание точки останова в текущей позиции курсора;

<F10> — пошаговое выполнение кода (отладчик выполняет текущую команду и переходит к следующей);

<F11> — возврат в точку вызова;

<F12> — выход из текущей функции или библиотеки DLL (переход `p ret`).

Ну вот, теперь вы знаете, что такое SoftICE. Осталось совсем чуть-чуть — познакомиться с основными брекпойнтами.

Наиболее важные точки останова

От правильной установки брейкпоинта напрямую зависит количество времени, затраченного на исследование защиты. В табл. 4.1 приведены только наиболее важные и часто используемые брекпойнты. Помните только, что не все брекпойнты работают во всех версиях Windows. Например, брекпойнт `bpx hmemcpy`, который прекрасно работал в Windows 9*x*, совсем не работает в Windows XP, т. к. в этой ОС нет функции `hmemcpy`. Это надо понимать.

Таблица 4.1. *Основные брекпойнты*

Назначение	Брекпойнт

Общее	`bpx memcpy`
	`bpx MessageBox`
	`bpx MessageBoxExA`
	`bpx MessageBeep`
	`bpx SendMessage`
	`bpx GetDlgItemText`
	`bpx GetDlgItemInt`
	`bpx GetWindowText`
	`bpx GetWindowWord`
	`bpx GetWindowInt`
	`bpx DialogBoxParamA`
	`bpx CreateWindow`
	`bpx CreateWindowEx`
	`bpx ShowWindow`
	`bpx UpdateWindow`
	`bmsg xxxx wm_move`
	`bmsg xxxx wm_gettext`
	`bmsg xxxx wm_command`
	`bmsg xxxx wm_activate`

Применяются при защитах, зависящих от времени	`bpx GetLocalTime`
	`bpx GetFileTime`
	`bpx GetSystemtime`
	`bpx GetTickCount`
	`bpx FileTimeToSystemTime`

Таблица 4.1 *(окончание)*

Назначение	Брекпойнт

Применяются при работе с CD-ROM (при "привязывании" программ к диску	`bpx GetFileAttributesA`
	`bpx GetFileSize`
	`bpx GetDriveType`
	`bpx GetVolumeInformation`
	`bpx GetLastError`
	`bpx ReadFile`

Применяются при работе с файловыми защитами	`bpx ReadFile`
	`bpx WriteFile`
	`bpx CreateFile`
	`bpx SetFilePointer`
	`bpx GetSystemDirectory`

Применяются при работе с INI-файлами	`bpx GetPrivateProfileString`
	`bpx GetPrivateProfileInt`
	`bpx WritePrivateProfileString`
	`bpx WritePrivateProfileInt`

Используются при работе с реестром	`bpx RegCreateKey`
	`bpx RegDeleteKey`
	`bpx RegQueryValue`
	`bpx RegQueryValueEx`
	`bpx RegCloseKey`
	`bpx RegOpenKey`

Применяются при отслеживании изменения флагов	`bpx cs:eip if EAX==0`

Интерактивный дизассемблер IDA

Напомню, зачем нужен дизассемблер, — он позволяет получить ассемблерный текст программы из машинного кода. Дизассемблеры называются так потому, что они занимаются дизассемблированием программы. *Дизассемблирование* — перевод двоичных данных программы в удобные для восприятия ассемблерные (мнемонические) инструкции. Однако это вовсе не означает, что любую программу можно перевести обратно на язык ассемблера, изменить

и откомпилировать заново. Это связано с тем, что ассемблирование — процесс однонаправленный, невозможно синтаксически отличить константы от адресов памяти, сегментов и смещений. В ряде случаев можно получить ассемблерный листинг программы, но это скорее исключение, чем правило.

Сейчас многие дизассемблеры могут определять имена вызываемых программой API-функций. IDA отличается от других дизассемблеров тем, что он способен опознавать имена не только API-функций, но и функций из MFC (Microsoft Foundation Classes) — стандартной библиотеки, используемой программами, написанными на Visual C++, и из OWL (Object Windows Library) — стандартной библиотеки, используемой программами, написанными на Borland C++, а также стандартных функций языка Си (таких как `fread()`, `strlen()` и т. д.), включенных в код программы.

Теперь несколько слов о дизассемблерах.

Дизассемблеры бывают двух видов — пакетные и интерактивные. В *пакетных* дизассемблерах анализ исследуемой программы производится автоматически на основе выбранных настроек. В *интерактивных* дизассемблерах можно контролировать весь процесс дизассемблирования.

Поясню это на примере. Мы запускаем IDA и дизассемблируем в нем программу. В ходе исследований часто обнаруживается, что некий кусок кода был дизассемблирован не совсем верно. Это можно исправить нажатием двух клавиш — достаточно установить курсор на начало некорректно дизассемблированного участка, нажать клавишу <U> для того, чтобы пометить этот участок как неисследованный, а потом нажать клавишу <A> (ASCII). Можно помечать участки и как код (клавиша <C>), как массив (клавиша <*>) и как данные (клавиша <D>). Если в команде

```
mov EAX, 123456789
```

вам не нравится представление числа `123456789`, то можно это число перевести в двоичный, восьмеричный, шестнадцатеричный формат или в код ASCII.

В IDA есть и такая замечательная возможность, как переименование невнятных меток, имен функций (например, `loc_43104E`) в более осмысленные (например, `func_secur`). И самое главное — такое изменение распространится на все участки кода, ссылающиеся на данную метку. Если понадобится, таким же способом можно добавить собственный комментарий (<;>). Как видите, все это намного облегчает дизассемблирование программы.

К пакетным дизассемблерам относится, например, Sourcer, являвшийся в свое время практически единственным дизассемблером. К интерактивным относятся, например, дизассемблеры IDA и hiew.

Пакетные дизассемблеры намного проще в управлении. Однако они имеют очень большой недостаток — ограничения, например, они практически бессильны против простых антиотладочных приемов и простых защит. Однако хочу сказать, что как раз в силу своей простоты они в свое время и получили такое распространение. К сожалению, сегодня они практически бесполезны. Они еще могут пригодиться для простого дизассемблирования своей программы, но их век уже окончен. На смену им пришли интерактивные дизассемблеры.

Простейший из интерактивных дизассемблеров это hiew. Про него мы еще поговорим. Скажу только, то что с его помощью идеально вскрываются компактные защиты и анализируются программы небольшого размера.

Дизассемблер IDA — это уникальный, единственный в своем роде инструмент, сочетающий в себе не только мощное интерактивное дизассемблирование, но и поддержку очень удобной навигации по анализируемому файлу. Как ни парадоксально, но именно из-за мощи IDA и применяют чаще, чем hiew. Главная причина этого — отсутствие у IDA какой-либо документации, кроме контекстной помощи на английском языке.

Но сейчас все сильно изменилось. А причиной всему легендарная книга К. Касперски "Образ мышления — дизассемблер IDA". Всем, кто собирается серьезно заниматься реверсингом, я настоятельно рекомендую ее приобрести. А в этой книге я опишу только основные функции дизассемблера IDA и приемы работы с ним — тот минимум, благодаря которому вы сможете исследовать программы.

Итак, начинаем. Первый вопрос, какой задают все, кто становится на нелегкий путь реверсинга: где взять IDA? Отвечаю: на сайте **www.datarescue.com**, который занимается поддержкой и продажей IDA, вы всегда найдете последнюю версию программы.

Установка программы обычно не вызывает никаких проблем. Достаточно распаковать дизассемблер в удобный для вас каталог, и он готов к работе (рис. 4.13).

Для того чтобы открыть в IDA файл для дизассемблирования, есть несколько способов:

- перетащить файл на главное окно с помощью мыши;
- передать файл в качестве параметра при запуске IDA;
- в меню **File** дизассемблера IDA выбрать команду **Open File**.

Как только файл откроется, IDA начнет процесс автоматического анализа файла, по окончании которого пользователь будет проинформирован об этом (рис. 4.14).

Рис. 4.13. Главное окно дизассемблера IDA

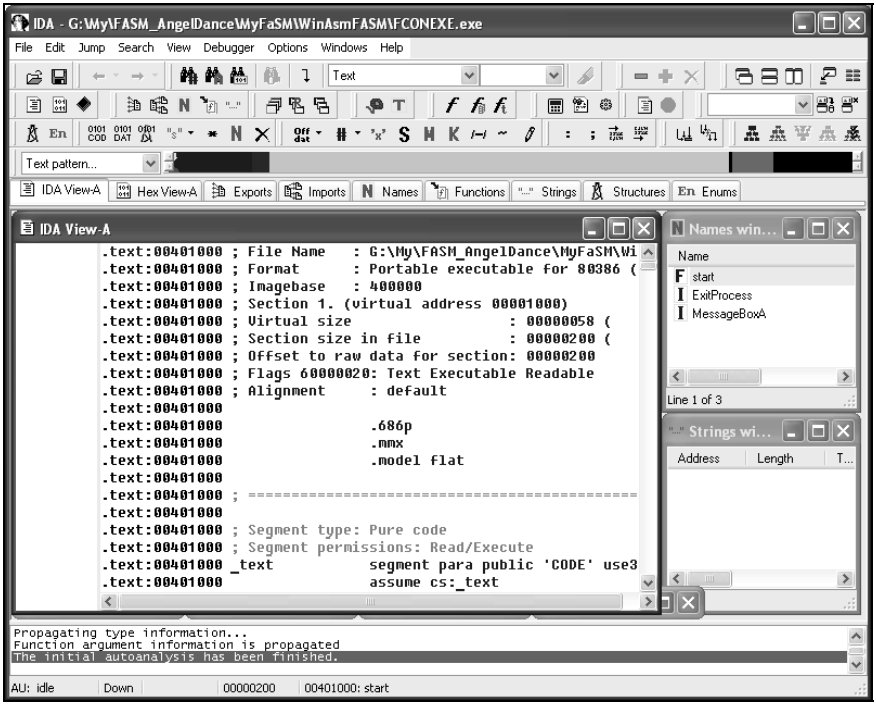

Рис. 4.14. Завершение автоматического анализа файла

После этого мы можем сделать очень полезную вещь: создать MAP-файл, который мы сможем загрузить в отладчик OllyDbg. Это поможет нам при исследовании программ, т. к. все имена функций, все метки и комментарии, которые у нас есть, появятся в отладчике. Для создания MAP-файла нужно выбрать команду **File\Produce file\Create MAP file** главного меню IDA. После этого следует указать место, где будет сохранен наш MAP-файл, и имя MAP-файла. Появится окно, в котором IDA спросит, что же именно мы хотим сохранить в MAP-файле (рис. 4.15). Я рекомендую сохранять все, т. е. установить все флажки, как показано на рис. 4.15.

Рис. 4.15. Создание MAP-файла

Нажмите кнопку **OK**, и MAP-файл будет создан.

Если теперь открыть в отладчике файл, который требуется исследовать, и загрузить туда же наш MAP-файл, то нам не придется гадать, что же скрывается под такой "информативной" функцией файла, как

```
call 4672F78
```

Мы увидим нечто вроде

```
call <CrackME_ ExitProg>
```

Согласитесь, что это намного удобнее и полезнее при отладке программы.

Ну, это я забегаю вперед, а пока вернемся к IDA.

После того как файл был проанализирован, чтобы посмотреть участок кода, на который ссылается некий условный/безусловный переход или вызов функции, достаточно навести курсор на соответствующую строку и нажать клавишу <Enter>.

Примечание

Чтобы вернуться к исходной точке в листинге, нажмите клавишу <Esc>.

В IDA есть некоторые полезные функции. Выбрав команду **View\Functions** главного меню, мы увидим окно, которое содержит описание всех найденных в программе функций: имя функции, ее начальный адрес, сегмент, которому принадлежит функция, ее длину и тип. Возможные типы функций:

- R — с возвратом в основную программу;
- F — дальний вызов;
- L — библиотечная функция;
- S — статическая функция;
- B — функция с передачей параметров через стек.

Для того чтобы перейти к требуемой функции, установите курсор на ее имя и нажмите клавишу <Enter>. Это очень полезно для исследования программы (я покажу вам все немного позднее, когда мы будем рассматривать разнообразные типы защиты и способы их обхода).

Иногда бывает нужно найти в программе строки, которые она выводит на экран. Например, вас в программе заинтересовала строка `'Serial_is_False'`. Выберите команду **View\Names** главного меню и найдите свою строку.

> **Примечание**
>
> В IDA строки автоматически именуются так: если это ASCII-строка, то начальная буква имени — "a" (ASCII), далее идет начальная часть строки. Если же буквы "a" нет, то это имя функции.

Для приведенного примера это выглядит как `aSerial_is_False`. Если вы встретили такое имя, то это имя нужной строки. Для перехода к соответствующей строке надо нажать клавишу <Enter>.

Как я уже говорил, у многих компиляторов есть свои уникальные особенности, свои встроенные функции. Например, программы, скомпилированные в Borland Builder C++ и Microsoft Visual C, сильно различаются. В результате при дизассемблировании мы получим множество функций с непонятными именами типа `sub_701347` и т. п. Понятно, что разобраться в том, что же делает данная функция, довольно сложно. Однако IDA потому и называется интерактивным дизассемблером, что успешно справляется с этой проблемой. При анализе файла IDA пытается автоматически определить тип использованного компилятора и загрузить соответствующую сигнатуру для него. Под *сигнатурой* следует понимать некую информацию, уникальную для каждого компилятора. В большинстве случаев это удается, если же этого не произошло, то откройте окно сигнатур и нажмите клавишу <Insert>. В появившемся списке доступных сигнатур выберите ту, что на ваш взгляд является наиболее

правильной. После этого сигнатура будет загружена и вид листинга изменится, его станет читать намного легче (или сложнее, если сигнатура была выбрана неправильно). Правда, для того чтобы менять тип компилятора, надо знать, каким компилятором скомпилирована программа. Выбрав в меню **View** команду **Signatures**, можно просмотреть уже загруженные сигнатуры. В результате IDA превращает неудобочитаемые имена вроде `call 413f5` в `call_main` или `call_exitProg`, что, согласитесь, намного удобнее.

Это была теория. А теперь посмотрим на практике, как же все это работает. Чтобы было интереснее, мы загрузим в IDA тот же файл, что мы использовали, когда изучали отладчик Turbo Debugger. Запускаем IDA, перетаскиваем в него наш файл и получаем окно, показанное на рис. 4.16.

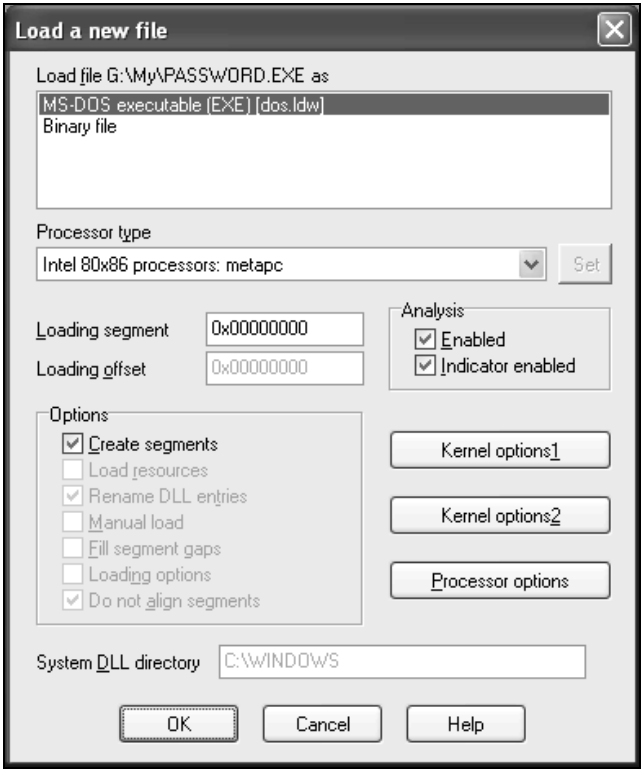

Рис. 4.16. Окно загрузки файла

В этом окне мы можем выбрать тип процессора, для которого написан наш файл, формат отображения файла и т. д. Я рекомендую оставить все как есть и нажать кнопку **OK**, чтобы загрузить файл в дизассемблер.

Как видите, дизассемблер представил нам подробный листинг нашей программы (листинг 4.2).

Листинг 4.2. Часть листинга программы из дизассемблера

```
seg000:0000 ; ------------- S U B R O U T I N E ------------------
seg000:0000
seg000:0000
seg000:0000 start   proc near
seg000:0000         mov     ax, seg dseg
seg000:0003         mov     ds, ax
seg000:0005         assume ds:dseg
seg000:0005         mov     es, ax
seg000:0007         assume es:dseg
seg000:0007         mov     ah, 9
seg000:0009         mov     dx, 34h ; '4'
seg000:000C         int     21h     ; DOS - PRINT STRING
seg000:000C                         ; DS:DX -> string terminated by "$"
seg000:000E         mov     dx, 31h ; '1'
seg000:0011         mov     ah, 0Ah
seg000:0013         int     21h     ; DOS - BUFFERED KEYBOARD INPUT
seg000:0013                         ; DS:DX -> buffer
seg000:0015         mov     si, 0
seg000:0018         mov     cx, 7
seg000:001B
seg000:001B loc_1B:                 ; CODE XREF: start+21□j
seg000:001B         xor     byte ptr [si+2Bh], 34h
seg000:0020         inc     si
seg000:0021         loop    loc_1B
seg000:0023         cld
seg000:0024         mov     si, 2Bh ; '+'
seg000:0027         mov     di, 33h ; '3'
seg000:002A         mov     cx, 7
seg000:002D         repe cmpsb
seg000:002F         jmp     short loc_33
seg000:0031 ; ---------------------------------------------
seg000:0031         jnz     short loc_3D
seg000:0033
```

```
seg000:0033 loc_33:                 ; CODE XREF: start+2F□j
seg000:0033         mov     ah, 9
seg000:0035         mov     dx, 0
seg000:0038         int     21h       ; DOS - PRINT STRING
seg000:0038                           ; DS:DX -> string terminated by "$"
seg000:003A         jmp     short loc_49
seg000:003A ; ---------------------------------------------------
seg000:003C         db 90h
seg000:003D ; ---------------------------------------------------
seg000:003D
seg000:003D loc_3D:                   ; CODE XREF: start+31□j
seg000:003D         mov     ah, 9
seg000:003F         mov     dx, 11h
seg000:0042         int     21h     ; DOS - PRINT STRING
seg000:0042                         ; DS:DX -> string terminated by "$"
seg000:0044         jmp     short loc_49
seg000:0044 ; ---------------------------------------------------
seg000:0046                 db 90h, 0EBh, 0E4h
seg000:0049 ; ---------------------------------------------------
seg000:0049
seg000:0049 loc_49:                 ; CODE XREF: start+3A□j
seg000:0049                         ; start+44□j
seg000:0049         mov     ax, 4C00h
seg000:004C         int     21h   ; DOS - 2+ - QUIT WITH EXIT CODE (EXIT)
seg000:004C start   endp          ; AL = exit code
seg000:004C
seg000:004C ; ---------------------------------------------------
seg000:004E         align 4
seg000:004E seg000  ends
seg000:004E
dseg:0000 ; ===================================================
```

В расположенном справа так называемом *поле комментария* IDA подписал назначение всех ключевых операторов и функций, что, согласитесь, довольно удобно при анализе программы.

В самом начале мы видим такие строки программы:

```
seg000:0000         mov     ax, seg dseg
seg000:0003         mov     ds, ax
```

Если подвести мышку к `dseg` и нажать клавишу <Enter>, то мы окажемся в начале секции данных. Для того чтобы вернуться назад, надо нажать клавишу <Esc>.

А вот как выглядит секция данных в дизассемблере IDA (листинг 4.3).

Листинг 4.3. Внешний вид секции данных

```
dseg:0000 ; ========================================================
dseg:0000
dseg:0000 ; Segment type: Pure data
dseg:0000 dseg            segment para public 'DATA' use16
dseg:0000                 assume cs:dseg
dseg:0000                 db  0Ah
dseg:0001                 db  0Dh
dseg:0002 aPasswordIsOk   db 'PASSWORD IS OK$'
dseg:0011                 db  0Ah
dseg:0012                 db  0Dh
dseg:0013 aPasswordIsNotC db 'PASSWORD IS NOT CORRECT$'
dseg:002B aQmgD           db '□qmg`{d$'
dseg:0033                 db  0Ah
dseg:0034 aInput          db 'Input'
dseg:0039                 db  20h
dseg:003A aPassword       db 'PASSWORD:$'
dseg:0044                 align 10h
dseg:0044 dseg            ends
dseg:0044
seg002:0000 ; ========================================================
```

Как видно из листинга, дизассемблер верно определил строковые переменные, используемые в программе. Но самое интересное — дизассемблер верно определил то, что тоже является строковой переменной:

```
dseg:002B aQmgD           db '□qmg`{d$'
```

А, как мы знаем, это зашифрованный пароль. Для сравнения попробуйте найти эту строку в программе Sourcer.

Как видите, IDA позволяет сделать детальный анализ файла. Конечно, для того чтобы разобраться во всех функциях этого дизассемблера, потребуется не одна книга, но на первых порах вам это и не нужно. Этих начальных знаний вам хватит для того, чтобы начать исследовать программы. Если кто-то

хочет более детально изучить дизассемблер IDA, обратитесь к документации, поставляемой с программой.

Отладчик OllyDbg

В этой главе я расскажу о таком мощном отладчике как OllyDbg (автор Oleh Yuschuk).

Если кому-то не терпится начать исследовать программы, то вы можете перейти к следующей главе, но тогда вы не поймете "как все это работает". Я рекомендую просто бегло прочитать данную главу, чтобы составить себе общее мнение об отладчике. А в дальнейшем вы сможете вернуться к ней, когда в этом возникнет необходимость.

Итак, отладчик OllyDbg. Это мой любимый отладчик уровня приложений (ring-3), поэтому я расскажу о нем более подробно. Последней на момент написания книги была версия OllyDbg v1.10 — последняя версия проекта, после чего проект будет закрыт, но автор обещал в скором времени выпустить OllyDbg 2.00, который он разработает "с нуля". Так что будем ждать. Для OllyDbg выпущено и выпускается очень много плагинов, позволяющих превратить этот отладчик в поистине незаменимый инструмент отладки.

Ну, а пока я расскажу вам об этом прекрасном отладчике. Для начинающих, да и для более опытных крекеров, на мой взгляд, возможностей этого отладчика хватает с лихвой. Судите сами — вот краткий перечень возможностей OllyDbg:

- 32-битный отладчик, анализирующий на уровне ассемблера, с интуитивным интерфейсом;
- полная поддержка Unicode (особенно важно!);
- полный анализатор кода: обозначение процедур, переходов, вызовов API, нахождение и обозначение констант и строк;
- подсветка кода: OllyDbg может подсвечивать различные типы команд, операнды и т. д.; вы можете сами создавать схемы подсветки;
- поддержка определяемых пользователем меток, комментариев, описаний функций и т. д.
- запись сделанных изменений в файл;
- не требуется инсталляция;
- поддержка плагинов;
- "узнавание" более 2300 стандартных API функций и функций языка C;

- автоматическое информирование по API из внешнего файла помощи;
- показ изменений в коде;
- поиск выборочных команд и бинарного кода по маске.
- поиск в памяти.
- поиск вызовов констант или чтение адресного пространства.
- просмотр и модификация памяти, установка точек останова и остановки программ "на лету"

…и многое другое.

Как видите, функциональность впечатляет. Сначала я опишу сам отладчик, а затем — дополнительные расширения (плагины) для него, позволяющие значительно расширить функциональность отладчика, повысить его надежность, ну и, наконец, просто облегчить процесс реверсинга.

Главное окно программы

На рис. 4.17 показано главное окно программы, очень похожее на главные окна других отладчиков (TD, SoftICE). Оно содержит:

- окно CPU (главное для крекеров);
- окно информации;
- окно регистров;
- панель стека;
- меню.

Рассмотрим их подробнее.

Окно CPU

Это самое важное окно отладчика, т. к. в нем ведется собственно отладка исследуемой программы. В нем есть четыре столбца: адрес, HEX-дамп, дизассемблированные команды и комментарии. Последний столбец может также отображать связанный исходный код или профиль выполнения.

Как видим, все инструкции в этом окне приведены в дизассемблированном виде. Вы можете перемещать указательную строку с помощью мыши или клавиш перемещения курсора. Можно выбирать данные, которые надо скопировать, команду, которую надо изменить, место установки брекпойнта и т. д. Слева красной стрелкой указан адрес следующей выполняемой команды.

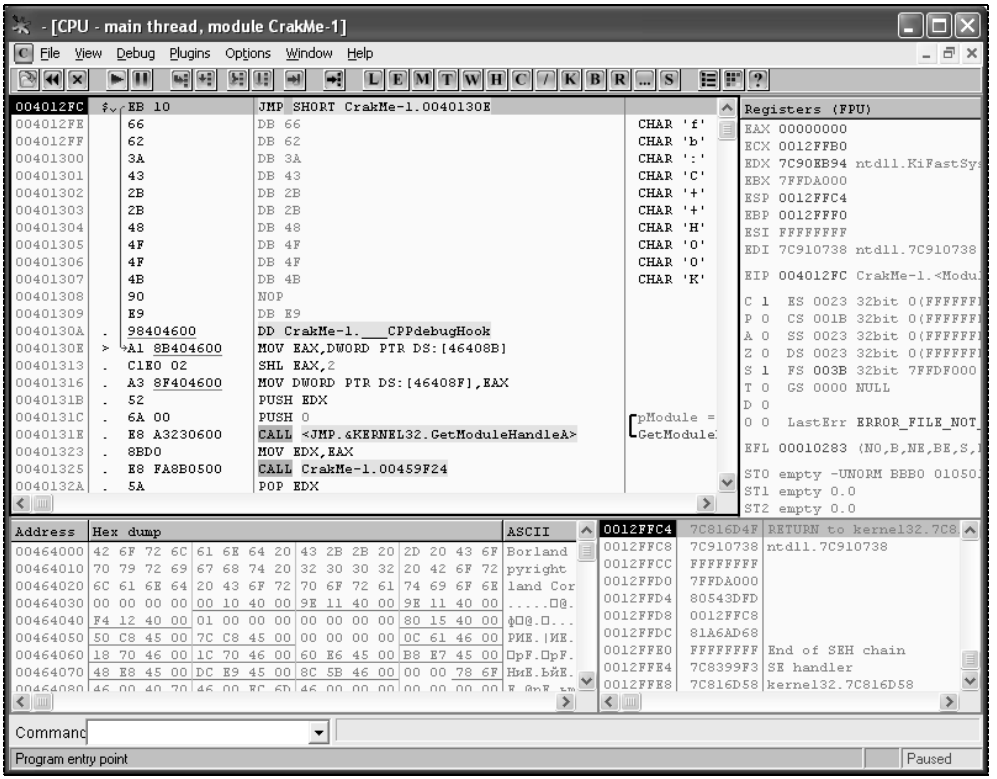

Рис. 4.17. Главное окно отладчика OllyDbg

Окно информации

Информационное окно в окне CPU расшифровывает параметры выбранной команды. Оно также отображает неявные параметры, например, содержимое AL и ECX в команде `REPE SCASB` или верх стека в команде `RETN`. Для проанализированного кода отображается список команд, с которых осуществляется переход на текущую команду. Если программа скомпилирована с отладочной информацией, то также отображается исходная строка, соответствующая выбранному фрагменту. Отсюда же можно переходить по адресу в дамп, изменять параметры, открыть файл с исходным кодом и многое другое.

Примечание

Дамп — копия содержимого оперативной памяти (программы), сохраненная на диске (или другом внешнем носителе).

Надо помнить, что все регистры используются программами очень интенсивно, их содержимое непрерывно изменяется. По умолчанию OllyDbg предполагает, что регистры верны только для текущей выполняемой команды. Помните об этом.

Окно регистров

В окне регистров отображаются все регистры процессора — регистр флагов и др. Эти регистры можно изменять прямо "на лету". Для того чтобы изменить регистр, надо щелчком выделить его и щелкнуть на нем еще раз. На экране появится окно, показанное на рис. 4.18. В этом окне можно ввести новое значение, и оно тут же будет применено.

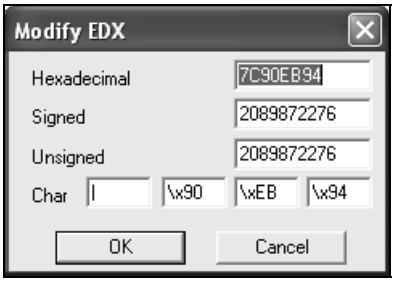

Рис. 4.18. Окно изменения регистра

Для того чтобы изменить значение флага, сделайте на нем двойной щелчок.

Панель стека

Панель стека (расположена в левой нижней части главного окна программы) тоже очень важна. Она отображает содержимое памяти или файла. Можно выбрать один из нескольких предопределенных форматов отображения: byte, text, integer, float, address, disassembly или PE Header. Когда мы изучали отладчик Turbo Debugger, я объяснил вам, как найти пароль к программе. Там было наглядно показано значение панели стека. Ни в коем случае не стоит ею пренебрегать. В этой панели мы можем изменять значения нужных байт в памяти, ставить брекпойнты и т. д.

Меню

Теперь давайте познакомимся с меню. Оно находится в верхней части окна OllyDbg и предоставляет доступ к различным командам отладчика. Мы рассмотрим наиболее важные элементы меню.

Начнем по-порядку.

Меню *File*

Это меню содержит стандартные для любой программы Windows команды:

- **Open** — открыть файл для отладки;
- **Attach** — присоединиться к уже запущенному процессу для отладки;
- **Exit** — выйти.

При выборе команды **Attach** появится окно (рис. 4.19), в котором следует выбрать запущенный процесс для отладки и нажать кнопку **Attach**.

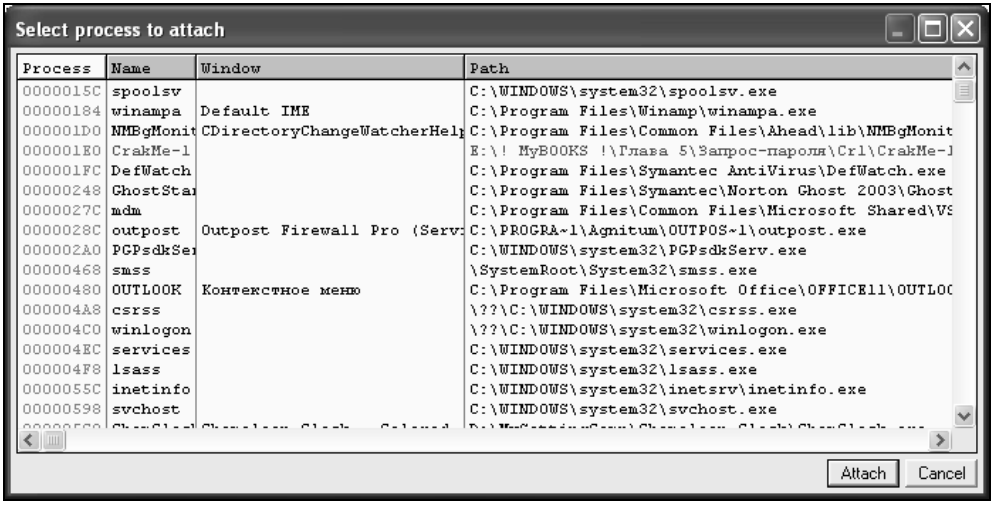

Рис. 4.19. Выбор процесса для отладки

Меню *View*

Это меню содержит команды (в скобках указана соответствующая команде комбинация горячих клавиш):

- **Log** (<Alt>+<L>) — позволяет просмотреть журнал (лог) загрузки OllyDbg, плагинов, файла и т. д. (рис. 4.20);
- **Executable modules** (<Alt>+<E>) — открывает подменю, позволяющее просмотреть список модулей, используемых исследуемой в отладчике программой;
- **Memory** (<Alt>+<M>) — позволяет просмотреть карту памяти процесса.

  Это очень полезная возможность отладчика. Она сберегает много времени при исследовании программ. Далее я покажу, как пользоваться ею при

анализе программы, а сейчас опишу наиболее важные столбцы окна **Memory map** (рис. 4.21):

- **Address** — адрес, по которому загружены различные секции файла, библиотек DLL;
- **Size** — размер секции в памяти компьютера;
- **Owner** — имя модуля;
- **Section** — название секции;
- **Contains** — содержимое секции;

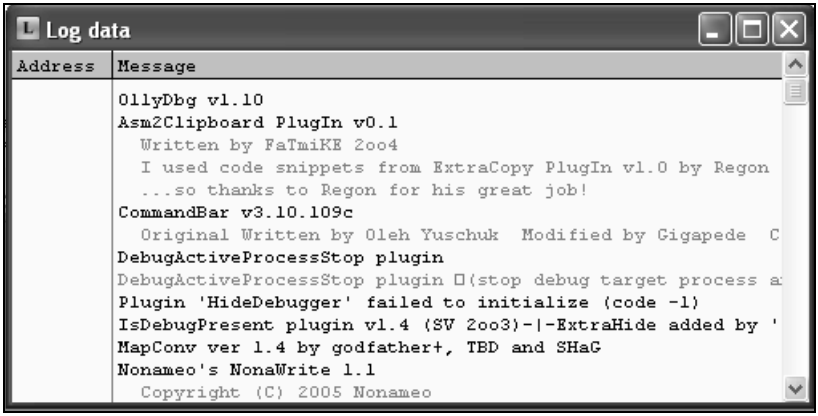

Рис. 4.20. Просмотр журнала

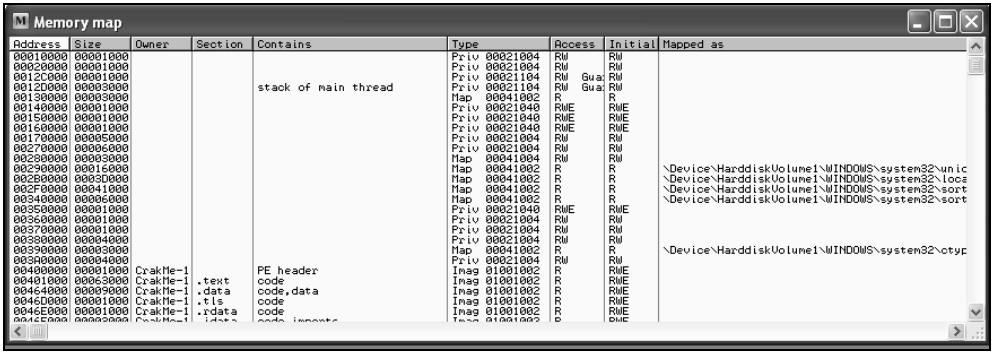

Рис. 4.21. Карта памяти

- **Threads** — позволяет просматривать все используемые программой потоки; особенность отладчика OllyDbg в том, что с его помощью можно просто и эффективно управлять потоками;

- **Windows** — открывает список всех окон, принадлежащих отлаживаемому приложению, и их наиболее важных параметров. Можно устанавливать брекпойнт на сообщение, которое выводит исследуемая программа (или на группу сообщений), просмотреть всю информацию об окнах, классах и других объектах, используемых программой. Для этого программа должна быть запущена (рис. 4.22);

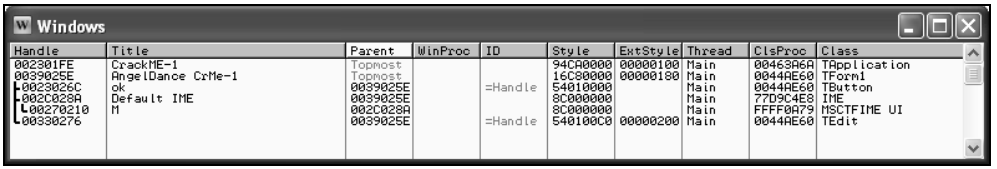

Рис. 4.22. Информация об окнах, классах и т. д.

- **Handles** — позволяет просмотреть список используемых процессом идентификаторов (рис. 4.23);

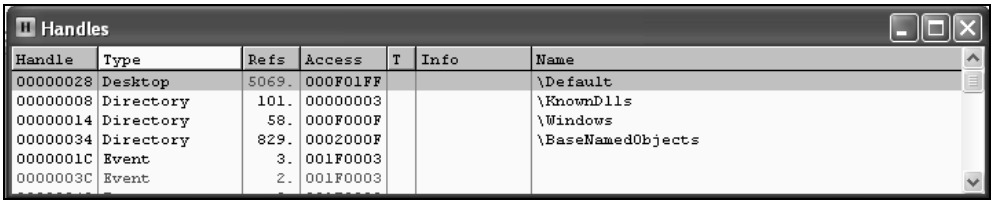

Рис. 4.23. Информация о маркерах

- **CPU** (<Alt>+<C>) — открывает окно CPU;
- **SEH Chain** — позволяет увидеть или установить брекпойнт на все объявленные в программе обработчики исключительных ситуаций (т. е. ошибок доступа к памяти, например, не выделенной, деления на 0 и т. п.);
- **Patches** (<Ctrl>+<P>) — позволяет просмотреть список всех изменений, которые мы внесли в программу. Там же можно выключить изменения или совсем их отменить и т. д.;
- **Call Stack** (<Alt>+<K>) — позволяет увидеть, что именно программа помещает в стек (рис. 4.24) — очень полезный режим;
- **Breakpoints** (<Alt>+<B>) — позволяет просмотреть все установленные нами брекпойнты, отключить/включить их, а также удалить совсем;
- **Watches** — вызывает окно инспектора, в котором можно показывать содержимое некоторой переменной, массива или даже выбранные элементы структуры;

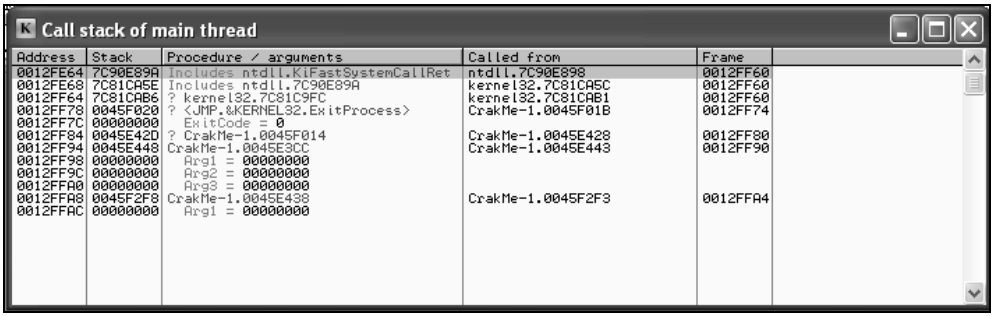

Рис. 4.24. Информация о стеке программы

- **Run Trace** — позволяет просмотреть журнал (лог) трассировки;
- **Source** — позволяет просмотреть исходный текст программы;
- **Source file** — позволяет просмотреть список исходных файлов;
- **File** — открывает шестнадцатеричный редактор файлов;
- **Text File** — позволяет просмотреть листинг файла.

Меню *Debug*

Содержит следующие команды:

- **Run** — запускает отлаживаемую программу на исполнение;
- **Pause** — приостанавливает исполнение программы;
- **Restart** — перезапускает отлаживаемую программу;
- **C0lose** — завершает работу с отлаживаемой программой;
- **Step into** (<F7>) — запускает пошаговое выполнение программы с заходом в процедуру;
- **Step over** (<F8>) — запускает пошаговое выполнение программы без захода в процедуру;
- **Animate into** — эквивалент постоянно нажатой клавиши <F7>;
- **Animate over** — эквивалент постоянно нажатой клавиши <F8>;
- **Execute till return** — предписывает выполнить программу до выхода из подпрограммы;
- **Trace into** — автотрассировка с заходом в подпрограммы;
- **Trace over** — автотрассировка без захода в подпрограммы;
- **Set condition** — позволяет назначить условие для автотрассировки;

- **Close run trace** — выполняет останов автотрассировки;
- **Hardware breakpoints** — позволяет просмотреть немаскируемые брекпойнты[1].

Меню *Plugins*

Это меню содержит дополнительные расширения отладчика OllyDbg. У каждого крекера оно свое, т. к. включает только ему удобные и необходимые плагины.

Меню *Option*

В этом меню осуществляется настройка программы. Оно содержит команды:

- **Appearance** — открывает одноименное диалоговое окно настройки интерфейса отладчика; доступ к параметрам настройки организован с помощью вкладок окна:

  Примечание

  Я рекомендую начинающим настроить отладчик, как показано на следующих рисунках. Когда вы начнете использовать отладчик более осмысленно, вы всегда сможете настроить его так, как вам хочется. Оно состоит из следующих подменю.

  - **General** (основные), рис. 4.25;
  - **Defaults** (по умолчанию), рис. 4.26;
  - **Dialogs** (диалоговые окна), рис. 4.27;
  - **Directories** (каталоги), рис. 4.28;
  - **Fonts** (шрифты), рис. 4.29;
  - **Colors** (цвета);
  - **Code highlighting** (подсветка кода);
- **Debugging options** (<Alt>+<O>) — открывает диалоговое окно настройки отладки.

  Это меню слишком обширно, чтобы полностью привести его на страницах данной книги, поэтому я ограничусь описанием только наиболее важных параметров, необходимых для того, чтобы начать отлаживать программы:

  - вкладка **Security**:
    - **Warn when breakpoint is outside the code section** (Предупреждать о том, что устанавливаемый брекпойнт находится вне секции кода) — конечно, эта опция не должна быть выбрана;

[1] Hardware (HW) — аппаратные средства, "железо". Можно устанавливать брекпойнты на аппаратные прерывания (прерывания от "железа").

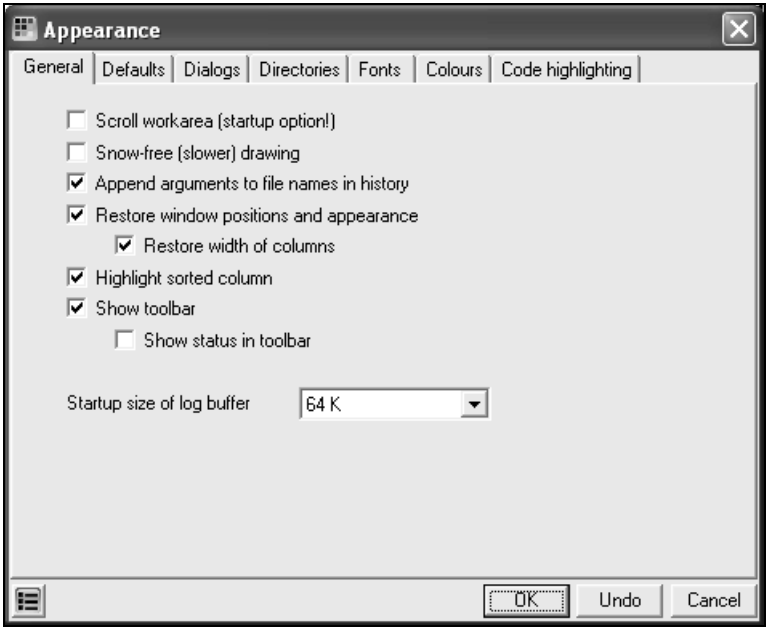

Рис. 4.25. Основные настройки OllyDbg

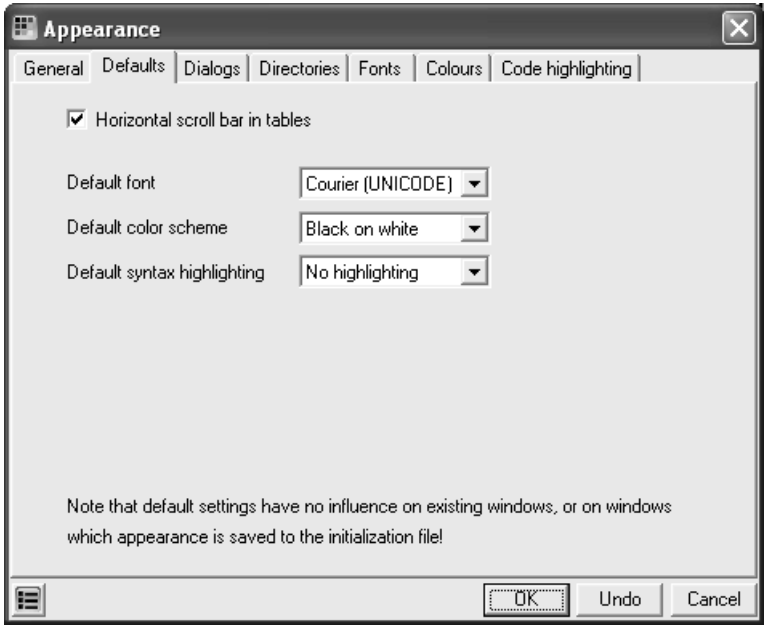

Рис. 4.26. Настройки схем

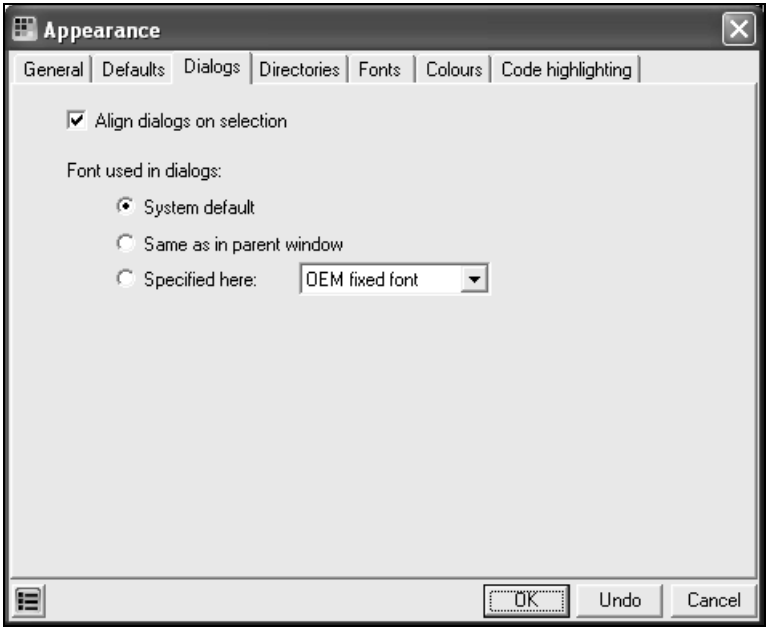

Рис. 4.27. Настройки диалогов

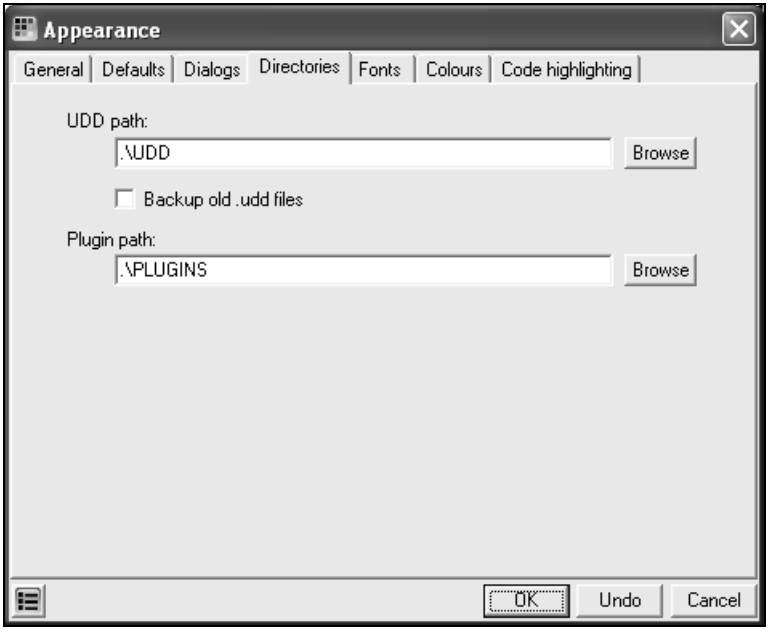

Рис. 4.28. Настройки путей

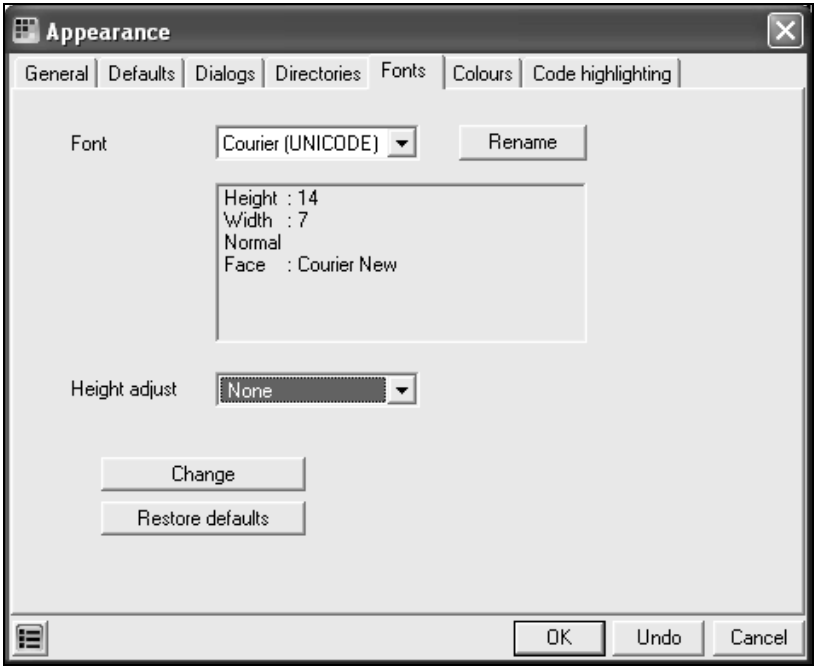

Рис. 4.29. Настройки шрифтов

- ◊ **Warn when terminating active process** (Предупреждать о завершении текущего процесса) — рекомендую выбрать данную опцию;
- ◊ **Warn if not administrator** (Предупреждать, если на данном компьютере у вас нет прав адинистратора) — рекомендую выключить данную опцию;

- вкладка **Debug**:
  - ◊ **Set high priority when tracing debugged process** (Устанавливать высокий приоритет для приложения при трассировке) — рекомендую вам выбрать данную опцию;
- вкладка **Events**:
  - ◊ **Make first pause at** (Сделать первую остановку на) — предлагается три варианта: **System breakpoint** (Системный брейкпоинт), **Entry point of main module** (Точка входа в главный модуль), **WinMain (if location is known)** (Главное окно). Тут каждый выбирает себе оптимальный вариант. Я рекомендую выбрать точку входа в главный модуль;

- вкладка **Exceptions**:
  - ◊ **Ignore memory access violations in KERNEL32** (Игнорировать ошибки памяти в Kernel32) — рекомендую включить данную опцию;
  - ◊ **Int3 breaks** (Не останавливаться на командах int3) — это очень полезная опция, рекомендую ее включить;
- вкладка **Trace**:
  - ◊ **Size of run trace buffer** (Выбрать размер буфера для трассировки) — лучше всего ставить максимальное значение;
  - ◊ **Log commands** (Вести лог работы трассировщика) — каждый выбирает сам; рекомендую включить данную опцию;
  - ◊ **Always trace over system DLLs** (Всегда трассировать мимо системных библиотек) — настоятельно рекомендую выбрать данную опцию;
  - ◊ **After Executing till RET, step over RET** (Останавливаться при нажатии <Ctrl>+<F9> не на команде `ret`, а после ее выполнения) — рекомендую включить данную опцию;
  - ◊ **Just-in-time debugging** (Назначить отладчиком по-умолчанию) — OllyDbg будет запускаться, когда какое-нибудь приложение вызовет ошибку (у меня эта опция выбрана);
  - ◊ **Add to Explorer** (Добавить в контекстное меню проводника пункт для отладки программы) — у меня эта опция также выбрана.

Горячие клавиши

Знание горячих клавиш отладчика OllyDbg (табл. 4.2) очень поможет не только начинающим, но и более опытным крекерам.

Таблица 4.2. *Горячие клавиши отладчика OllyDbg*

Клавиша или комбинация клавиш	Действие отладчика
<F2>	Установить брекпойнт в выделенной строке
<F3>	Открыть файл и определить для него параметры
<F4>	Запустить до выделенного. На первой выбранной команде устанавливается однократный брекпойнт, выполнение отлаживаемой программы продолжается
<F5>	Развернуть/восстановить активное окно

Таблица 4.2 *(продолжение)*

Клавиша или комбинация клавиш	Действие отладчика
<F6>	Активизировать следующее окно
<F7>	Трассировать с заходом в подпрограммы
<F8>	Трассировать без захода в подпрограммы
<F9>	Запуск программы на выполнение
<F10>	Открыть контекстное меню
<F12>	Приостановить запуск
<Ctrl>+<F1>	Открыть раздел справки, связанный с символическим именем в первой выбранной строке при условии, что файл справки API выбран
<Ctrl>+<F2>	Перезапустить отлаживаемую программу
<Ctrl>+<F4>	Закрыть активное окно
<Ctrl>+<F5>	Открыть исходный файл, соответствующий первой выбранной команде
<Ctrl>+<F7>	Включить режим выполнения кода, как при постоянно нажатой клавише <F7>
<Ctrl>+<F8>	Включить режим выполнения кода, как при постоянно нажатой клавише <F8>
<Ctrl>+<F9>	Выполнить программу до выхода из подпрограммы
<Ctrl>+<F10>	Открыть контекстное меню
<Ctrl>+<F11>	Запустить автоматическую трассировку с заходом в подпрограммы
<Ctrl>+<F12>	Запустить автоматическую трассировку без захода в подпрограммы
<Alt>+<F2>	Закрыть файл
<Alt>+<F5>	Установить окно OllyDbg поверх всех окон. Это особенно полезно, если вы отлаживаете приложение, окно которого занимает собой весь экран. Повторное нажатие комбинации клавиш <Alt>+<F5> восстанавливает обычный режим отображения окна OllyDbg
<Alt>+<F7>	Перейти к предыдущей найденной ссылке
<Alt>+<F8>	Перейти к следующей найденной ссылке
<Shift>+<F2>	Установить брекпойнт с условием на первой выбранной команде

Таблица 4.2 *(продолжение)*

Клавиша или комбинация клавиш	Действие отладчика
<Shift>+<F4>	Установить брейкпоинт с условием дополнительной проверки значения некоторого выражения, когда условие выполнено
<Shift>+<F6>	Активизировать предыдущее окно
<Shift>+<F7>	Тот же, что <F7>, но если отлаживаемая программа остановилась на некотором исключении, OllyDbg сначала попробует передать управление обработчику, указанному в отлаживаемой программе
<Shift>+<F8>	То же, что <F8>, но если отлаживаемая программа остановилась на некотором исключении, OllyDbg сначала попробует передать управление обработчику, указанному в отлаживаемой программе
<Shift>+<F9>	Тот же, что <F9>, но если отлаживаемая программа остановилась на некотором исключении, OllyDbg сначала попробует передать управление обработчику, указанному в отлаживаемой программе
<Alt>+<B>	Открыть/восстановить окно **Breakpoints**
<Alt>+<C>	Открыть/восстановить окно **CPU**
<Alt>+<E>	Открыть/восстановить список модулей
<Alt>+<K>	Открыть/восстановить окно **Call stack**
<Alt>+<L>	Открыть/восстановить окно журнала
<Alt>+<M>	Открыть/восстановить окно **Memory map**
<Alt>+<O>	Открыть диалоговое окно **Options**
<Ctrl>+<A>	Провести анализ кода
<Ctrl>+<B>	Запуск бинарного поиска
<Ctrl>+<C>	Копировать данные
<Ctrl>+<E>	Редактировать выделенное в бинарном (или шестнадцатеричном) формате
<Ctrl>+<F>	Запуск поиска команды
<Ctrl>+<G>	Перейти к адресу
<Ctrl>+<P>	Открыть окно **Patches** (Изменения)
<Ctrl>+<R>	Найти ссылки на выбранную команду: просмотреть исполняемый код активного модуля и найти все ссылки (константы, переходы или вызовы) на первую выбранную команду. Далее можно использовать комбинации клавиш <Alt>+<F7> и <Alt>+<F8> для перехода по ссылкам. Для удобства упомянутая команда также включена в список ссылок

Таблица 4.2 *(окончание)*

Клавиша или комбинация клавиш	Действие отладчика
<Ctrl>+<S>	Выполнить поиск команды. Отображается диалоговое окно **Find command**, где можно ввести команду в формате ассемблера; ищется следующее вхождение этой команды
<Ctrl>+<T>	Открыть окно настройки автотрассировщика
<Ctrl>+<L>	Искать далее. Повторяется последний произведенный поиск
<Ctrl>+<J>	Перечислить все вызовы/переходы к текущему местоположению. Прежде чем использовать эту функцию, необходимо проанализировать код
<Ctrl>+<K>	Показать дерево вызовов, связанное с текущей процедурой. Прежде чем использовать эту функцию, необходимо проанализировать код
<Ctrl>+<N>	Открывает список имен, меток текущего модуля
<Shift>+<F8>	Продолжить трассировку программы, даже если возникла исключительная ситуация
<Shift>+<F9>	Продолжить запуск программы, даже если возникла исключительная ситуация
<+>	Если трассировка не активна, перейти к команде, которая должна выполняться следующей по логике программы; иначе перейти к следующей записи в данных отладки
<Ctrl>+<+>	Перейти к началу следующей процедуры
<->	Если трассировка не активна, перейти к предыдущей по логике программы команде; иначе перейти к предыдущей записи в данных отладки
<Ctrl>+<->	Перейти к началу предыдущей процедуры
<Пробел>	Отобразить диалоговое окно ассемблирования, где можно редактировать фактическую команду или вводить новые команды на языке ассемблера
Двоеточие (:)	Добавить метку. Отображается диалоговое окно для добавления новой или редактирования существующей метки
Точка с запятой (;)	Добавить комментарий. Отображается диалоговое окно для добавления нового или редактирования существующего комментария

Плагины для отладчика OllyDbg

Плагинов для OllyDbg великое множество (некоторые из них вы найдете на сопроводительном компакт-диске этой книги). Даже простое их перечисление займет не одну книгу. Я приведу тут лишь самые распространенные пла-

гины, которые помогут вам в процессе реверсинга, облегчая тяжелый труд крекера.

Несмотря на все многообразие плагинов, их можно разделить на группы:

- морально устаревшие или редко используемые;
- популярные плагины:
  - антиотладка
  - скриптовые плагины
  - дамперы
  - другие

Из морально устаревших я упомяну только одного представителя, который может помочь вам в деле реверсинга, — широко известный плагин Anti Asprotect_1.2x plugin for olly, применяемый для снятия такой известной защиты, как ASProtect. Сейчас, к сожалению, он почти не используется, т. к. появились новые версии ASProtect.

Рассмотрим популярные плагины подробнее.

Антиотладка

Эти плагины применяются для того, что бы скрыть процесс отладки от программы. Дело в том, что многие защиты определяют работу под отладчиком и либо не работают, либо работают неправильно. Для того чтобы этого избежать и добраться до внутренностей защиты, и применяются данные плагины. В *главе 7* я расскажу вам о методах противодействия отладчикам и дизассемблерам.

AntiDetectOllyforExeCryptor. Этот плагин разработан для борьбы с набирающим обороты протектором ExeCryptor (думаю, он может помочь и для скрытия от других протекторов).

Anti-anti. Очень хороший плагин, скрывающий OllyDbg от большинства попыток обнаружения, таких как patch PEB: IsDebuggerPresent, ProcessHeap (GetProcessHeap) и NtGlobalFlag; защищает от определения через EnableWindow; не позволяет изменять регистры DRx, скрывает от ZwSetInformationThread, ZwQueryInformationProcess.

В комплект поставки плагина входит проверочный файл, который продемонстрирует вам результаты работы отладчика с включенным и выключенным плагином.

AntiDetectOlly_v2.2.4. Данный плагин, разработанный нашим соотечественником TeSt, позволяет скрывать OllyDbg от протектора ExeCryptor.

Antidrx. Защищает регистры DRx от сброса и изменения, соответственно защищает HW-брекпойнты (аппаратные брекпойнты, которые позволяет устанавливать процессор, максимум — 4).

Hidedbg. Очередной плагин от loveboom, который скрывает OllyDbg. Поддерживает:

- скрытие отладчика от обнаружения: IsDebuggerPresent, NtGlobalFlag, ProcessHeapFlag;
- предотвращение обнаружения с помощью патчей: ZwQueryInformationProcess (UnhandledExceptionFilter), ZwSetInformationThread, CheckRemoteDebuggerPresent, OutputDebugStringA.

HideOD. Скрывает OllyDbg от обнаружения (подарок китайских друзей). Поддерживает: IsDebuggerpresent; NtGlobalFlags; HeapFlags; ForceFlags; SetDebugPrivilege; OutDebugStringA; CheckRemoteDebuggerPresent; ZwSetInformationThread; UnhandledExceptionFilter; Process32Next; ZwQueryInformationProcess; ZwSetInformationThread.

Fkrdtsc. Превосходный плагин; обезоруживает защиты, которые пытаются определить отладчик с помощью замеров между участками кода.

Hidecapt100. Скрывает заголовки (caption) окон OllyDbg. Это помогает обойти защиты, основанные на FindWindow.

HideDebugger123 by Asterix. Еще один плагин, который скрывает OllyDbg.

IsDebug&ExtraHide. Это плагин IsDebug, пропатченный нашим соотечественником TeSt для еще более хорошего скрытия OllyDbg.

IsDebuggerPresent1.4. Скрывает OllyDbg только от функции IsDebuggerPresent.

Olly_Invisible_0.9.0.6. Скрывает процесс OllyDbg, глобально перехватывает API-функции, присоединяя свою DLL к каждому процессу. Фактически плагин не работает как надо, т. к. рассчитан на более старую версию OllyDbg.

OllyGhost. Предназначен для скрытия OllyDbg в ОС Windows 9*x*.

Re_Pair_0.1. Еще один плагин, скрывающий заголовки окон OllyDbg.

UnhandledExceptionFlter. Скрывает OllyDbg от обнаружения.

ollydbg_hardware_breakpoint. Защищает от сброса HW-брекпойнтов.

ntglobalflag. Скрывает OllyDbg.

Скриптовые плагины

Это плагины, которые добавляют в отладчик OllyDbg разнообразные языки, позволяющие писать сценарии (скрипты), автоматизирующие процесс реверсинга.

OllyMashine. Виртуальная машина для интерпретации asm-подобного языка.

OllyPerl. Виртуальная машина для выполнения Perl-скриптов.

Дамперы

К этому классу относятся плагины, позволяющие делать дамп программ. Зачем это нужно, я собираюсь рассказать в следующей книге.

peDumper303. Прекрасный дампер, разработанный нашим соотечественником FKMA.

Ollydump300110. Отличный дампер процессов.

CLBPlus1.0. Плагин снимает дамп памяти размером *Y*, начиная с адреса *X*.

Другие полезные плагины

ApiBreak. Плагин полезен в основном для новичков, которые пока что плохо осознают, когда и какие брекпойнты надо устанавливать. Все функции отсортированы по категориям.

Asm2Clipboard. Превосходный плагин, позволяющий копировать asm-код.

Копирование OllyDbg:

```
00401012   > A1 1BE14A00    mov EAX,[DWORD DS:4AE11B]
00401017   . C1E0 02        shl EAX,2
```

Копирование с помощью плагина:

- с опцией `Copy fixed asm code`:

```
mov EAX,[DWORD DS:4AE11B]
shl EAX,2
```

- с опцией `Copy asm code`:

```
/*401012*/  mov EAX,[DWORD DS:4AE11B]
/*401017*/  shl EAX,2
```

Attachanyway. Позволяет "прицепиться" (подсоединиться) к запущенному процессу, даже если установлен хук (перехват) на `NtContinue`. В комплект поставки плагина входит тестовый файл — запустите его с активацией плагина и без и ощутите разницу.

BASE64. Позволяет декодировать BASE64[1]; поставляется вместе с исходными текстами.

BP_OLLY. Добавляет возможность удобного набора API; аналог плагина ApiBreak.

[1] Алгоритм шифрования.

Cleanupex112. Позволяет удалять временные файлы и лог-файлы, создаваемые отладчиком (в процессе дизассемблирования отладчик создает файлы иногда весьма внушительного размера, которые после исследования могут оказаться бесполезными).

CmdBar 310109c. Настоятельно рекомендую данный плагин всем. Любители SoftICE сразу оценят этот плагин. Он позволяет устанавливать брекпойнты, производить вычисления, просматривать дамп памяти и т. п.

Пример:

```
bpx MessageBoxA
```

С помощью этой инструкции мы устанавливаем брекпойнт на все очевидные вызовы API из программы.

Имя функции нужно вводить в соответствии с регистром букв. Если вы запишете `bpx messageboxa` или `bpx MessageBoxa`, то брекпойнт не будет установлен. Например, есть код:

```
call MessageBoxA
```

На этот код отладчик установит брекпойнт. А вот такой код

```
mov  EAX, Addr MessageBoxA
call EAX
```

отладчик пропустит. Поэтому я рекомендую использовать глобальный брекпойнт

```
bp MessageBoxA
```

В данном случае брекпойнт устанавливается в системной библиотеке USER32.DLL, и любой вызов будет перехвачен отладчиком.

Учтите, что при срабатывании брекпойнта `bp` или `bpx` в точке останова значение байта подменяется на 0CCh (`int 3`), и многие защиты проверяют наличие отладчика, например, так:

```
mov EAX, addr MessageBoxA
cmp byte [EAX], 0CCh
```

Если сравниваемые значения равны — мы в отладчике, поэтому на особо злых защитах рекомендую использовать HW-брекпойнт на исполнение:

```
he RegOpenKeyExA
```

Установка данного брекпойнта не меняет код в памяти, и защиты, проверяющие первый (*n*-й) байт на равенство 0CCh, не сработают.

Также удобно пользоваться калькулятором. Для этого ставим знак вопроса и выражение, которое требуется вычислить; например:

```
? EAX
```

покажет нам десятичное `DEC: 10`, шестнадцатеричное `HEX: 0Ah` и ASCII.

Еще пример:

```
? EAX+EAX*EDI+12
```

Примечание

Учтите, что вычисления производятся с целыми числами, поэтому если вы разделите 28 на 10, то получится не 2,8, а 2. Это нужно запомнить.

Также частенько приходится использовать

```
d addr
```

для отображения интересующих данных в окне дампа. Еще примеры:

```
d EAX
d 0AFDDDD11
```

Примечание

Обратите внимание: если адрес или значение начинается с буквы, то необходимо ставить перед числом 0 (естественно, это не относится к регистрам).

Также можно задавать условия для уменьшения ложных срабатываний:

```
BP 77DA3C77, EAX=="PASS"
```

В данном случае отладчик остановится, только если регистр EAX будет указывать на строку `"PASS"`.

Можно устанавливать брекпойнты на участи памяти, например:

- `mr` — чтение памяти;
- `mw` — запись в память;
- `md` — удалить брекпойнт;
- `mr EDX`.

И последняя полезная команда — `tc` (идти по коду, пока не выполнится условие):

```
tc EIP==77DA3C77
```

Cmdline. Аналог плагина CmdBar 310109c.

CodeRipper1_beta4. Позволяет дизассемблировать код в формате asm, Delphi и C++.

DebugActiveProcessStop. Предназначен для остановки отладки какого-либо процесса в пределах отладочной сессии, т. е. "отпустить на волю" программу, которую вы отлаживаете.

Dejunk_v0.13. Превосходный плагин от китайских друзей. Служит для отчистки "разбавленного" кода в различных протекторах и защитах. С помощью этого плагина изучать "замусоренный" код гораздо приятнее. Например, при изучении мусорного кода AsProtecta мы, применив данный плагин, из 10 команд получаем 2, что сильно облегчает отладку.

ExtraCopy0.9. Прекрасный плагин, позволяющий копировать куски кода, пересчитывая смещения:

```
00455B0E  |. 803D 7C104600 >CMP [BYTE DS:46107C],0
00455B15  |. 75 25          JNZ SHORT CrakMe-6.00455B3C
00455B17  |. 803D 7D104600 >CMP [BYTE DS:46107D],0
00455B1E  |. 74 14          JE SHORT CrakMe-6.00455B34
```

Результат работы данного плагина (обратите внимание на адреса перехода):

```
00455B29     803D 7C104600 >CMP [BYTE DS:46107C],0
00455B30     75 0A          JNZ SHORT CrakMe-6.00455B3C
00455B32     803D 7D104600 >CMP [BYTE DS:46107D],0
00455B39    ^74 F9          JE SHORT CrakMe-6.00455B34
```

вот так копирует OllyDbg:

```
00455B29     803D 7C104600 >CMP [BYTE DS:46107C],0
00455B30     75 25          JNZ SHORT CrakMe-6.00455B57
00455B32     803D 7D104600 >CMP [BYTE DS:46107D],0
00455B39     74 14          JE SHORT CrakMe-6.00455B4F
```

Fader2. Если хотите, чтобы OllyDbg был прозрачным, — пожалуйста. Правда в отладчике могут появиться глюки.

GoDup1_2. Лучший плагин! Настоятельно рекомендую иметь его всем! Самая крутая вещь из всех, что я видел, — умеет применять сигнатуры от IDA PRO, MAP-файлы, созданные в IDA PRO, выдает информацию по процессам, просматривает ресурсы, Блокнот.

Для корректной работы нужно прописать путь к файлу DUMPSIG.EXE (рис. 4.30) и к каталогу сигнатур IDA (рис. 4.31).

До применения сигнатур:

```
0045C047  |. 51 PUSH ECX                          ; |Arg1
0045C048  |. E8>CALL CrakMe-6.00459730             ; \CrakMe-6.00459730
0045C04D  |. 83>ADD ESP,8
0045C050  |. 8B>MOV EAX,[DWORD DS:ESI+28]
0045C053  |. 50 PUSH EAX                          ; /Arg1
0045C054  |. E8>CALL CrakMe-6.004575F0             ; \CrakMe-6.004575F0
```

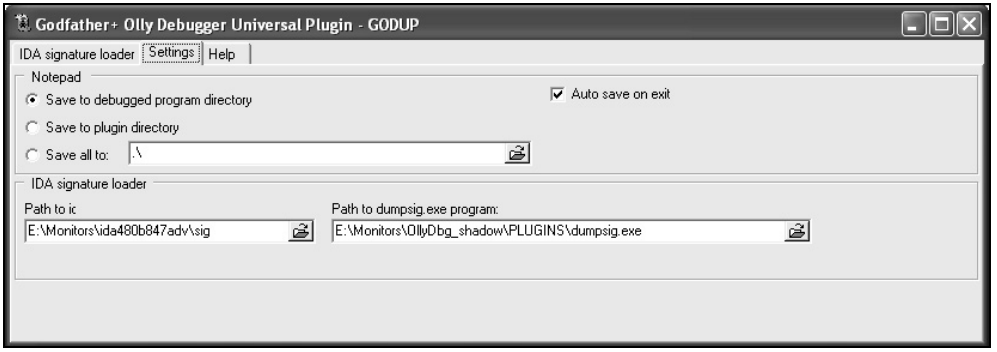

Рис. 4.30. Настройки плагина

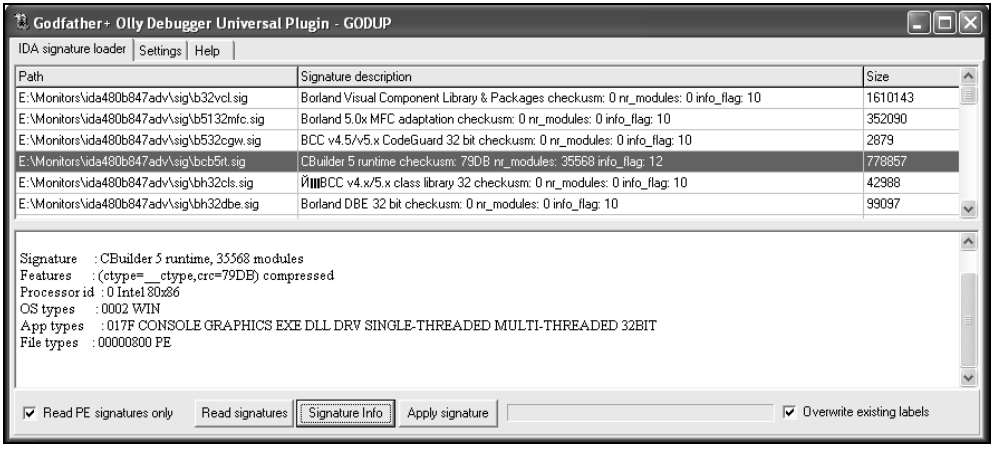

Рис. 4.31. Загрузка сигнатур

После применения сигнатур:

```
0045C047   |. 51 PUSH ECX                          ; |Arg1
0045C048   |. E8>CALL <CrakMe-6.__initmatherr>      ; \CrakMe-6.00459730
0045C04D   |. 83>ADD ESP,8
0045C050   |. 8B>MOV EAX,[DWORD DS:ESI+28]
0045C053   |. 50 PUSH EAX                          ; /Arg1
0045C054   |. E8>CALL <CrakMe-6.__initfmode 0010:_>; \CrakMe-6.004575F0
```

Как видите, читабельность листинга намного улучшилась. Мы будем применять данный плагин в исследованиях.

j10n140106. Меняет шрифт строки состояния в главном окне отладчика.

Labelmaster. Позволяет сохранять/загружать пользовательские комментарии.

MapConv_14. Аналог плагина GoDup1_2, только хуже. Позволяет импортировать MAP-файлы из SoftICE или IDA.

mYoLLYpLUG. Предназначен для разработки плагинов.

olly_bp_man. Очень хорошая штука. Бывает, что при изучении программы происходит порча UDD-файла, и тогда все ваши брекпойнты теряются. Хорошо, если вы их записали, иначе придется изучать все с начала. Данный плагин позволяет экспортировать/импортировать ваши наработки.

olly2html. Perl-скрипт для перевода необходимых кусков asm-кода, скопированного из OllyDbg, в формат HTML.

OllyAdvanced. Очень полезная вещь:

- исправляет NumOfRvaAndSizes — многие протекторы в последнее время записывают в это поле значение, не равное 10, и OllyDbg останавливается в системных функциях; многих это сбивает с толку;
- добавляет опции в меню плагинов;
- защищает от метода обнаружения отладчика (output debug-strings);
- добавляет опцию, предписывающую не выводить сообщение "Entry point alert" ("Ошибка точки входа");
- добавляет опцию, предписывающую не выводить сообщение "Too many patches" ("Слишком много изменений").

И еще несколько маленьких дополнений.

OllyFlow, ollygraph. Использует возможности IDA\WINGRAPH32.EXE. Попробуйте, интересная штучка.

OllyPad. Позволяет делать заметки.

OllyUni. Поддерживает расширенную работу с Unicode.

PluginLoader. Отличная вещь. Позволяет настроить для часто используемых плагинов горячие клавиши.

PuntosMagicos. Очень полезный плагин для эмуляции брекпойнта `hmemcpy` для NT-платформ (сейчас это называется `H-POINT`). Необходим, когда в программе выполняется копирование одного фрагмента памяти (например, регистрационного кода) в другой stayontop-1_0.

Позволяет выбрать окно, которое в OllyDbg будет поверх всех остальных.

Просто незаменимый плагин. Позволяет в Windows 2K/XP/2003 установить брекпойнт, аналогичный `hmemcpy` для Windows 9*x*, если вам нужно отследить ввод и сравнение строк/серийных номеров (паролей) и т. п.

Sleepp. Снижает загрузку процессора отладчиком OllyDbg.

Snd-dataripper1.2. Суперплагин. Позволяет извлекать массивы байт для использования в своих программах. Поддерживает asm, Delphi, C++, CSV, String.

TableExporter. Экспортирует из окна дампа массив байт в формате C и Delphi:

```
const
  unsigned char Values[32] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x02, 0x8D, 0x40, 0x00}
```

TBD_DebugPlugin. Попробую написать попроще. Это плагин для отладчика, помогающий отлаживать плагины для этого отладчика...

vb_plugin_for_olly. Пример плагина на Visual Basic.

WindowInfos. Позволяет получить дескриптор выбранного окна.

WindowJuggler_v0.06. Аналог программы MsSpy. Позволяет включать заблокированные кнопки, изменять размеры окон и т. п.

Ustrref. Ищет все строки ASCII или Unicode. Поставляется вместе с исходными текстами.

Вот вы и познакомились с прекрасным отладчиком OllyDbg. Надеюсь, вы многое узнали. Теперь мы переходим к практической части реверсинга.

Глава 5

Как именно взламывают программы

В данной главе мы рассмотрим на примерах, как именно взламываются программы, типовые защиты программ, а также инструменты, применяемые крекерами.

Первые шаги в реверсинге или "С чего начать?"

Это самый главный вопрос, какой задают все начинающие крекеры. Начав писать эту книгу для того, чтобы помочь им в этом нелегком деле, я не претендую на то, что предлагаемый мной путь к достижению высокого мастерства крекера — единственный правильный. Таких путей множество, и каким пойти — каждый решает для себя сам. Мой путь — один из возможных. Но именно этот путь (или, по-другому, система) помог мне эффективно проанализировать защиту многих программ. Как сказал CyberManiac: "Главным критерием эффективности того или иного пути может служить лишь то, достиг или нет крекер конечного результата." Вот с этим трудно не согласиться. Это основа всего крекерства. Поэтому я хочу вам сказать: забудьте любые утверждения о "некрасивости", "неспортивности" тех или иных приемов борьбы с защитами и ориентируйтесь лишь на достижение конечной цели.

Основные проблемы начинающих крекеров возникают от того, что авторы многочисленных пособий уровня "для попугаев" не объясняют читателю, почему они поступают именно так, а не иначе. В результате начинающий крекер спокойно ломает новые версии одной и той же программы, но теряется перед подобной защитой, реализованной с небольшими отличиями. В этой книге я расскажу вам о способах защиты и о том, почему надо делать именно так. Книга ни в коей мере не претендует на полноту описания всех защит

программного обеспечения, но я постарался дать в ней необходимые основы, применяя которые вы сможете анализировать программы.

В крекинге есть два пути. Первый путь — анализировать и изучать то, как работает программа. Этот путь весьма надежен, но для получения гарантированного результата он требует много времени, усилий и практического опыта. Так как вы только начали обучаться крекингу, у вас просто нет нужных знаний и опыта, чтобы понять, в каком направлении двигаться. Получается парадокс: чтобы приобрести опыт, нужно ломать программы, причем ломать успешно (и чем больше — тем лучше), а ломать их не получается из-за недостатка опыта. Поэтому есть второй путь.

Второй путь — исследовать программу, наблюдая за производимыми ею внешними эффектами. То есть вы не сразу начинаете выяснять, что и как происходит в недрах программного кода, а сначала строите предположения "что это может быть такое", "как это может быть сделано" и "как бы я добился такого эффекта, будь я программистом". Успехов на этом пути вам помогут добиться опыт программиста и богатая фантазия.

В любом случае вам необходимо знать команды ассемблера, способы передачи параметров в функции и процедуры, назначения API-вызовов ОС (в этом вам может помочь моя книгу "Assembler: Экспресс-курс"). Также вам понадобится знание языков программирования высокого уровня. Чем больше вы знаете, тем легче и быстрее вы будете исследовать защиты программ.

Но строить предположения о работе программы на пустом месте невозможно, значит, вам нужны стартовые знания о программе. Поэтому собирайте как можно больше информации о самой программе — узнайте, упакована ли она, зашифрована ли, какие ограничения содержатся в незарегистрированной программе и как выглядит процесс регистрации. Выясните, что произойдет, если вы попытаетесь использовать программу дольше, чем это предусмотрено ограничениями. Посмотрите, какие текстовые строки и ресурсы содержатся внутри программы. Поинтересуйтесь, к каким файлам и ключам реестра программа обращается при запуске, и т. д. Не поленитесь заглянуть в справочную систему программы — там вы можете найти описание различий между зарегистрированной и незарегистрированной версиями. Значительная часть этой информации вам, скорее всего, не пригодится, но лишней информации не бывает. Поэтому — собирайте информацию!

Как это работает на практике? Представьте себе программу, которая что-то делает. Например, отказывается запускаться после 30 дней использования, выводя стандартное окошко с сообщением (как создается такое окно, знают все программисты — это стандартная функция `MessageBox`). Итак, у нас есть первое наблюдение: программа создает окно. Если перевод системного вре-

мени на компьютере не помогает обмануть программу и заставить ее работать дольше, чем положено, — это второе наблюдение. Из него следует, что программа проверяет текущую дату не на основе показаний внутренних часов Windows. Предполагаем, что скорее всего программа либо уже сделала пометку "больше не запускаться" где-нибудь в реестре или на диске, либо все-таки определяет текущее время, но каким-либо хитрым способом. Например, читая дату последнего доступа или дату модификации какого-либо файла. Если программа не просто "задумывается" при запуске, но еще и шуршит винчестером, вероятность второго варианта сильно повышается. Теперь начинаем проверять эти варианты. В первом случае нам однозначно проще докопаться до истины, установив точки прерывания на все вызовы `MessageBox` в программе, и выяснять, какой из условных переходов позволяет избежать появления этого сообщения. Во втором случае в качестве отправной точки можно использовать всевозможные функции `GetFileTime`, `CompareFileTime`, `FindFirstFile`, `FindNextFile` и т. д.

У большинства защит, как бы аккуратно они ни были реализованы, все-таки есть слабое место, которое может быть спрятано глубоко в недрах программы, "раскидано" по множеству процедур или же быть совершенно незаметным — но оно есть. Главное его найти.

Если мы знаем, что нам нужно искать, остается только определить, как выглядит это слабое место. Наиболее "легки" и "удобны" для крекера, прежде всего, глобальные переменные. В такой переменной может храниться, например, состояние программы (зарегистрирована/не зарегистрирована), число запусков (или количество дней) до истечения демонстрационного периода или результат проверки серийного номера на правильность (правильный/не правильный) и т. д. Как-то мне попалась программа с глобальной переменной, значение которой по-умолчанию было равно нулю и менялось на единицу, если серийный номер, извлекаемый из реестра, был верен (помнишь, nice?). Несмотря на то, что сам алгоритм проверки был очень хитрым, исправление одного-единственного бита превратило программу в зарегистрированную.

Другой встречавшийся мне прием, эффективный, в основном, при ограничении максимального или минимального значения какого-либо числового параметра (это может быть, например, количество создаваемых документов, число запусков и т. п.) — поиск константы, с которой производится сравнение, и модификация либо самой константы, либо условия проверки. Таким образом, заменив значение константы, мы можем пользоваться программой, несмотря на то, что она не зарегистрирована.

Другая проблема для реализующих защиту программистов, сильно облегчающая жизнь крекерам, — это "проблема условного перехода", которая

состоит в том, что реализовать проверку какого-либо условия без применения команды условного перехода довольно сложно. Дело тут в том, что любой такой переход очень просто превратить в такой же, но с противоположным условием, — обычно для этого достаточно исправить ровно один бит! Но хотя это и простой путь к преодолению защиты, реализовать его бывает достаточно затруднительно. Причина в том, что условных переходов, имеющих отношение к защите, в программе может быть довольно много (представьте себе 100—200 таких переходов, разбросанных по всей программе!); их поиск требует особой внимательности. Достаточно пропустить хоть один и программа будет работать неправильно.

Теперь вам известны наиболее часто встречающиеся слабые места программ, в выявлении которых заключена половина успеха крекера. Но в этом деле вас ожидают и трудности. Прежде всего, это проблема неверной интерпретации собранной информации. Как говорится "легче всего спрятать на видном месте". Что это значит? Лучше я поясню это на примере.

Допустим, есть программа, которая поддерживает использование плагинов и при запуске сканирует папку с плагинами на наличие файлов с расширением PLG. Вы, вполне логично, предполагаете, что программа строит список плагинов для дальнейшей загрузки и подключения. А потом вы можете очень долго искать механизм определения даты первого запуска — и не найти его. Почему? А потому, что он уже отработал. Когда? Как раз при поиске плагинов. Как такое может быть? Да очень просто. Делается это следующим образом: в комплект поставки программы включается как минимум один плагин. Далее программа проверяет дату последнего обращения к файлу этого плагина. Вот так автор защиты может спрятать свою защиту "на самом видном месте".

Инструменты — главные орудия крекера

Без инструментов в крекинге никуда не денешься. В то же время сами по себе инструменты — не более чем "металлолом", бесполезный набор программ. Однако для использования этих программ, как правило, требуются соответствующие знания. Насколько будет полезен отладчик, если вы не знаете, что означают те хитрые цифры и буквы, которые он показывает на экране? Поэтому помните: прежде чем задать вопрос, вы должны быть уверены, что сможете понять ответ. Но, тем не менее, от качества этих инструментов будут во многом зависеть скорость и эффективность ваших действий. Ваш инструментарий должен постоянно обновляться, вы должны быть в курсе всех новых возможностей этих инструментов. В этом может помочь такой известный сайт, как **cracklab.ru**. Там есть множество инструментов, которые помогут вам в нелегком труде.

Но есть и проблема: инструментов для крекера очень много, и чтобы не потонуть в этом изобилии, а выбрать лучшие из лучших, вам надо для начала узнать, какие они вообще бывают. И превозносить тут одни инструменты перед другими бесполезно, т. к. каждый выбирает инструменты для себя сам. Я могу только предложить следующий подход: возьмите инструменты, которые я рекомендую в начале книги, попользуйтесь ими некоторое время, посмотрите, чем они вас не устраивают или наоборот, может быть, вы решите, что это самое лучшее для вас. Получив определенное представление о своих предпочтениях в крекинге, вы сможете найти инструменты себе по душе.

Конечно, найдутся и те, кто возразит, что широкое использование готовых утилит якобы мешает самостоятельному мышлению, чрезмерно упрощает процесс взлома и т. п. Так вот. Помните, что я вам говорил? Главное — это достичь результата, и если вам надо для этого использовать готовые утилиты — используйте без колебания. По мне так хоть тузы в рукавах, хоть электромагниты в ботинках, главное — получить результат. Поэтому если вам вдруг покажется, что надо задействовать иголку с ниткой, ручку и отвертку — вперед! Используйте все, что только сочтете нужными, если это поможет вам добиться желаемого результата. Конечно, рано или поздно вы перейдете от использования чужих программ к написанию собственных утилит, но такой переход должен быть продиктован только насущной необходимостью. В конце концов, большинство программ написано именно для того, чтобы люди ими пользовались.

Вот совершенно необходимые нам инструменты.

- *Дизассемблеры и отладчики.* Традиционно оба эти инструмента используются в паре, поскольку дизассемблер выдает лишь "чистый код". Хотя современные дизассемблеры способны также распознать вызовы стандартных функций, выделить локальные переменные и т. д. Работая с дизассемблером, вы можете лишь догадываться о том, какие данные получает та или иная функция в качестве параметров и что они означают; чтобы выяснить это, чаще всего требуется изучение если не всей программы, то довольно значительной ее части. Отладчики выполняют другие функции — они позволяют анализировать код в процессе его работы, отслеживать и изменять состояние регистров и стека, править код "на лету". Обратной стороной медали является "неинтеллектуальность" многих отладчиков. Например, SoftICE — "лучший отладчик всех времен и народов" — ничего не знает о типах данных. К счастью, сейчас появились отладчики, сочетающие высокое качество дизассемблирования и анализа кода с широкими возможностями по его отладке (например, OllyDbg).
- *Декомпиляторы и специализированные отладчики.* С ростом мощности ЭВМ довольно широкое распространение получили компиляторы, соз-

дающие не "чистый" машинный код, а некий набор условных инструкций, выполняемый при помощи интерпретатора. Интерпретатор может как поставляться отдельно (Java), так и быть "прикрепленным" к самой программе.

Примечание

Формально не интерпретатор прикрепляется к программе, а программа к интерпретатору. Примером может служить Visual Basic при компиляции в p-code.

Для анализа таких программ используются специализированные утилиты, переводящие код, понятный лишь интерпретатору, в форму, более удобную для понимания человеком. Также некоторые декомпиляторы могут извлекать информацию об элементах интерфейса, созданных визуальными средствами (например, DeDe).

- *Распаковщики и утилиты для снятия дампа процессов.* Дизассемблировать запакованную или зашифрованную программу невозможно, но если очень хочется получить хоть какой-то листинг, можно попытаться извлечь из памяти компьютера "снимок" (дамп) программы в момент ее работы. Этот дамп уже можно более или менее успешно дизассемблировать. Более того, на основе дампа можно воссоздать исполняемый файл программы, причем этот файл будет успешно загружаться, запускаться и работать. Именно на этом принципе и основана работа большинства современных распаковщиков: подопытная программа запускается под управлением распаковщика; распаковщик ждет некоторого события, говорящего о том, что программа полностью распаковалась, тут же "замораживает" программу и сбрасывает ее дамп на диск. Защитные системы нередко пытаются противодействовать получению работоспособного дампа при помощи манипуляций с сегментами и таблицами импорта/экспорта. В этих случаях приходится применять PE-реконструкторы, т. е. утилиты, обнаруживающие в дампе некорректные ссылки на функции и пытающиеся их восстановить. Многие продвинутые дамперы и распаковщики оснащены встроенными средствами восстановления секций импорта.
- *Утилиты анализа файлов.* Очень часто требуется быстро узнать, каким упаковщиком или защитным ПО обработана та или иная программа, найти все текстовые строки в дампе памяти, просмотреть содержимое файла в виде текстовых строк, вывести в файл список импортируемых и экспортируемых программой функций и многое другое. Для всех этих целей есть огромное количество специализированных утилит (например, PEiD), позволяющих быстро проанализировать файл на наличие тех или иных признаков. Эти утилиты, как правило, не являются жизненно необходимыми, но, при надлежащем качестве, способны сэкономить огромное количество времени и сил.

- *Шестнадцатеричные редакторы и редакторы ресурсов.* Шестнадцатеричные редакторы — это, несомненно, наиболее древние из программистских инструментов. Они наиболее часто используются крекерами. Сейчас лишь крекеры занимаются тем, что исправляют байты в готовых программах. Редакторы ресурсов, в принципе, занимаются тем же самым, но в отношении присоединенных к исполняемому файлу ресурсов. Именно при помощи редакторов ресурсов выполняется значительная часть работ по "независимой" русификации программ и доработке интерфейсов.
- *Утилиты мониторинга.* Очень часто требуется узнать, какие именно действия выполняет та или иная программа, откуда читает и куда записывает данные, какие стандартные функции и с какими параметрами она вызывает. Получить эти знания как раз и помогают утилиты мониторинга. Такие утилиты делятся на две большие группы:
  - утилиты, которые отслеживают сам факт возникновения каких-либо событий;
  - утилиты, которые позволяют выявить один или несколько специфических типов изменений, произошедших в системе за некий промежуток времени.

  К первой группе относятся всевозможные API-шпионы, перехватывающие вызовы системных (а более продвинутые — и не только системных) функций, хорошо известные утилиты RegMon, FileMon, PortMon и т. д., перехватчики системных сообщений и многие другие. Эти утилиты, как правило, предоставляют весьма подробную информацию об отслеживаемых событиях, но генерируют весьма объемные и неудобные для анализа лог-файлы, если отслеживаемые события происходят достаточно часто.

  Вторая группа представлена всевозможными программами, создающими снимки реестра, жесткого диска, системных файлов и т. п. Эти программы позволяют сравнить состояние компьютера "до" и "после" некоего события, построить список различий между состояниями и на основе этого списка сделать определенные выводы. Основная проблема при работе с такими утилитами хорошо описывается словами "«после»" — не значит «вследствие»", т. е. кроме интересующей вас деятельности программы вы можете получить лог "жизнедеятельности" других параллельно работающих программ и операционной системы.
- *Прочие утилиты.* Есть огромное количество утилит, не вписывающихся в приведенные категории или попадающих в несколько категорий. Крекинг — занятие весьма многогранное, и столь же многогранны инструменты, в нем используемые. Более того, некоторые из утилит, которые могут быть полезны для крекера, создавались для совершенно иных целей

(хорошим примером может служить программа ArtMoney, которая вообще-то была предназначена для того, чтобы продлевать жизнь в компьютерных играх). Поэтому еще раз обращу ваше внимание: важно не только качество инструмента, но и умение его применить.

Следующий шаг при анализе программы — это установка самой программы на компьютер крекера. Тут надо сказать вот о чем. Допустим, у нас есть программа с защитой, которую нужно обезвредить. Инсталляционный файл программы представляет собой инсталляционный набор файлов. Однако не спешите запускать инсталляцию программы. Дело в том, что многие программы в настоящее время поставляются в виде инсталляционного пакета. Для установки программы, как правило, требуется либо запустить один из файлов пакета (это всем известные SETUP.EXE), либо открыть файл при помощи другой заранее установленной программы (например, файлы с расширением msi, созданные инсталлятором Microsoft).

Инсталляционные пакеты кроме самих файлов, которые требуется установить на машину пользователя, содержат описание сценария инсталляции в том или ином виде. Инсталляционный сценарий описывает режим, в котором устанавливается тот или иной файл (добавление, замена, замена с предварительной проверкой версии и т. д.), данные, которые необходимо внести в реестр или файлы конфигурации, порядок запуска программ (до, в процессе и после инсталляции), а также многое другое. Из всего сказанного понятно, что сценарий может делать и такие интересные вещи, как проверка серийных номеров, создание записей в реестре, а также распаковка и запуск программ.

Ну, как? Догадались об опасности создания ключей в реестре или запуска каких-либо программ? Нет? Тогда поясню на примере.

Если вы взламываете какую-либо программу, оснащенную ограничением на время использования или на число запусков, то счетчик может находиться именно в инсталляционном сценарии. Представьте себе такую ситуацию: программа хранит дату первого запуска или какую-либо другую информацию, необходимую для проверки на истечение срока пробного использования, в реестре. Разумеется, информация закодирована, предприняты меры, чтобы защиту не могло обмануть изменение системной даты, возможно даже соответствующий ключ реестра хорошо замаскирован. Но, тем не менее, существует одна лазейка, а именно если триальный ключ в реестре отсутствует, защита считает, что раньше программа не запускалась. Поэтому защиту можно обмануть, просто удалив из реестра лишние ключики.

А теперь представьте, что триальный ключ создается в процессе инсталляции инсталляционным сценарием, а программа при старте проверяет его наличие,

и если он отсутствует, программа не запускается вообще. Что тогда? А тогда вы долго будете искать, почему это программа не работает. Вам потребуются весьма значительные усилия, чтобы отыскать этот маленький, но зловредный ключик в огромном реестре еще более огромной ОС Windows. Вам нравится такая перспектива? Если встроенных средств инсталлятора оказывается недостаточно для создания триальной "метки", это можно реализовать при помощи небольшого исполняемого файла, который распаковывается в процессе инсталляции, запускается, делает свое черное дело и сразу после этого удаляется. Упрощенный вариант этого приема может выглядеть как автоматический запуск приложения после окончания инсталляции, чтобы защита смогла создать триальные "метки" на компьютере пользователя.

Какие средства мы можем применить, чтобы обнаружить и обезвредить эти и другие подобные приемы? Наиболее радикальным средством, разумеется, является декомпиляция инсталляционного сценария — в этом случае мы получаем практически полную информацию о том, что происходит в процессе установки программы, а в некоторых случаях даже можем повлиять на этот процесс, внеся исправления в инсталлятор. Но этот путь — не самый простой. Он может оказаться просто очень трудным в реализации, если инсталляционную программу программист писал сам. К счастью, на практике пользуются другим способом, позволяющим обнаружить произошедшие изменения. Этот прием заключается в использовании утилит мониторинга, делающих "снимки" системы (реестра, размеров и дат создания и модификации файлов) до и после установки и затем анализирующих различия между снимками. Подробный журнал изменений, выдаваемый такими программами, позволяет легко обнаружить подозрительные ключи и файлы, появившиеся в процессе инсталляции. Добавлю, что такие же "снимки" рекомендуется делать и при прохождении других критических периодов работы программы — при первом запуске, при последнем запуске перед окончанием триального срока, при первом запуске по истечении испытательного срока. Установить факт запуска каких-либо программ во время инсталляции можно при помощи утилит, отслеживающих создание и завершение процессов.

Однако созданием триальных ключей функции программ, запускаемых в процессе инсталляции, не ограничиваются. Дело в том, что набор функций, поддерживаемых инсталляторами, обычно довольно невелик, и некоторые действия (например, проверку серийного номера с использованием достаточно сложного алгоритма) выполнить средствами инсталляционных сценариев просто невозможно. Один из приемов, применяемых в этом случае, — запуск исполняемого файла, который и выполняет все необходимые операции, а затем возвращает управление инсталлятору. В частности, есть защиты, в которых проверка серийного номера реализована именно так. В некоторых инсталляторах

для этих же целей предусмотрен интерфейс, позволяющий использовать плагины (плагины в виде динамически загружаемой библиотеки также упрятываются внутрь инсталлятора, в нужный момент распаковываются во временную папку и после использования удаляются). Такие исполняемые файлы и плагины, разумеется, невозможно модифицировать напрямую и чаще всего не удается извлечь из инсталлятора для дизассемблирования и изучения, т. к. они хранятся внутри инсталлятора в сжатом виде, а многие коммерческие инсталляторы несовместимы по формату с обычными архиваторами. Если вы хотите исследовать такой исполняемый файл, вам почти наверняка потребуется снять с него дамп, чтобы получить материал для загрузки в дизассемблер. Но это тема отдельной книги, или второго тома данной серии.

Типовая защита программ "Запрос пароля"

Наконец-то мы закончили с теорией и подошли к практической реализации снятия защит. Здесь я покажу, как именно крекеры снимают защиты с программ. Это поможет программистам лучше защитить свои программы.

Наибольшее распространение получили программы типа "запрос пароля" или "запрос ввода регистрационного номера". Они реализуются по-разному, но сводятся к одному алгоритму:

1. Программа запрашивает пароль у пользователя.
2. Что-то с ним делает, например, шифрует или получает хеш-функцию.
3. Сравнивает полученную на шаге 2 функцию или пароль и запрошенный пароль с шага 1.

Без шифрования сообщений

Простейший пример программы представлен в листинге 5.1.

Листинг 5.1. Запрос пароля

```
// Функция ввода пароля
Me_Pass=Password_is_In();
// Сравнение
if(Me_Pass == "Pas1234")
          MessageBox(0,"PASSWORD OK","Password",MB_OK);
else
          MessageBox(0,"password FALSE","Password",MB_OK);
```

Как видите, здесь нет ничего сложного. Для обхода такой защиты достаточно найти место сравнения паролей и либо просто переписать пароль, либо (что еще проще) изменить программу так, чтобы она всегда считала, что введен правильный пароль (такой взлом чаще называют бит-хаком). Давайте пока не будем останавливаться на бит-хаке. Об этом мы поговорим попозже.

Для начала экспериментов мы возьмем программу CrakMe-1. Эту программу, как и многие другие, вы найдете на моем сайте **www.abyss-soft.narod.ru** и на сопроводительном компакт-диске данной книги. Как я уже сказал, данная программа имеет очень простую реализацию защиты. По странному стечению обстоятельств такую защиту имеет большинство программ, основанных на запросе пароля. Для других защит, основанных на "запросе пароля", данная защита является как бы основой, фундаментом. Какую бы защиту ни придумал программист, как бы ни усложнял проверку пароля, в конечном итоге все сводится к этой простой проверке.

Итак, начинаем исследовать. Запускаем программу.

Мы видим окно (рис. 5.1) с текстовым полем, в котором надо ввести пароль. Давайте попробуем что-нибудь ввести и нажать кнопку **OK**. Я ввел цифры "12345" и нажал кнопку **OK**. В результате получено сообщение о том, что пароль неверный (рис. 5.2).

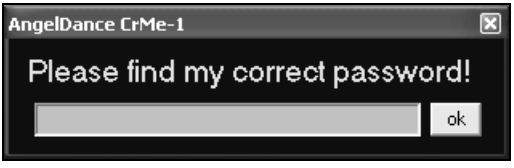

Рис. 5.1. Главное окно программы

Рис. 5.2. Окно сообщения о неверном пароле

Как я уже говорил, перед реверсингом программы мы должны узнать о программе как можно больше. Самое первое, что мы должны узнать, это (именно в таком порядке!):

1. Зашифрована программа или нет.
2. На каком языке она написана. Это очень важно, т. к. для того, чтобы поставить точки останова (брекпойнты) в отладчике, мы должны знать, на какие собственно функции их ставить. У каждого языка программирования они свои. Например, у Visual C++ или API они одни, а у Delphi или Builder C++ (компиляторов фирмы Borland) — совсем другие (я уже не говорю о Visual Basic!).

Давайте займемся нашим "узнаванием". Для реализации первого пункта запускаем утилиту PEiD (или, полностью, PE iDentifier v0.92 by snaker, Qwerton & Jibz).

При запуске этой утилиты появляется окно (рис. 5.3).

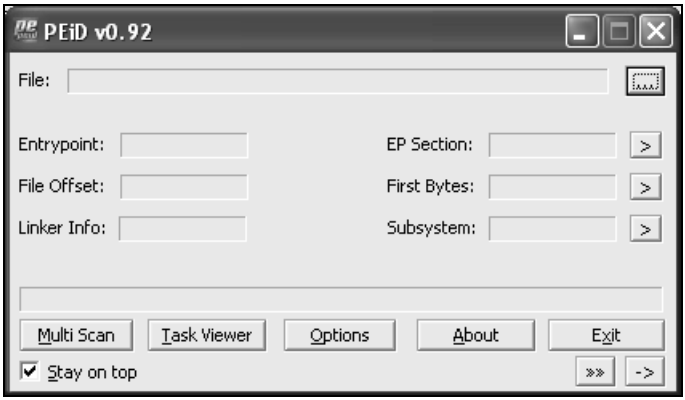

Рис. 5.3. Запуск PEiD

Нажимаем кнопку, расположенную справа от поля **File**, находим и открываем наш файл CRAKME-1.EXE.

Примечание

Можно просто перетащить наш файл CRAKME-1.EXE в окно PEiD с помощью мыши.

Программа PEiD проанализирует файл и выведет информацию о нем в своем окне (рис. 5.4).

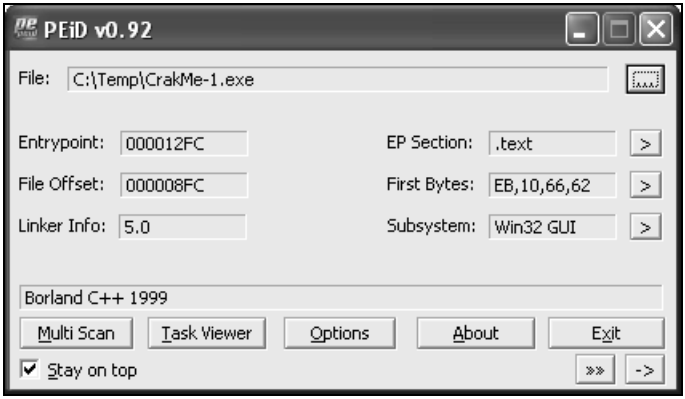

Рис. 5.4. Результат анализа исследуемого файла, выполненного программой PEiD

Как мы видим, в нижней строке нам написали "Borland C++ 1999". Это значит, что исследуемая программа написана на языке программирования C++ фирмы Borland и ничем не зашифрована и не запакована (имеются в виду шифровальщики, или протекторы, и паковщики сторонних разработчиков, такие как UPS, ASPPack и т. д.).

Примечание

Описание шифровальщиков (протекторов), паковщиков и их снятие с исполняемых файлов — тема отдельной книги. Скорее всего, это будет моя вторая книга по реверсингу.

Тут хочу сказать следующее. Никогда не полагайтесь только на одну программу-идентификатор протекторов/паковщиков. Это связано с тем, что есть множество способов обмана той или иной программы-идентификатора. Лучше используйте все, что у вас есть.

Итак, раз программа ничем не запакована, то мы можем смело запускать отладчик и загружать программу в него. Сами отладчики и работу с ними я уже описал. Поэтому здесь я по ходу дела буду давать только комментарии, необходимые для работы.

Я люблю отладчик OllyDBg (автор Oleh Yuschuk). У него, на мой взгляд, более удобный интерфейс, нежели у SoftICE. Впрочем, и SoftICE, и OllyDbg — очень хорошие отладчики, и тут каждый волен выбирать, что ему больше нравится.

В общем, запускаем OllyDBg и загружаем программу в него. Отладчик лучше всего развернуть на полный экран (рис. 5.5).

Как видим, OllyDBg остановился на самом начале файла, точнее, на точке входа в программу. Раз программа ничем не запакована, есть шанс, что программист не стал мучаться и делать защиту в программе. Очень странно, но большинство программистов именно так и поступает. Думаю, дело в том, что все свои силы программист отдает созданию программы, а чтобы придумать защиту, их у него уже не остается.

Первое, что может прийти в голову, это постараться найти в теле программы строки, которые она выводит на экран (например, "password FALSE"). Большинство программистов не шифрует выводимые данные в программе, а раз так, этим можно воспользоваться.

Поэтому давайте выполним поиск всех текстовых строк в файле. Для этого мы в главном окне (окне кода) нажимаем правую кнопку мыши и выбираем в контекстном меню команду **Search for > All referenced text strings**. Отладчик выдаст нам все найденные текстовые строки (рис. 5.6).

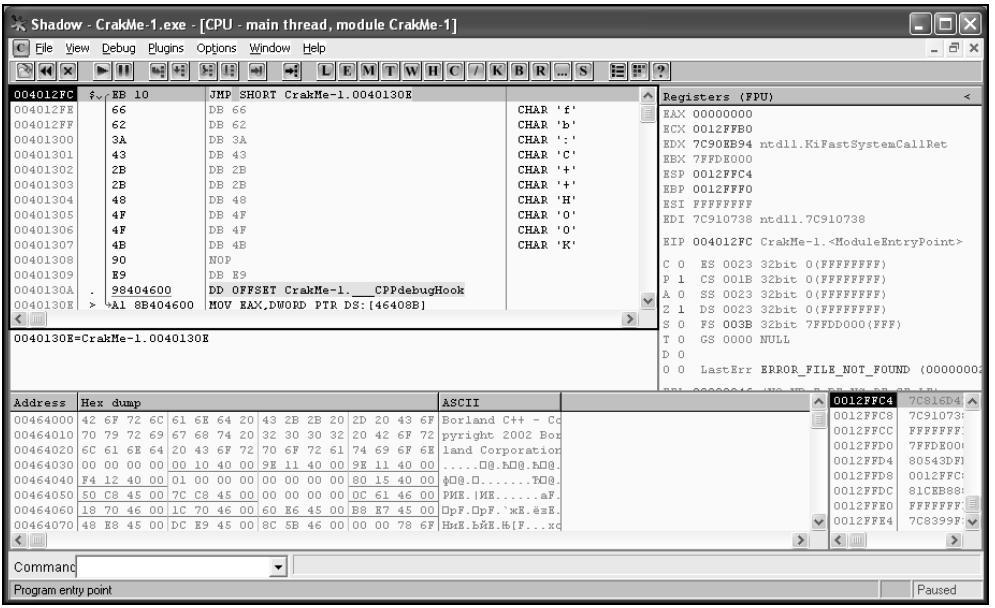

Рис. 5.5. Отладчик OllyDBg

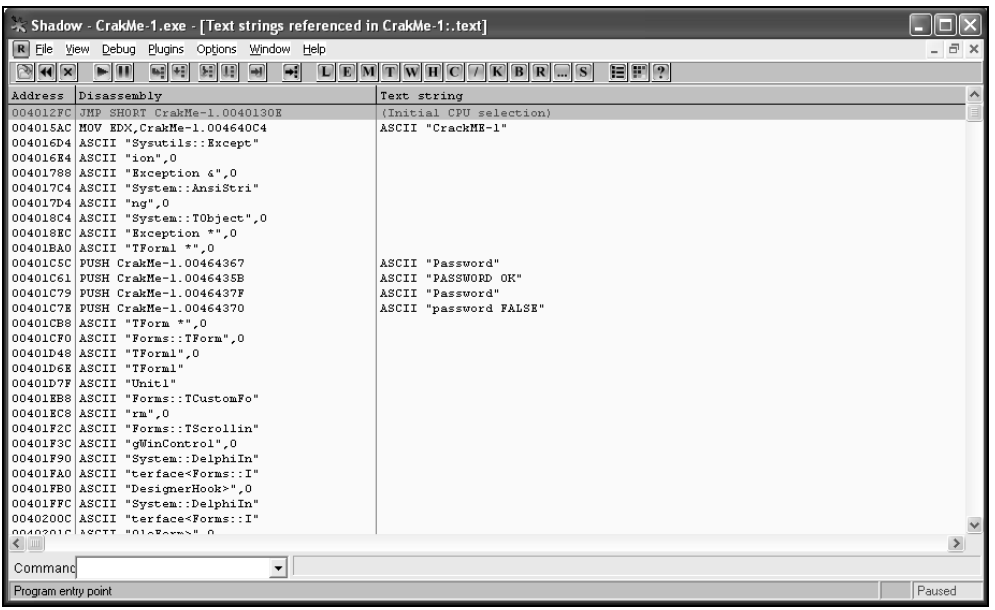

Рис. 5.6. Вывод всех текстовых строк в OllyDBg

Мы видим такие строки:

```
00401C5C    PUSH    CrakMe-1.00464367       ASCII   "Password"
00401C61    PUSH    CrakMe-1.0046435B       ASCII   "PASSWORD OK"
00401C79    PUSH    CrakMe-1.0046437F       ASCII   "Password"
00401C7E    PUSH    CrakMe-1.00464370       ASCII   "password FALSE"
```

Что значат эти строки? Давайте еще раз запустим программу CRAKME-1.EXE, введем в ее окне любые символы в качестве пароля и нажмем кнопку **OK**. Мы увидим окно, изображенное на рис. 5.2. Это же очень похоже на строчку:

```
00401C7E    PUSH    CrakMe-1.00464370       ASCII   "password FALSE"
```

Сообщение о неправильном пароле не зашифровано. Нам очень повезло — попался ленивый программист. Нажимаем мышкой на эту строчку в отладчике и попадаем на код, срабатывающий в том случае, если введенный пароль не верен (рис. 5.7).

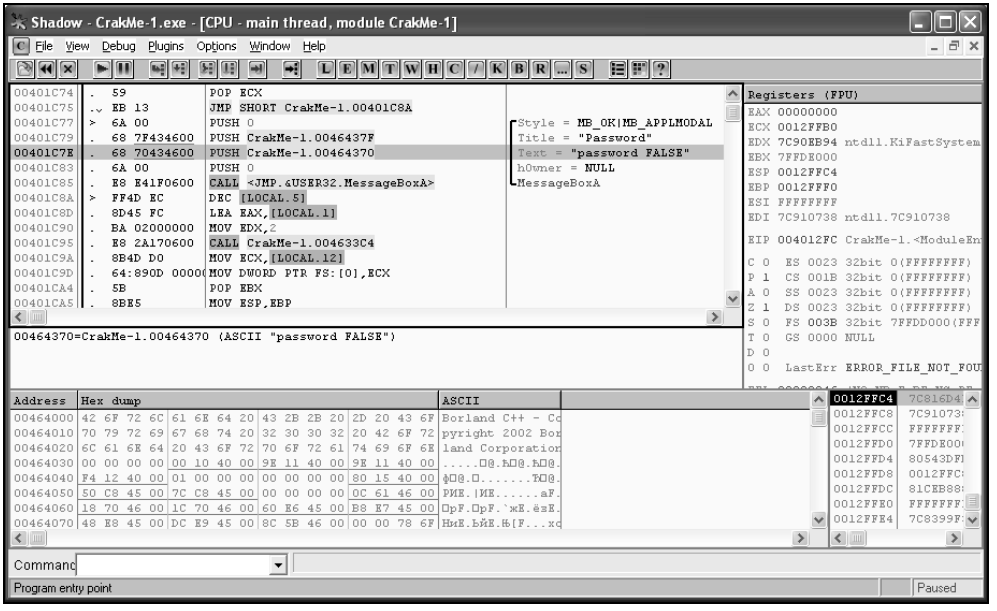

Рис. 5.7. Код вывода сообщения о неверном пароле

На рис. 5.7 мы видим вызов функции `MessageBoxA`. Помните, что я вам говорил об необходимости знать API-функции Windows? Вот тут-то они нам и пригождаются, т. к. на рис. 5.7 виден вызов стандартной функции.

Но нам неинтересно, что делается в программе после того, как нам сообщили о неверном пароле. Нам нужен сам пароль. Поэтому давайте пролистаем

содержимое главного окна отладчика немного вверх — до тех пор, пока не увидим сообщение о том, что введен правильный пароль (рис. 5.8).

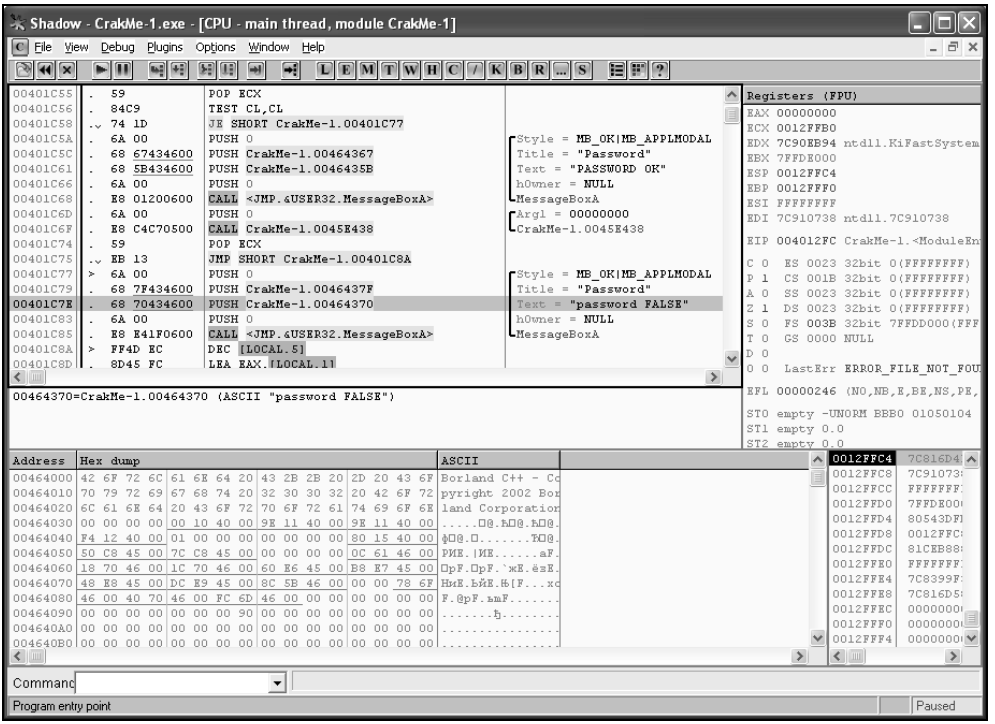

Рис. 5.8. Код вывода сообщения о верном пароле

Обратите внимание на стрелочки, которые находятся левее кода инструкции (рис. 5.9).

Эти стрелки показывают направление перехода. То есть откуда или куда происходит переход после срабатывания логики в программе. Если нажать мышкой на эту стрелочку (или просто выделить строку рядом со стрелкой), то мы наглядно увидим данный переход. Давайте выделим мышкой строку (рис. 5.10).

Мы видим, что переход на наше сообщение о неверном пароле осуществляется со строки

```
00401C58    74 1D       JE SHORT CrakMe-1.00401C77
```

Как вы знаете, в ассемблере функция `JE` осуществляет условный переход. Исходя из этого, сделаем вывод, что некая функция проверяет наш пароль на правильность и, если он верен, переходит на строку:

```
00401C5A    6A 00       PUSH 0
```

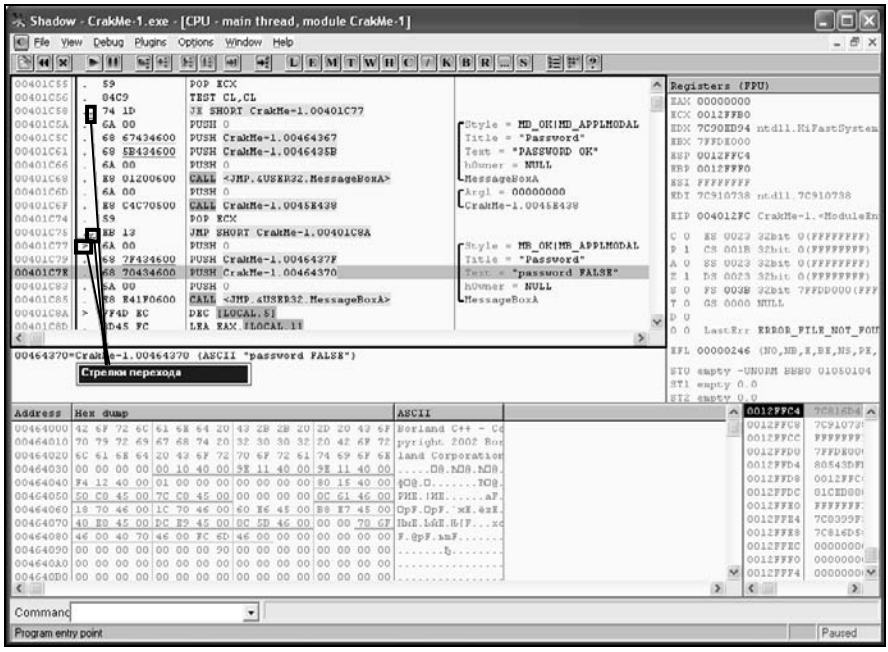

Рис. 5.9. Направление перехода

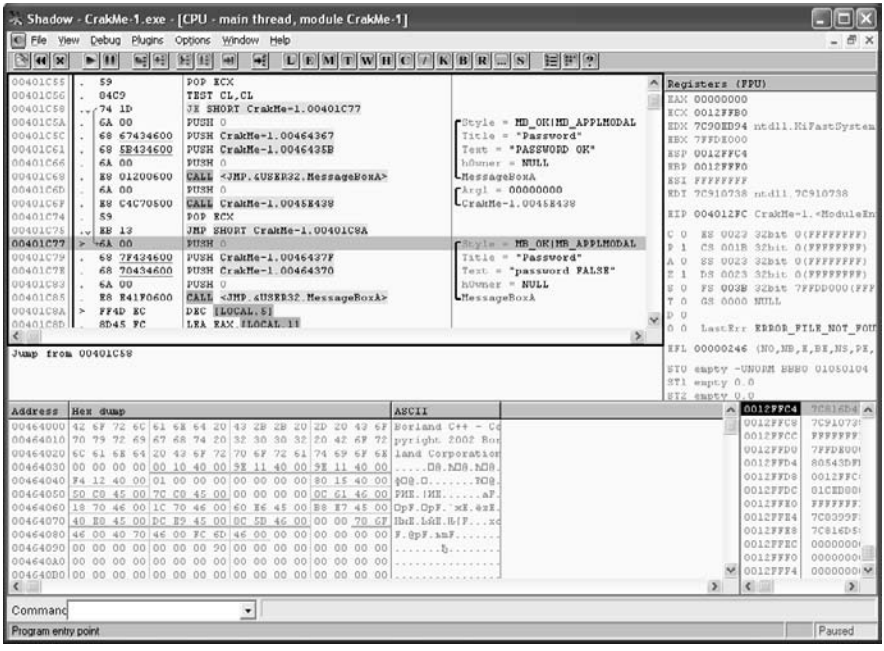

Рис. 5.10. Направление перехода — выделена строка

а если не верен, то на строку

```
00401C77    6A 00       PUSH 0
```

То есть все происходит в соответствии с теорией, которую мы рассмотрели в самом начале главы.

Таким образом, получается, что функция проверки правильности пароля находится выше строки:

```
00401C58    74 1D             JE SHORT CrakMe-1.00401C77
```

Давайте пролистаем главное окно отладчика немного вверх, вот до таких строк:

```
00401C0D    837D FC 00        CMP [LOCAL.1],0
00401C11    74 05             JE SHORT CrakMe-1.00401C18
00401C13    8B45 FC           MOV EAX,[LOCAL.1]
00401C16    EB 05             JMP SHORT CrakMe-1.00401C1D
00401C18    B8 5A434600       MOV EAX,CrakMe-1.0046435A
00401C1D    66:C745 E0 2000   MOV WORD PTR SS:[EBP-20],20
00401C23    33D2              XOR EDX,EDX
00401C25    8955 F4           MOV [LOCAL.3],EDX
00401C28    8D55 F4           LEA EDX,[LOCAL.3]
00401C2B    FF45 EC           INC [LOCAL.5]
00401C2E    8B83 F4020000     MOV EAX,DWORD PTR DS:[EBX+2F4]
00401C34    E8 FBC30400       CALL CrakMe-1.0044E034
00401C39    8D45 F4           LEA EAX,[LOCAL.3]
00401C3C    8D55 FC           LEA EDX,[LOCAL.1]
00401C3F    E8 50180600       CALL CrakMe-1.00463494
00401C44    50                PUSH EAX
```

Как вы знаете, функция ассемблера `CMP` выполняет сравнение. А после сравнения сразу стоит переход:

```
00401C11    74 05             JE SHORT CrakMe-1.00401C18
```

Давайте поставим брекпойнт на функцию:

```
CMP [LOCAL.1],0
```

Для этого нужно выделить строку с помощью мыши и нажать клавишу <F2>. Адрес окрасится красным цветом (рис. 5.11).

Теперь запустим программу в отладчике. Для этого нажмем клавишу <F9>. Наша программа запустилась, но в отладчик не попала. Что это значит? Это значит, что пока код, где мы поставили точку останова (брекпойнт), не выполнялся. Давайте введем любой пароль в окне программы CRAKME-1.EXE и нажмем кнопку **OK**. Вот! Мы попали в отладчик.

Теперь давайте будем трассировать программу с помощью клавиши <F8>, вот до этого момента (рис. 5.12).

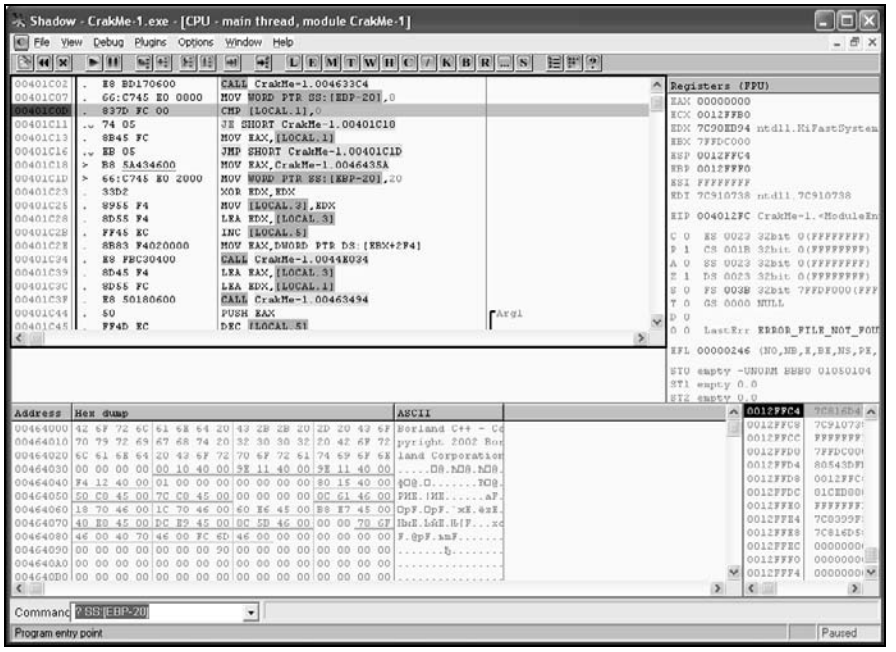

Рис. 5.11. Установка брекпойнта в OllyDbg

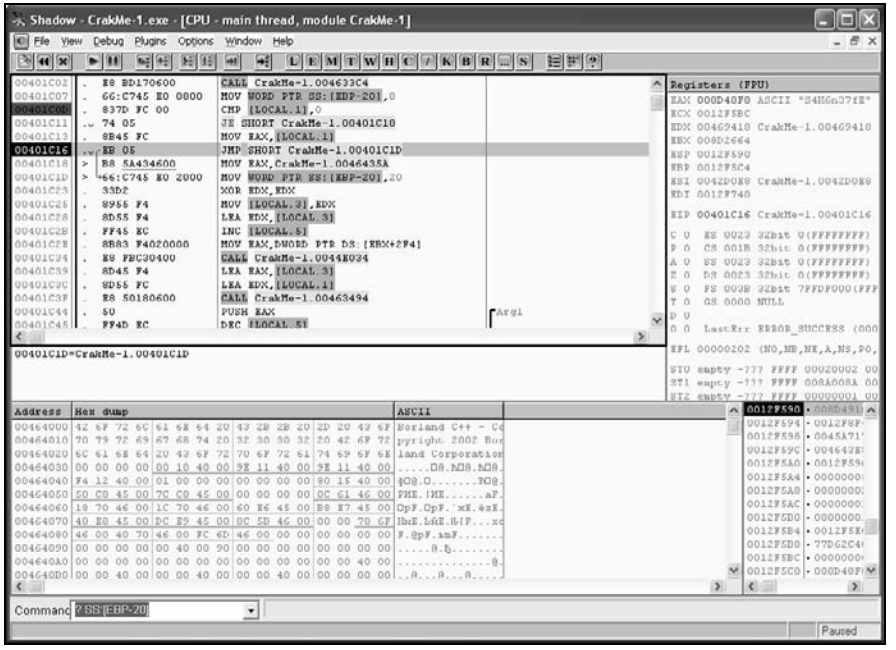

Рис. 5.12. Трассировка программы для нахождения пароля

И что же мы видим в окне регистров? А, точнее, в регистре EAX?

```
EAX      008D48F8      ASCII "S4K6n37fE"
```

Мы видим наш заветный пароль: `S4K6n37fE`

Давайте попробуем ввести его в окне исследуемой нами программы. Мы получим сообщение (рис. 5.13).

Рис. 5.13. Сообщение о верном пароле

Вот таким образом обходятся подобные защиты. Еще раз хочу заметить, что тут все несложно.

Примечание

Если вы не хотите, чтобы вашу программу взламывали таким простым способом, запомните правило: нельзя хранить свои сообщения в открытом виде!

С шифрованием сообщений

Мы продолжаем увлекательное исследование программ типа "запрос пароля". Таких программ большинство, поэтому мы и рассмотрим их подробнее.

В предыдущем разделе выяснилось, что ничего сложного в таких программах нет. Все у нас получилось, и мы даже не заметили, как быстро нашли пароль. Теперь давайте посмотрим, что получится, если мы применим на практике совет, что я дал в прошлой главе.

Мы попробуем подобрать пароль к программе CRAKME-2.EXE. Итак, начинаем. Запускаем программу (рис. 5.14).

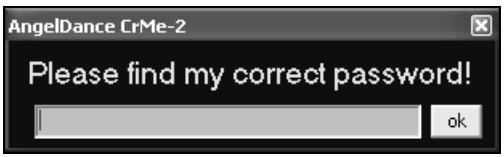

Рис. 5.14. Главное окно программы

Мы видим практически такое же окно. Пробуем ввести любой номер. Получаем окно с сообщением о неверном пароле (рис. 5.15). Как видите, пока все точно так же, как и в первом случае.

Рис. 5.15. Окно сообщения о неверном пароле

Давайте посмотрим, запакована программа или нет. Запускаем программу PEiD, загружаем в него нашу исследуемую программу (рис. 5.16).

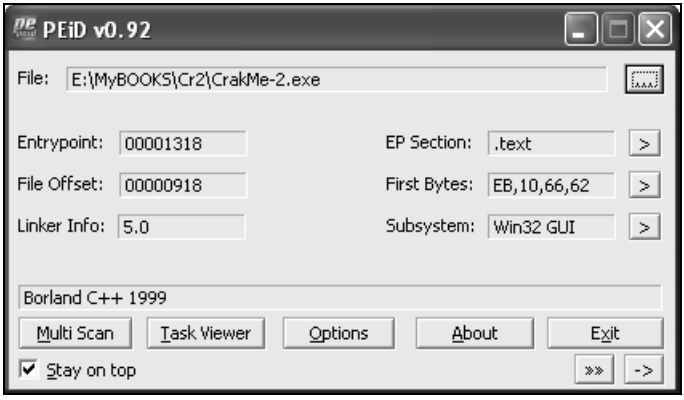

Рис. 5.16. Результаты анализа исследуемого файла, выполненного программой PEiD

Так, смотрим: программа тоже не запакована и не зашифрована. Ну, что же, очень хорошо. Запускаем отладчик OllyDbg и загружаем нашу программу в него (рис. 5.17).

Как видим, пока отличий нет. Все повторяется в точности, как и с программой CrakMe-1. Попробуем найти строки текста, как я описал это ранее. Что же мы видим? Текста нет! Это значит, что программист, создававший программу, постарался защитить ее от изменения (например, с помощью редактора ресурсов).

Как же быть? Можно, конечно, потихоньку трассировать всю программу, но так мы точно ничего не добьемся, поскольку программа может быть очень большой. Чтобы облегчить себе жизнь и не гадать, что за функция скрывается под именем `CrakMe-2.0045BA38`, мы сделаем следующее: запустим дизассемблер IDA, откроем в нем исследуемый файл CRAKME-2.EXE и подождем, пока IDA проанализирует наш файл (рис. 5.18).

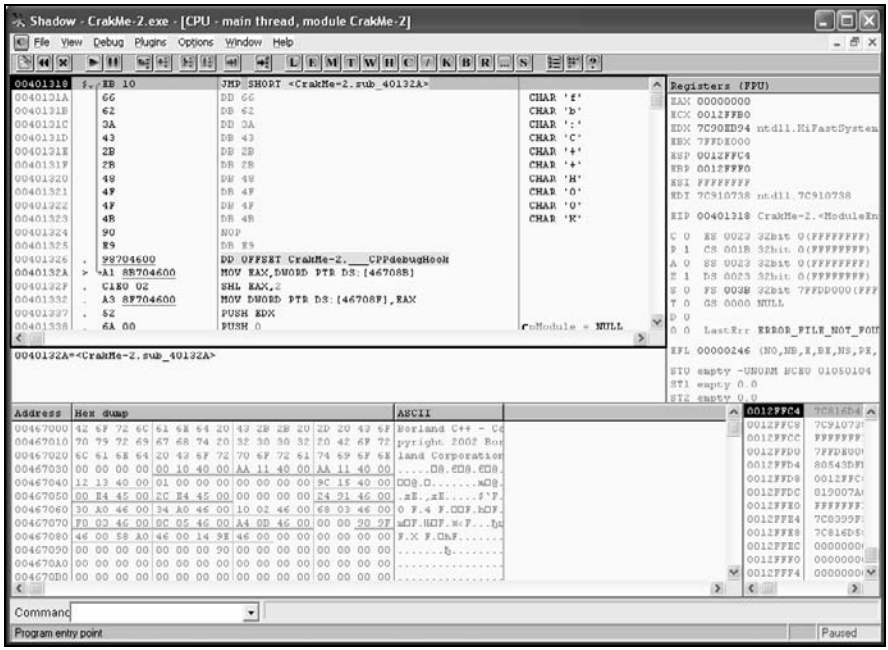

Рис. 5.17. Отладчик OllyDbg

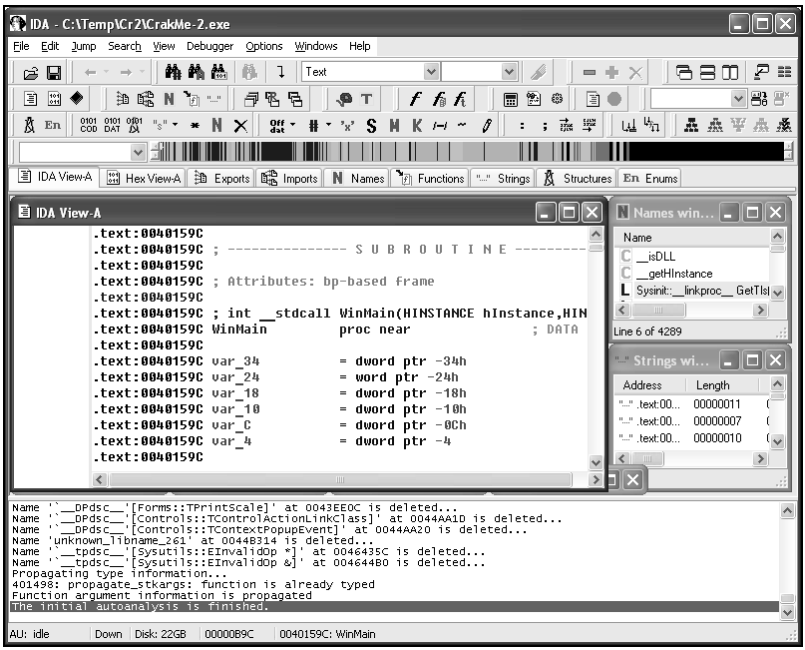

Рис. 5.18. Дизассемблер IDA

После этого выберем команду **File > Produce file > Create MAP file** и сохраним MAP-файл в той же папке, где находится файл CRAKME-2.EXE. Закрываем окно IDA — он нам больше не понадобится.

После этого в окне отладчика OlleDBg выберем команду **Plugins > GODUP Plugins > Map Loader > *Наш MAP-файл***. Наши функции сразу преобразились (рис 5.19).

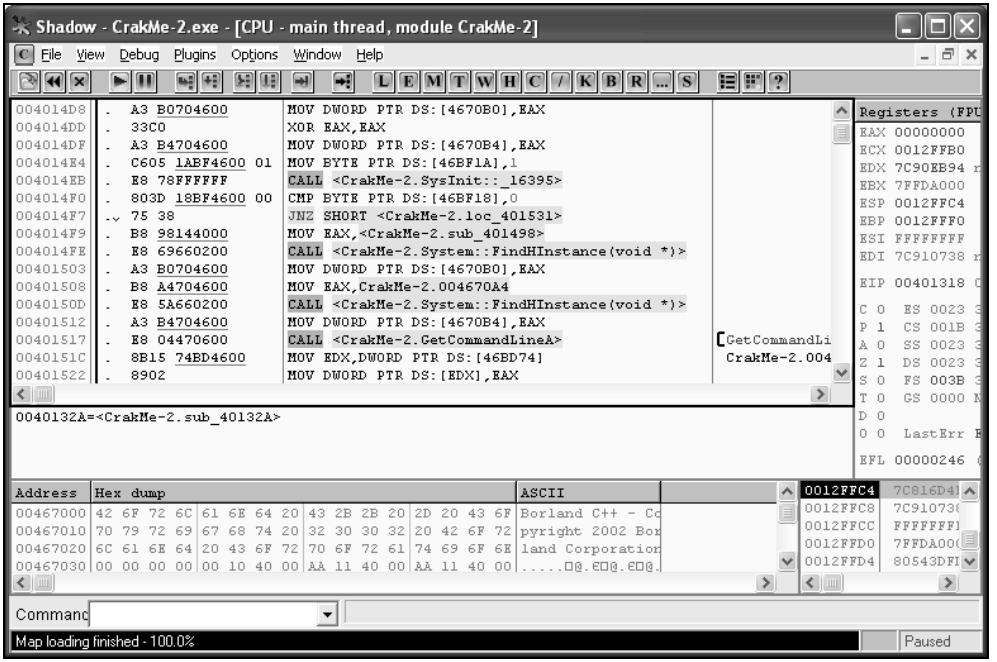

Рис. 5.19. Имена функций после загрузки MAP-файла

Читать функции стало намного приятнее. Но вот для того, чтобы найти в программе заветный пароль, надо "сказать" OllyDbg, чтобы он прервал выполнение исследуемой нами программы в момент ввода пароля или его анализа. Что для этого потребуется? Надо поставить брекпойнт. Но на какую функцию его поставить? Вот для того чтобы узнать, какие функции можно использовать, мы и запускали программу PEiD. Эта программа показала, что наша программа написана на Borland C++ 1999, значит, мы можем использовать все функции из этого компилятора.

Примечание

Список всех используемых программой функций можно посмотреть в отладчике OllyDbg — для этого надо нажать комбинацию клавиш <Ctrl>+<M>. Запомните эту полезную для исследования программ комбинацию.

Так, теперь нам нужны функции работы со строками. Почему именно они? Дело в том, что раз мы вводим пароль в виде строки, то скорее всего и проверять его программа будет с помощью функций для работы со строками. А вот чтобы узнать эти функции, вам и пригодится знание языка программирования высокого уровня.

Выбираем экспортируемые программой функции для работы со строками. Например, давайте попробуем поставить брекпойнт на функцию `lstrcpyA` и посмотреть, что получится.

Брекпойнт поставлен, но при запуске программы мы в OllyDbg не попадаем. Это значит, что мы поставили брекпойнт не на ту функцию. Давайте подумаем и внимательно посмотрим на программу. Если запустить CrakMe-2 и ввести любой пароль, то при нажатии кнопки **OK** мы получаем окно (см. рис. 5.15) Вам не кажется, что это окно похоже на результат вызова обычной функции `MessageBox`? Попробуем поставить брекпойнт на `MessageBoxA`. Поставили? Нажимаем в OllyDbg клавишу <F9>. Запустилась наша программа. Введем в окне любые символы (я ввел "12345") и нажимаем кнопку **OK**. Вот! Мы в отладчике. Что же мы видим (рис. 5.20)?

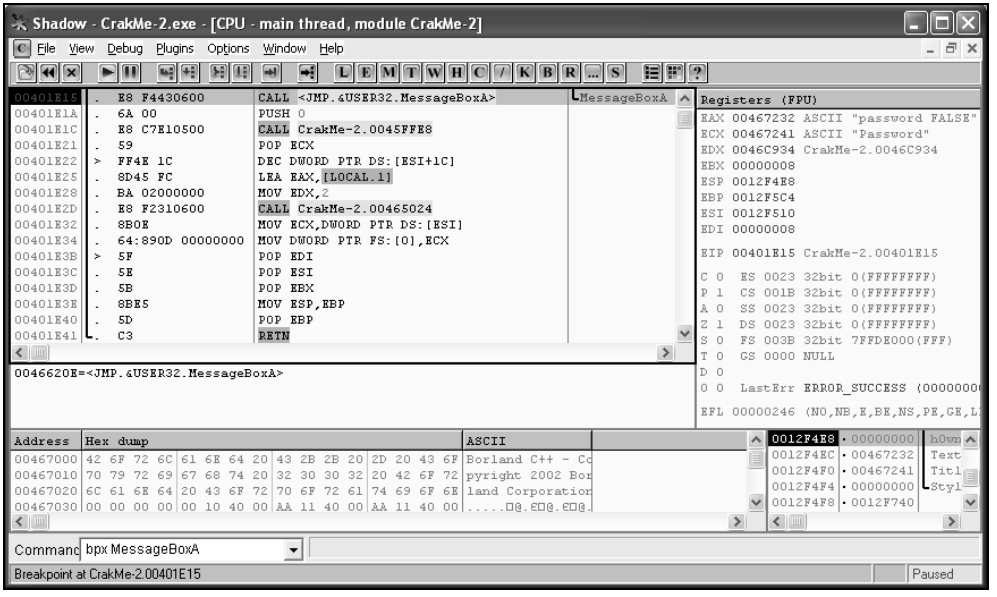

Рис. 5.20. Срабатывание брекпойнта на `MessageBoxA`

Мы остановились как раз на вызове функции вывода `MessageBox`. В регистрах EAX, ECX, EDX мы видим строки текста, которые отображаются в окне

сообщения о неверном пароле. Чтобы нам удобнее было читать, загрузим в отладчик наш MAP-файл. Все функции приобрели понятный вид. Пролистаем главное окно немного вверх (рис. 5.21).

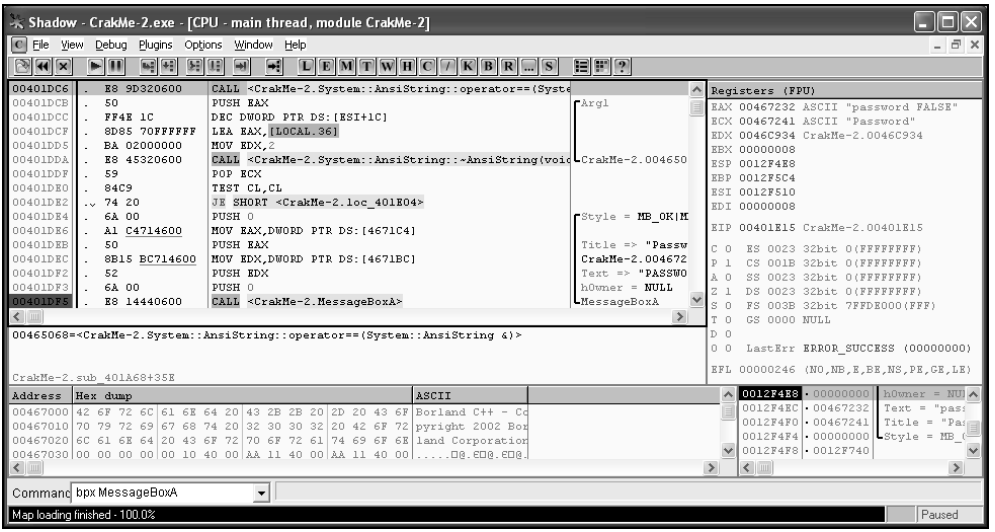

Рис. 5.21. Функции проверки пароля

Знакомый нам вид — это функции проверки пароля. А чуть выше мы видим интересную строку:

```
00401DC6 E8 9D320600 CALL <CrakMe-2.System::AnsiString::operator==
  (System::AnsiString &>
```

На что это похоже? На первый взгляд, функция сравнивает две строки текста. А зачем программа что-то сравнивает? Правильно! Она берет введенную нами строку и сравнивает ее с зашифрованной в программе. Если есть совпадение, значит, появится сообщение о том, что пароль верный, а если нет — о том, что неверный. Перезапустим программу и установим брекпойнт по адресу 00401DC6. Вводим в окне программы символы пароля и попадаем на наш брекпойнт (рис. 5.22).

Теперь, чтобы посмотреть, что же там сравнивает программа, нам необходимо зайти в эту функцию. Для этого трассируем программу с помощью клавиши <F7>. Вот мы "попали" внутрь функции сравнения. Доходим до этого места (рис. 5.23).

Что же мы видим в регистрах EAX и EDX? В EAX — введенные нами символы, а в EDX — символы, с которыми сравниваются наши. Вывод: символы

из регистра EDX и есть правильный пароль. Проверим это, введя их в окне нашей программы CrakMe-2. Вот мы и получили сообщение, которое видим на рис. 5.24.

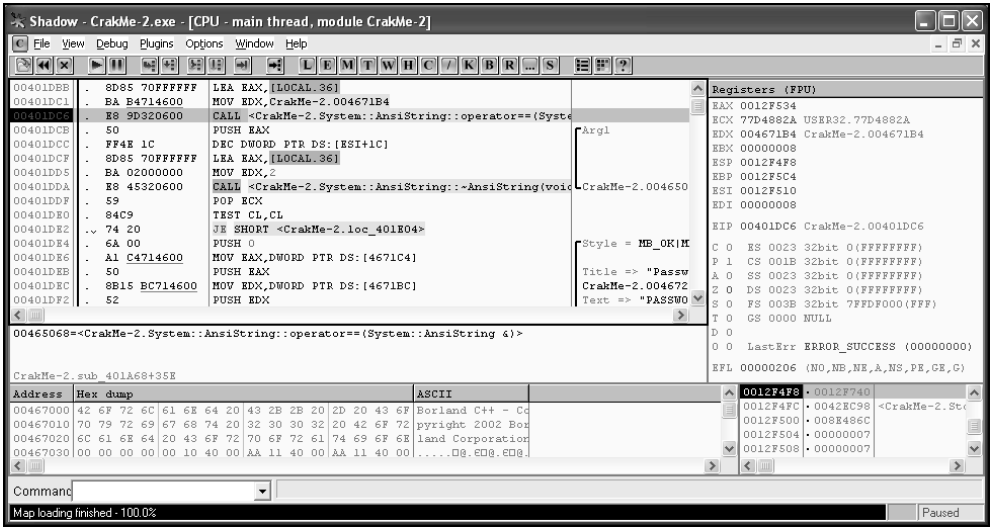

Рис. 5.22. Срабатывание брекпойнта на функции сравнения

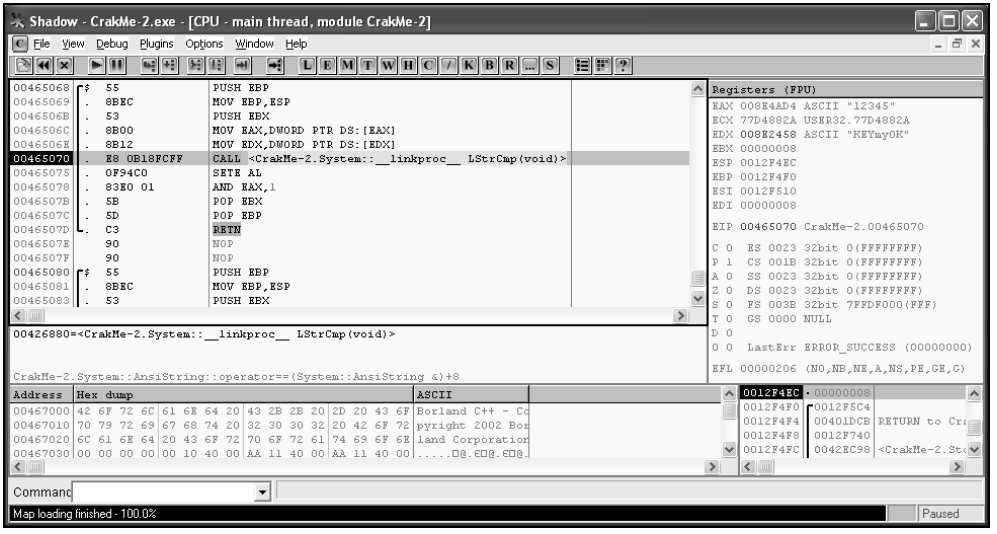

Рис. 5.23. Внутри функции сравнения

Рис. 5.24. Сообщение о верном пароле

Вот и все! Как видите, шифрование сообщений сильно усложняет жизнь взломщику. Хотя это и не даст 100%-ной гарантии от взлома вашей программы, но большинство начинающих взломщиков остановит.

Примечание

Чтобы еще больше усложнить жизнь взломщику, запомните правило: **никогда не выводите сообщение СРАЗУ после проверки пароля на правильность.** Это все равно, что указать взломщику, что вот тут у вас проверяется пароль.

Улучшенная защита программ с шифрованием сообщений

Как вы видели, защитить программу бывает довольно непросто, однако даже простые методы усложняют крекеру жизнь. К сожалению (или к счастью, это кому как), программисты не хотят тратить время даже на простую, элементарную защиту своих программ. А ведь это не так сложно, как кажется на первый взгляд. Я покажу на примере, как вырастут затраты труда и времени, если сделать простейшую защиту.

Давайте посмотрим, что же изменится, если применить правила, о которых я говорил, а именно:

- никогда не храните ваши сообщения в открытом виде;
- никогда не выводите сообщения сразу после проверки пароля на правильность.

Будем исследовать программу CRAKME-3.EXE. Запускаем ее и смотрим на окно программы (рис. 5.25).

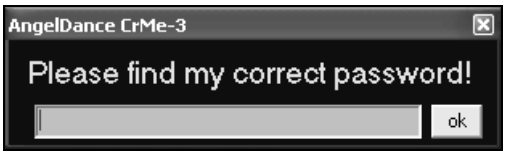

Рис. 5.25. Главное окно программы

Мы опять видим практически то же самое окно с полем для ввода пароля. Попробуем ввести любые символы и посмотреть на результат. Вводим "12345", нажимаем кнопку **OK**. Сначала ничего не происходит; немного погодя получаем сообщение о неверном пароле (рис. 5.26).

Рис. 5.26. Сообщение о неверном пароле

Что же мы имеем на этот раз? На первый взгляд, ничего нового. Однако если хорошо подумать, то получается, что программа запрашивает у нас пароль, что-то с ним делает (или ждет какое-то время) и выдает нам результат (мы уже занимаемся анализом!). Давайте запустим PEiD и посмотрим, что он нам скажет (рис. 5.27).

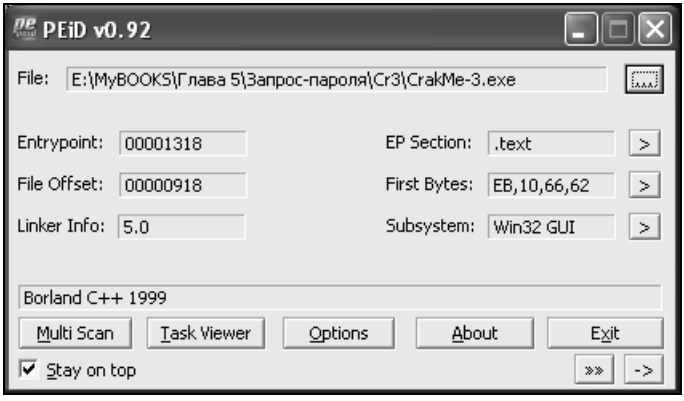

Рис. 5.27. Результаты анализа исследуемого файла, выполненного программой PEiD

Так. Значит, программа не запакована и не зашифрована. Это уже хорошо. Запускаем OllyDbg и загружаем в него нашу программу (рис. 5.28).

Теперь пробуем найти все текстовые строки программы. Вдруг нам повезло, и программист не стал шифровать свои данные, как это было в программе CRAKME-1.EXE. Мы видим, что не все строки зашифрованы, но вот самих сообщений, которые выводятся в случае правильности (или неправильности) пароля нет. Значит, программист все-таки зашифровал свои сообщения.

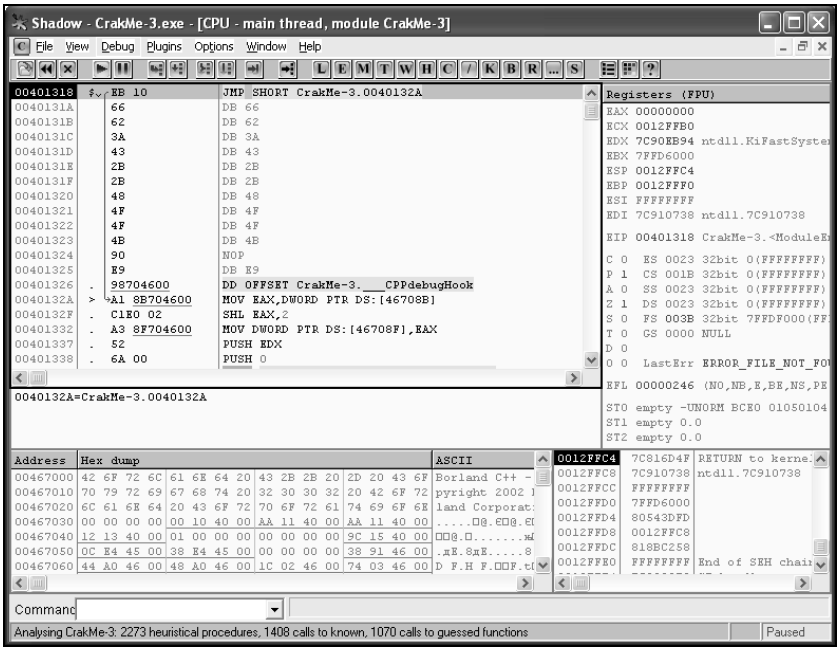

Рис. 5.28. OllyDbg с загруженным для исследования файлом

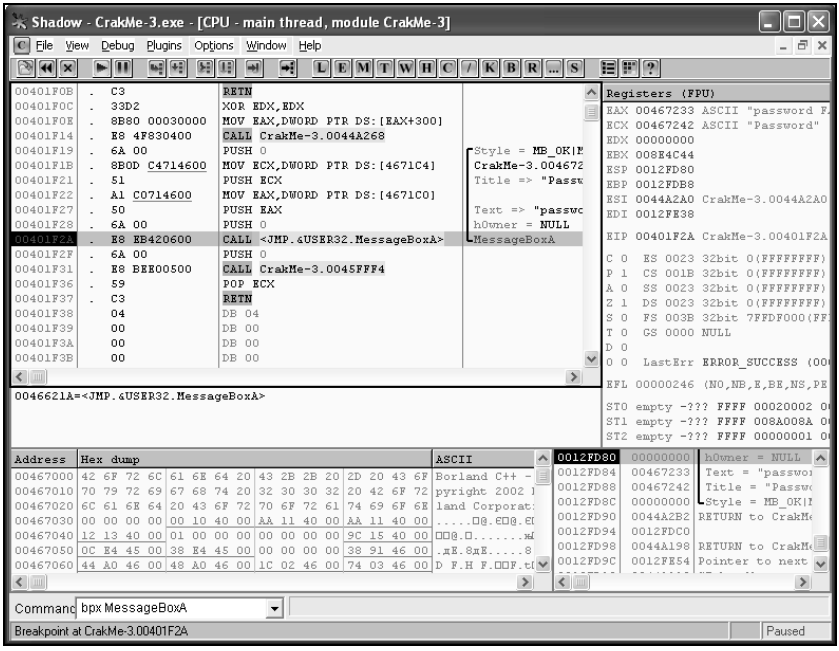

Рис. 5.29. OllyDbg: брекпойнт на функции `MessageBoxA`

Что же нам делать? Попробуем поставить брекпойнт на функцию `MessageBoxA`. Запускаем программу в отладчике, вводим в окне пароль "12345" и нажимаем кнопку **OK**. Теперь мы в отладчике (рис. 5.29).

Что же мы видим? Вывод сообщения о неверном пароле. Если посмотреть чуть выше, то мы увидим, что там находится сообщение о верном пароле. Причем все сообщения уже расшифрованы. Это значит, что где-то в программе сообщения расшифровываются. А раз так, то расшифровывается и пароль (хотя формально это вовсе не обязательно, поскольку сравниваться могут не пароли, а хеш-функции). Что же делать? Можно, конечно, попытаться проследить, откуда эта функция вызывается, но нет гарантии, что мы найдем нужное место достаточно быстро (хотя этот метод довольно часто срабатывает). В данном случае этот способ нам не подходит. Давайте подумаем. Есть программа, в которую вводится пароль, он сравнивается с паролем, который есть в самой программе. Стоп! Раз он "сравнивается", значит можно попробовать "отловить" функцию сравнения. Какие же функции нам нужны? А вот для этого мы и пытались определить язык, на котором написана программа. Поскольку она написана на Borland C++, берем справочник по этому компилятору. Там есть несколько функций сравнения:

```
extern PACKAGE int __fastcall CompareStr(const AnsiString S1,
const AnsiString S2);
extern PACKAGE int __fastcall CompareText(const AnsiString S1,
const AnsiString S2);
int strcmp(const char *s1, const char *s2);
```

Эти функции предназначены для сравнения строк. И не забудьте, что еще есть простой оператор сравнения:

```
(const1*)==(const2*);
```

Его применяют, когда надо сравнить, например, так, как показано в листинге 5.2.

Листинг 5.2. Пример сравнения строк

```
AnsiString pas1="Pas1234";
AnsiString pas2="456Pas3";
if(pas1==pas2)
       ShowMessage("Password is TRUE");
else
       ShowMessage("Password is FALSE");
```

Эту комбинацию довольно часто применяют в программах на языке C++, поэтому о ней ни в коем случае не стоит забывать.

Теперь давайте поставим брекпойнт на функцию `CompareStr`. Поставили, запускаем программу, вводим в нее любые значения, нажимаем кнопку **OK** и… Ничего не происходит. Значит, как я уже говорил, эта функция не используется. Попробуем поставить брекпойнты на функции `CompareText` и `strcmp`. Поставили, запускаем программу. И опять ничего. Значит и эти функции не используются для сравнения.

Примечание

Маленький секрет: можно посмотреть, какие функции используются (вызываются) в программе — для этого в окне OllyDbg надо нажать комбинацию клавиш <Ctrl>+<N>. Мы получим вот такое окно (рис. 5.30).

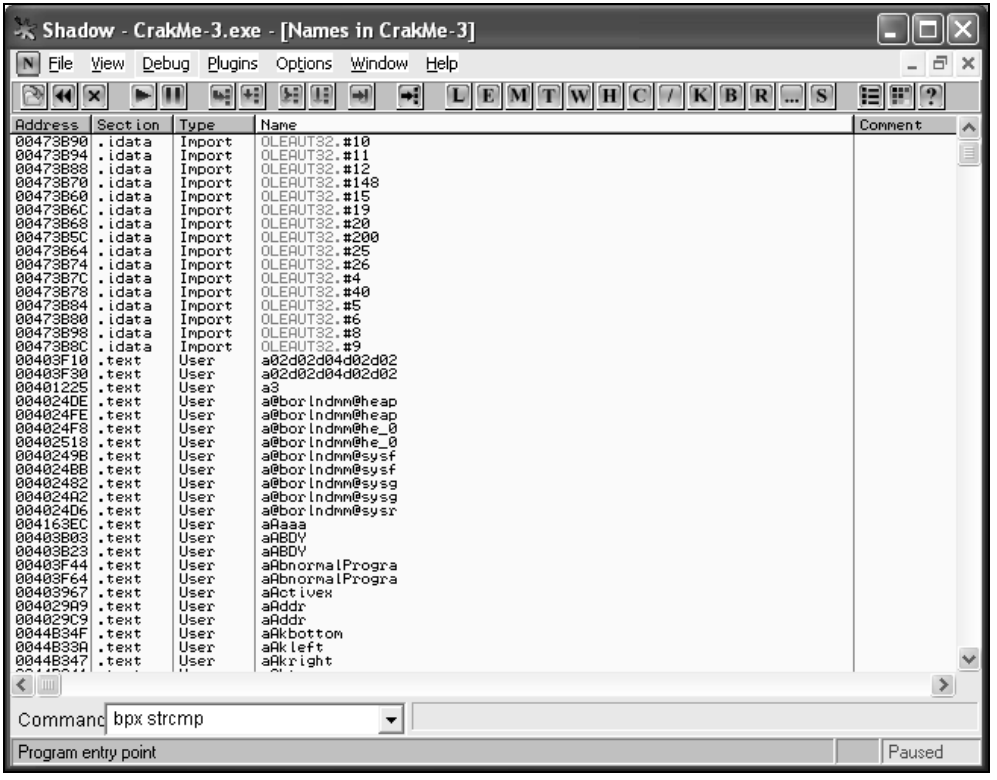

Рис. 5.30. Функции, используемые программой

В столбце **Name** отображается имя функции. Конечно их тут много, но ведь нам не нужны все. Самые интересные — системные функции. Если просто набрать "system", то быстрый поиск перенесет нас к ним (рис. 5.31).

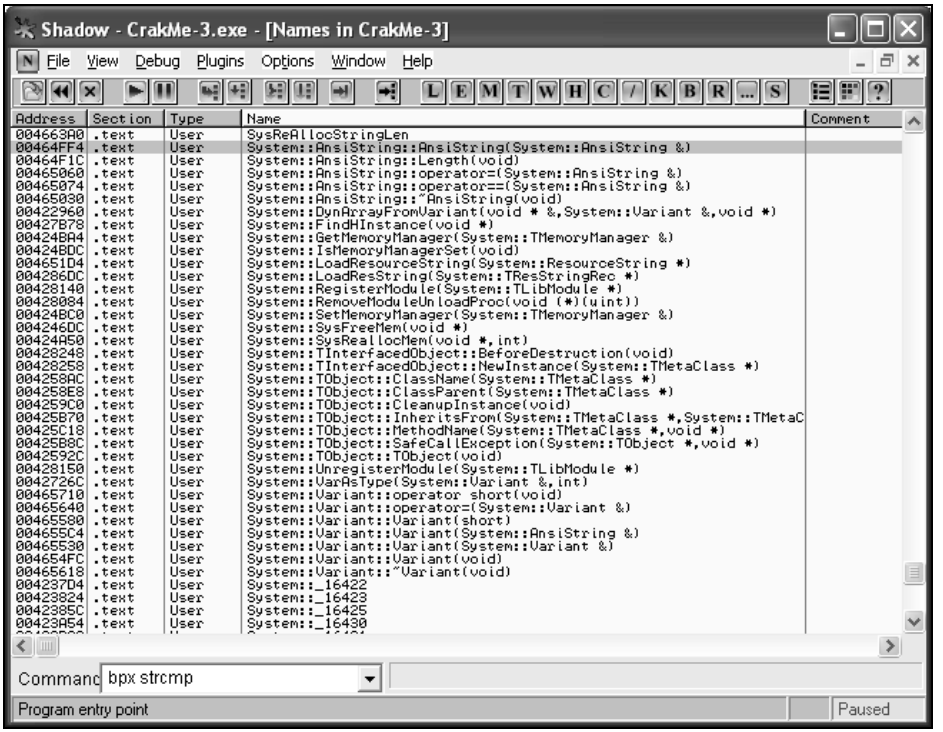

Рис. 5.31. Системные функции, используемые программой

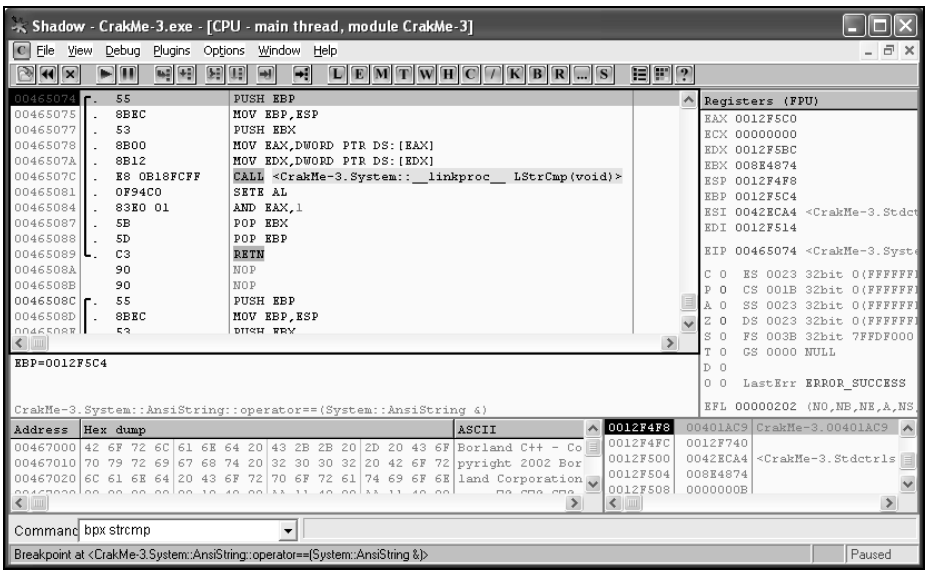

Рис. 5.32. OllyDBg при срабатывании брекпойнта

Мы видим очень интересные системные функции, в том числе:

```
System::AnsiString::operator==(System::AnsiString &)
```

Давайте поставим на нее брекпойнт. Для этого выделим ее строку и нажмем клавишу <F2>. Запускаем программу, вводим любые символы (я ввел "12345") и попадаем вот сюда (рис. 5.32).

Начинаем трассировать программу с помощью клавиши <F9> до строки.

```
0046507C  E8 0B18FCFF  CALL <CrakMe-3.System::__linkproc__ LStrCmp(void)>
```

При этом вызове функции ничего не происходит. Нажимаем клавишу <F9> еще раз и попадаем сюда же (рис. 5.33).

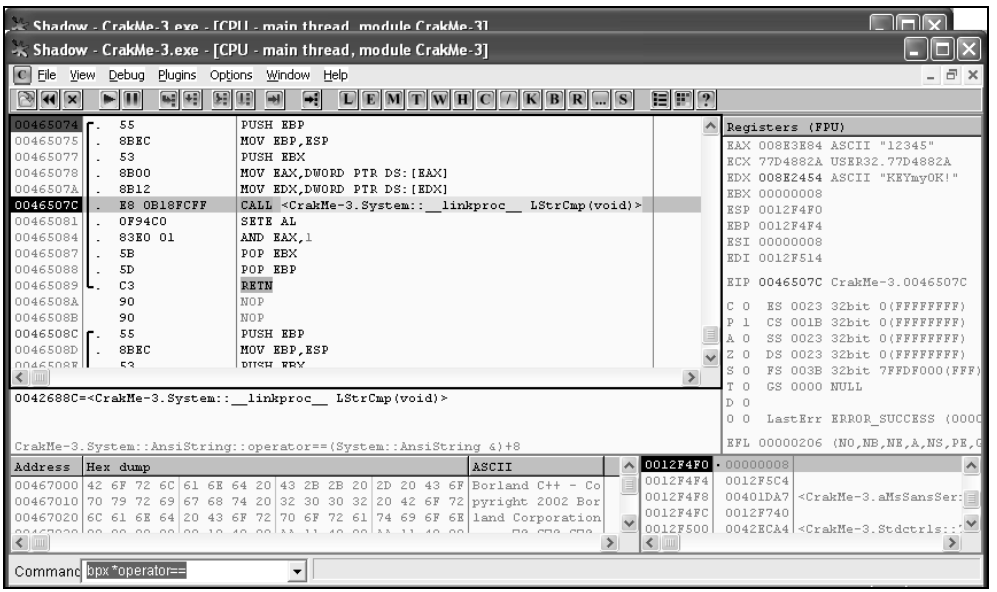

Рис. 5.33. OllyDBg при срабатывании брекпойнта

А вот тут при вызове функции в регистрах EAX, EDC есть интересные значения! Мы нашли пароль!

Вот так использование программистом правил помогает защитить программу. Конечно, защита не будет полной, если использовать только эти два правила, но начинающего крекера уже сможет остановить. А для еще большего осложнения жизни крекеров рекомендуется использовать в программе две строки — одну зашифрованную, а другую нет. С зашифрованной работают, а незашифрованную оставляют для ловушки (или просто ничего с ней не делают).

Крекеру придется поломать голову, каким образом вы добираетесь до этой строки (про ловушки мы поговорим позднее).

Примечание

Для тренировки попробуйте найти пароль в программе CrakMe-4.

Программы с динамической проверкой ввода пароля

Здесь мы рассмотрим программы с такой известной защитой, как *динамическая проверка ввода пароля*. На самом деле в этот класс попадает множество программ, различающихся многообразием способов проверки. Мы рассмотрим такой распространенный класс, как проверка пароля программой в случайный промежуток времени.

Что представляют собой такие программы? Представьте себе обычную программу. Как происходит в ней проверка на правильность пароля? Вы вводите в поле какую-либо информацию, нажимаете кнопку и программа проверяет, верный пароль или нет. В программах с динамической проверкой пароля это может выполняться несколькими способами:

- правильность ввода символов проверяется "на лету", т. е. одновременно с вводом;
- есть специальная функция, которая активируется с каким-то промежутком времени; чаще всего это обычный таймер — как только он активируется, проверяется правильность пароля и программа принимает решение о дальнейших действиях.

Как видите, это все не так уж сложно. Конечно, это сложнее, чем фиксированная проверка, но ненамного. Первое, что нам потребуется, — установить брекпойнт на доступ к памяти с вводимыми нами данными. Далее достаточно перехватить в самой программе процесс сравнения этих символов, и мы найдем верный пароль.

Это все теория, теперь давайте попробуем сделать это на практике.

Возьмем программу CrakMe-4. При запуске она покажет нам окно (рис. 5.34).

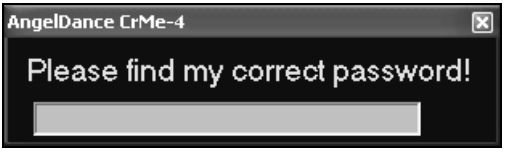

Рис. 5.34. Главное окно программы

Как видите, оно почти ничем не отличается от окон из наших предыдущих экспериментов. Есть только одна маленькая деталь: кнопки проверки пароля в нем нет. Попробуем ввести что-нибудь в окне, например, "1234567890". Ничего не произошло.

Проанализируем эту программу. Для начала посмотрим, что покажет PEiD. Утилита PEiD показывает нам, что программа ничем не запакована и не зашифрована (я еще раз хочу подчеркнуть, что при анализе нельзя доверять только одной программе. Лучше всего использовать 3-4 разных анализатора. У нас программы простые, созданные для демонстрации, поэтому мы и доверяем только одной программе). Раз программа ничем не зашифрована и не запакована, давайте загрузим эту программу в OllyDbg (рис. 5.35).

Примечание

Обратите внимание: на рис. 5.35 видно, на каком языке написана наша программа.

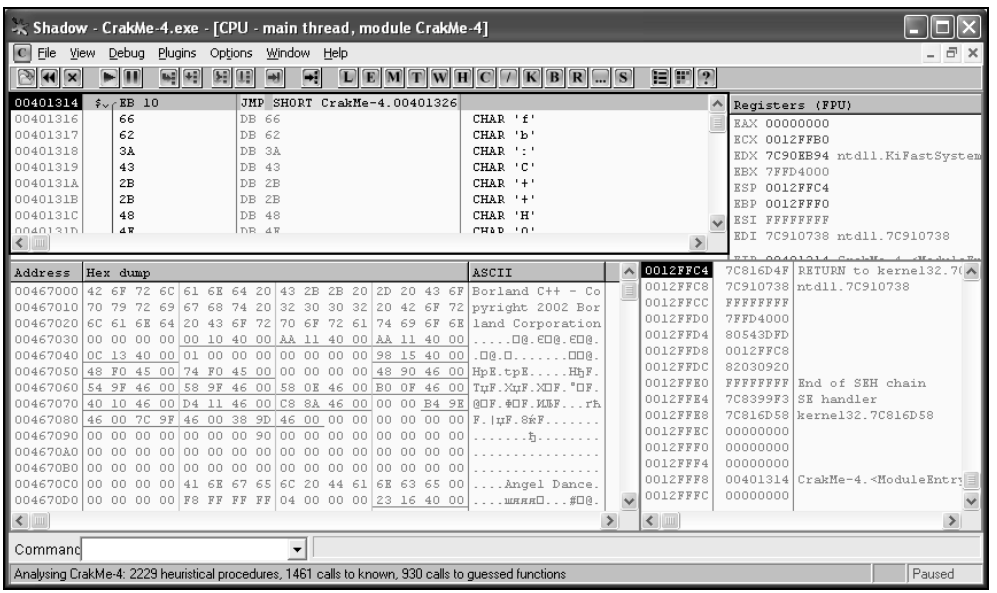

Рис. 5.35. Программа в отладчике

Давайте попробуем установить брекпойнт на функцию доступа к введенной нами строке. Точнее, мы установим брекпойнт на функцию доступа к памяти, содержащей данную строку.

Для этого загрузим программу в OllyDbg, введем в ее окне любые символы (я ввел "1234567890") и нажмем кнопку паузы в окне OllyDbg. Мы заморозили

программу CrakMe-4. Теперь нужно исследовать всю память, используемую программой. Для этого нажмем в окне OllyDBg комбинацию клавиш <Alt>+<M> и получим окно, показанное на рис. 5.36.

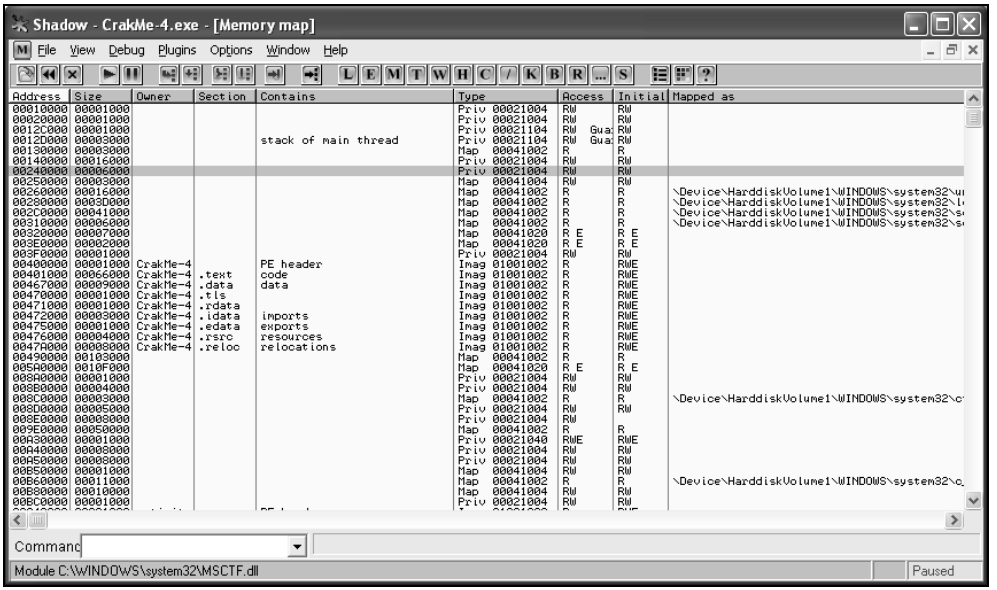

Рис. 5.36. Окно **Memory map** отладчика OllyDBg

Далее нам необходимо найти область памяти, которая содержит искомую строку. Для этого нажимаем комбинацию клавиш <Ctrl>+<B>, в открывшемся окне (рис. 5.37) вводим искомые символы и нажимаем кнопку **OK**.

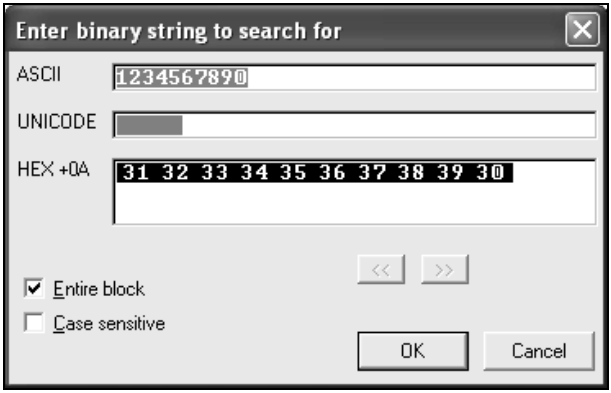

Рис. 5.37. Поиск строки в памяти

Так, отладчик OllyDbg нашел в памяти наши символы. Установим брекпойнт на доступ к этим ячейкам памяти (рис. 5.38) и "разморозим" нашу программу нажатием клавиши <F9>.

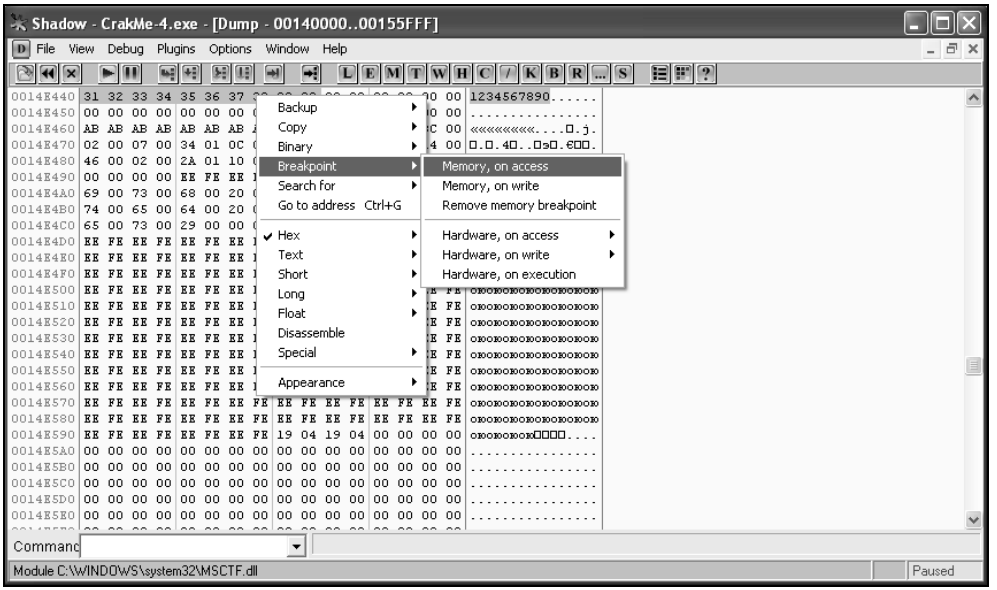

Рис. 5.38. Установка брекпойнта на доступ к памяти

После этого мы попадаем в то место программы, где происходит доступ к нашим данным. Как найти данные, вы уже можете догадаться сами.

А теперь я хочу рассказать вам о более быстром способе нахождения пароля.

Для начала мы создадим MAP-файл программы CrakMe-4 в IDA. При открытии программы в OllyDbg мы откроем ее вместе с MAP-файлом (как это делать, я уже неоднократно говорил). Без этого, конечно, можно и обойтись, просто так удобнее разбираться в ссылках программы.

Далее устанавливаем брекпойнт таким способом. Сначала осуществляем поиск в нашей программе всех используемых модулей (рис. 5.39).

Затем ищем модуль сравнения (точно так же, как в предыдущих примерах), т. е. строки, показанные на рис. 5.40.

Нас интересуют строки, осуществляющие операцию сравнения. Поскольку мы знаем, что программа написана на языке Borland C++ (это нам показала программа-анализатор PEiD), значит, и операция сравнения будет выглядеть так, как принято в этом языке, а именно:

```
operand1 == operand2
```

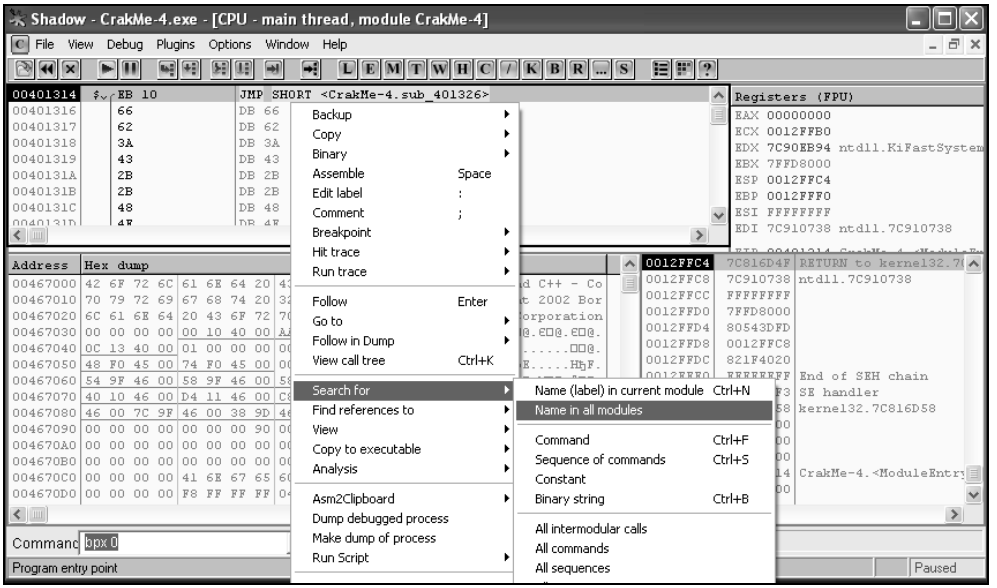

Рис. 5.39. Поиск модулей в программе

00465BEC	CrakMe-4	.text	User	System::AnsiString::operator=(System::AnsiString &)
74784BEC			User	System::AnsiString::operator=(System::AnsiString &)
77DA5060	USER32	.rsrc	User	System::AnsiString::operator=(System::AnsiString &)
7C865060	kernel32	.text	User	System::AnsiString::operator=(System::AnsiString &)
00465C8C	CrakMe-4	.text	User	System::AnsiString::operator==(System::AnsiString &)
74784C8C			User	System::AnsiString::operator==(System::AnsiString &)
77DA5074	USER32	.rsrc	User	System::AnsiString::operator==(System::AnsiString &)
7C865074	kernel32	.text	User	System::AnsiString::operator==(System::AnsiString &)

Рис. 5.40. Функции сравнения, используемые программой

А таких инструкций мы видим несколько, начиная с

`System::AnsiString::operator==(System::AnsiString &)`

Какая же нам нужна? Проще всего поставить брекпойнты на все эти операции.

Таблица 5.1. *Операции сравнения*

Address	Module	Section	Type	Name
00465BEC	CrakMe-4	.text	User	System::AnsiString::operator= (System::AnsiString &)
74784BEC			User	System::AnsiString::operator= (System::AnsiString &)
77DA5060	USER32	.rsrc	User	System::AnsiString::operator= (System::AnsiString &)

Таблица 5.1 *(окончание)*

Address	Module	Section	Type	Name
7C865060	kernel32	.text	User	System::AnsiString::operator= (System::AnsiString &)
00465C8C	CrakMe-4	.text	User	System::AnsiString::operator== (System::AnsiString &)
74784C8C			User	System::AnsiString::operator== (System::AnsiString &)
77DA5074	USER32	.rsrc	User	System::AnsiString::operator== (System::AnsiString &)
7C865074	kernel32	.text	User	System::AnsiString::operator== (System::AnsiString &)

Но если немного подумать, то станет очевидным, что нам нужен только один брекпойнт на функцию:

```
00465C8C   CrakMe-4   .text    User   System::AnsiString::operator==
                                       (System::AnsiString &)
```

Почему именно здесь? А давайте посмотрим на табл. 5.1. Видим, что функции сравнения идут, только начиная с адреса 00465C8C. То есть значения сравнивают именно эти функции:

```
00465C8C    CrakMe-4    .text   User    System::AnsiString::operator==
                                        (System::AnsiString &)
74784C8C                        User    System::AnsiString::operator==
                                        (System::AnsiString &)
77DA5074    USER32      .rsrc   User    System::AnsiString::operator==
                                        (System::AnsiString &)
7C865074    kernel32    .text   User    System::AnsiString::operator==
                                        (System::AnsiString &)
```

А теперь посмотрим на столбец под названием **Module**. Что же мы там видим? Все функции сравнения выполняются в разных модулях. Например,

```
7C865074    kernel32    .text   User    System::AnsiString::operator==
                                        (System::AnsiString &)
```

выполняется в модуле `kernel32`, а

```
77DA5074    USER32      .rsrc   User    System::AnsiString::operator==
                                        (System::AnsiString &)
```

выполняется в модуле `USER32`.

Нам же надо найти функцию, выполняющую сравнение в программе. Такая тут только одна:

```
00465C8C    CrakMe-4    .text   User    System::AnsiString::operator==
                                        (System::AnsiString &)
```

Вот на нее-то мы и поставим брекпойнт, а после этого запустим нашу программу.

Что мы видим теперь? Ничего не произошло. Давайте порассуждаем — почему? Раз у нас нет кнопки проверки пароля, значит, проверка осуществляется динамически, т. е. посимвольно.

Давайте проверим эту идею. Для этого будем вводить не сразу все символы "1234567890", а постепенно, т. е. введем "1", подождем секунду, потом введем "2", опять подождем и т. д. Попробуем. Получилось! Как только мы ввели "8", мы вернулись в отладчик OllyDbg. Означать это может только одно: программа проверяет, сколько символов мы ввели, и если введено больше или меньше 8 символов, она не работает. Это в свою очередь означает, что пароль состоит из восьми символов.

Итак, мы в отладчике. Протрассируем программу на пять шагов до функции (рис. 5.41):

```
00465C94    E8 F7E5FBFF   CALL <CrakMe-4.System::__linkproc__ LStrCmp(void)>
```

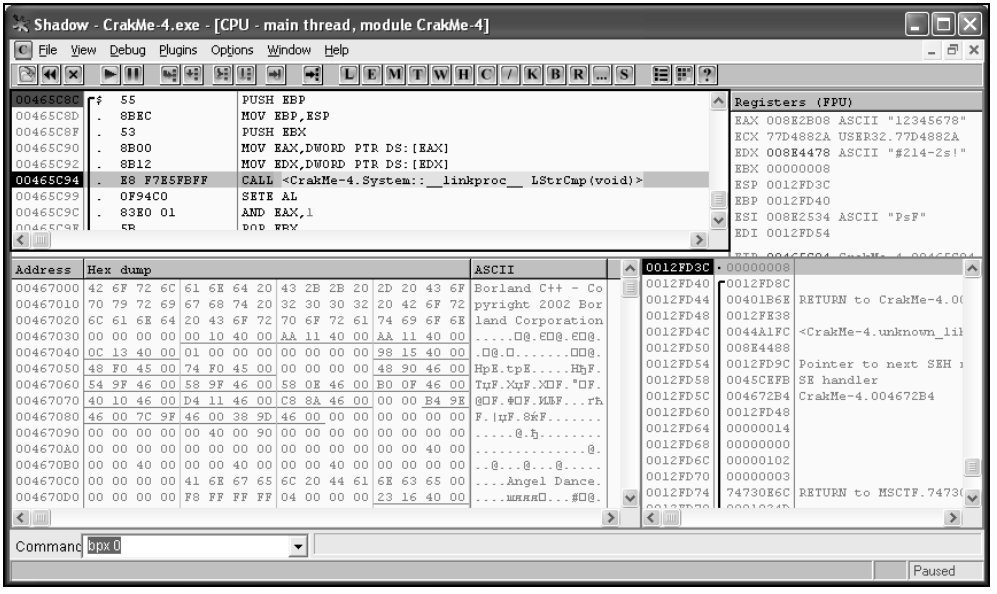

Рис. 5.41. Вызов программой функции сравнения

Мы остановились на функции сравнения. Она должна сравнивать между собой регистры, и, как видим, именно это она и делает. А в окне регистров мы видим, что именно она сравнивает:

```
EAX   008E2B08   ASCII "12345678"

EDX   008E4478   ASCII "#214-2s!"
```

В регистре EAX мы видим введенные нами символы, а в регистре EDX — данные, сравнение с которыми выполняется. А зачем программе сравнивать наши данные с другими? Правильно, это и есть верный пароль.

Попробуем его ввести (рис. 5.42).

И получим ответ программы, показанный на рис. 5.43.

Рис. 5.42. Ввод пароля

Рис. 5.43. Окно сообщения о верном пароле

Вот верный пароль и найден!

Несложно сделать так, чтобы программа выводила сообщение о верном пароле при вводе любого набора из восьми символов. Для этого нужно выполнить трассировку до места сравнения регистров:

```
EAX   008E2B08   ASCII "12345678"
EDX   008E4478   ASCII "#214-2s!"
```

После этого с помощью клавиши <F8> трассируем программу до ближайшей функции сравнения (рис. 5.44).

Это оказывается строка:

```
00401BA0   84C9    TEST CL, CL
```

после которой идет оператор перехода

```
00401BA2   74 3A   JE SHORT <CrakMe-4.loc_401BDE>
```

Что тут происходит? Все очень просто. После сравнения, если пароль неверный, будет переход на `JE SHORT <CrakMe-4.loc_401BDE>`, а если верный, т. е. введенные нами символы соответствуют заданному в программе паролю, то перехода не будет, и программа продолжает выполнение со следующего

операнда. Для того чтобы пароль всегда был верным, необходимо этот переход "занопить", т. е. записать туда команду `NOP`. Получаем:

```
00401BA0    84C9    TEST CL,CL
00401BA2    90      NOP
00401BA3    90      NOP
00401BA4    33D2    XOR EDX,EDX
```

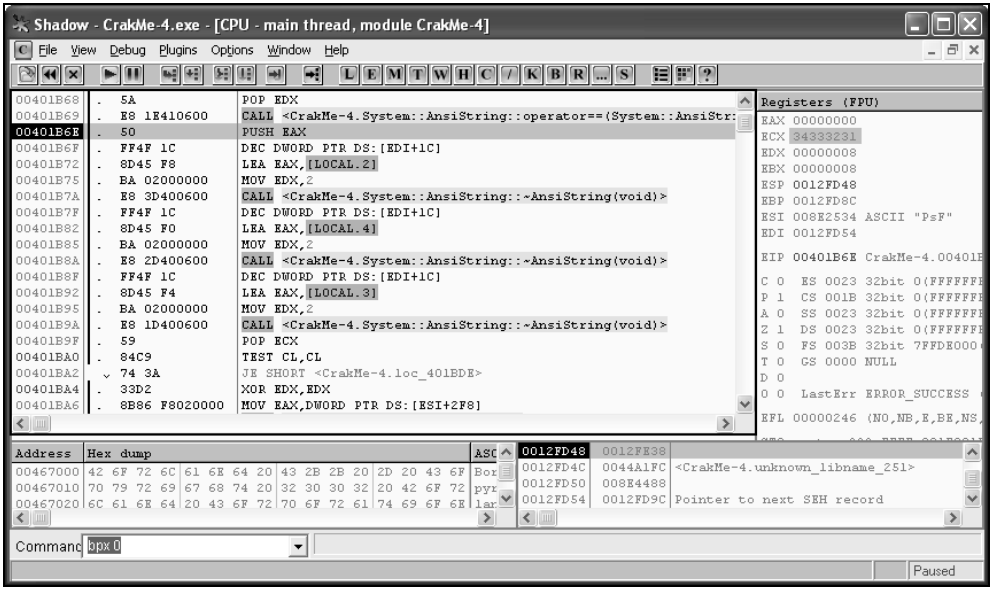

Рис. 5.44. Поиск функции сравнения в файле

Теперь при вводе любого восьмисимвольного пароля наша программа будет считать его верным. Как убрать ограничение на восемь символов, вы уже догадались.

Глава 6

Основные способы защиты программ

В этой главе описаны распространенные способы защиты программ:

- проверка даты и времени;
- инструмент InqSoft Sign Of Misery;
- фиксированное количество запусков;
- CRC

и др.

Проверка даты и времени

Сейчас мы рассмотрим такой способ защиты программ, как проверка даты (TimeLimit) или времени (программы "с проверкой времени" практически не встречаются, но когда-то этот вариант защиты был очень популярен). Как бы то ни было, идея и у тех и у других достаточно проста: при запуске программа проверяет текущую дату (это может быть год, месяц, день, час), и если она не совпадает с запланированной, то программа отказывается работать или работает неправильно. Такие защиты имеются у так называемых триал-программ, т. е. у программ, использование которых ограничено по времени.

Для взлома такой программы остается перехватить в программе проверку даты (времени) и установить требуемую дату (время) — и программа запустится.

Это была краткая теория, без которой трудно снимать такие защиты. Теперь давайте займемся практикой.

Практический пример

Запустив программу CrakMe-5, мы увидим сообщение об ошибочной дате (рис. 6.1).

Рис. 6.1. Окно сообщения об ошибочной дате

Проверим, не зашифрована или не запакована ли наша программа. Запускаем PEiD и смотрим (рис. 6.2).

Рис. 6.2. Результаты анализа файла, выполненного программой PEiD

Как мы видим, наша программа ничем не зашифрована и не запакована. Она написана на языке программирования Borland C++. Это нам очень пригодится, т. к. я уже говорил, нужно как можно больше знать об исследуемой программе.

Загружаем программу в отладчик OllyDbg. (Надеюсь, вы не забыли создать MAP-файл программы с помощью IDA?)

Теперь надо поставить брекпойнт, но какой? Тут нам как раз поможет окно сообщения, показанное на рис. 6.1. Что там написано? "Date is Error!" Ага, значит "Date" (дата) — это может быть и месяц, и год, и день. Но в языке C++ все это запрашивается одной функцией:

```
getdate ();
```

Примечание

Если вы не знаете, какая функция запрашивает дату или время, или программа написана на неизвестном вам языке программирования, то вам придется открыть справочник по языку и найти нужную функцию там.

Давайте поставим брекпойнт на эту команду. Для этого выведем имена всех модулей, используемых программой (табл. 6.1), и найдем среди них функцию чтения даты (как это делать, я говорил в предыдущих главах).

Таблица 6.1. *Модули, используемые программой*

Address	Module	Section	Type	Name
7C8660C4	kernel32	.text	User	GetCursorPos
77C1F785	msvcrt	.text	Library	Getcwd
77C1F785	msvcrt	.text	Export	_getcwd
0045AB9C	CrakMe-5	.text	User	_getdate
771212BC	OLEAUT32	.text	Import	KERNEL32.GetDateFormatA
77C11240	msvcrt	.text	Import	KERNEL32.GetDateFormatA
77D6F504	USER32	.text	User	GetDateFormatA
77DA5C3E	USER32	.rsrc	User	GetDateFormatA
7C826E0C	kernel32	.text	Library	GetDateFormatA
7C826E0C	kernel32	.text	Export	GetDateFormatA
7C865C3E	kernel32	.text	User	GetDateFormatA
5D091298	COMCTL32	.text	Import	KERNEL32.GetDateFormatW
771212C4	OLEAUT32	.text	Import	KERNEL32.GetDateFormatW
7C827C79	kernel32	.text	Library	GetDateFormatW
7C827C79	kernel32	.text	Export	GetDateFormatW
77C48450	msvcrt	.text	Export	_Getdays
77C48450	msvcrt	.text	Library	Getdays

Как видим, наша программа использует только одну функцию:

```
0045AB9C   CrakMe-5   .text   User   _getdate
```

Ставим на нее брекпойнт и запускаем программу. Мы попадаем внутрь функции проверки даты (рис. 6.3).

Трассируем ее с помощью клавиши <F8> до строки

```
0045ABC4   5D   POP EBP    0012FED0
```

Регистр EAX сейчас содержит 000007D6. Это значение в шестнадцатеричном формате; переведем его в десятичный и получим 2006. На что это похоже? Правильно, это год.

Теперь продолжим трассировать программу до строки, показанной на рис. 6.4.

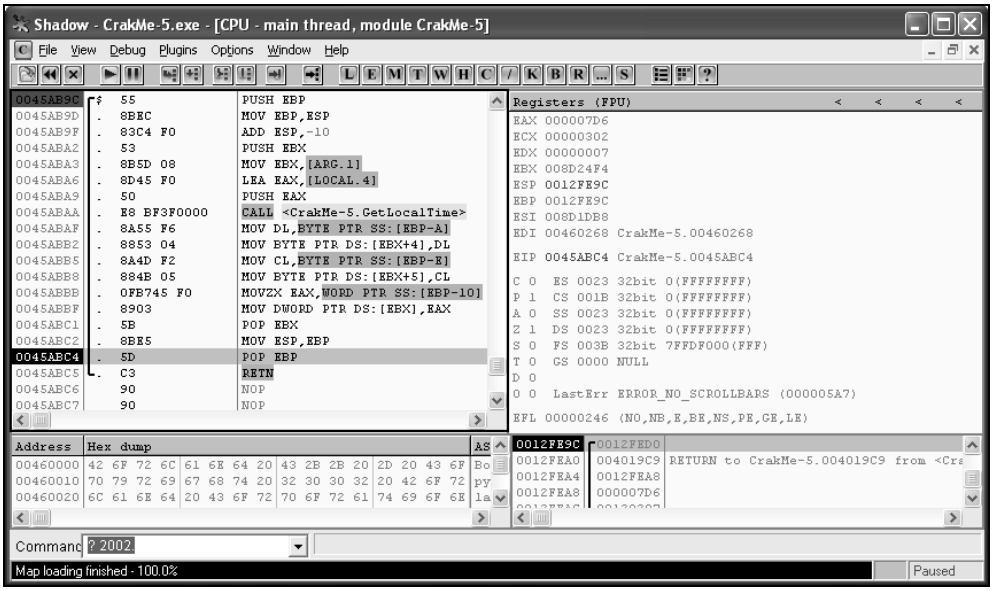

Рис. 6.3. Вид окна отладчика при срабатывании брекпойнта

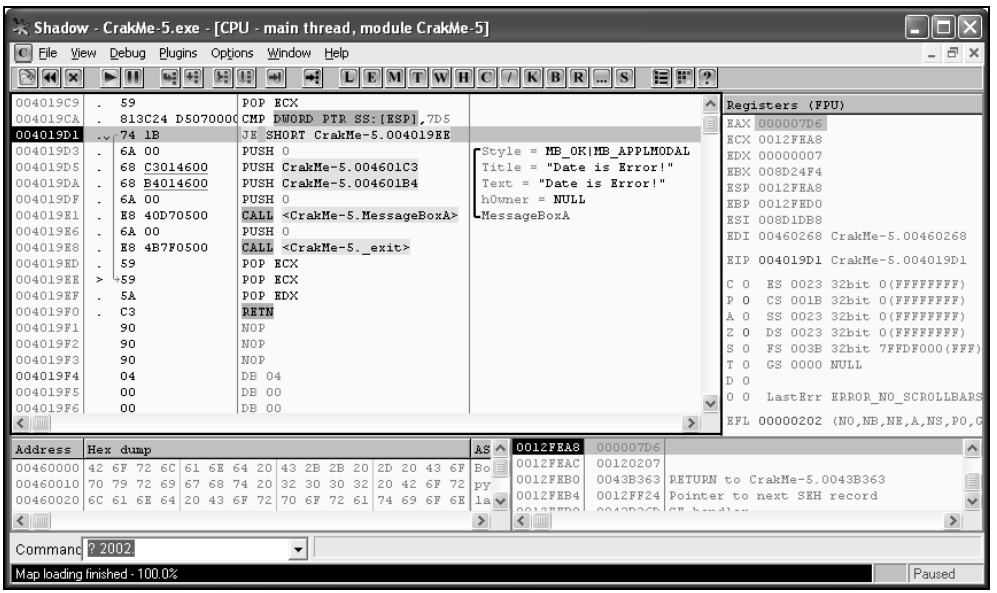

Рис. 6.4. Реализация проверки даты

Мы видим, что в строке с адресом 004019D1 выполняется сравнение с 7D5 (шестнадцатеричный формат). Если перевести значение 7D5 в десятичный формат, то получим 2005. Значит, программа проверяет текущее значение года, и если оно превышает 2005, то программа не работает. Изменим

```
004019D1   74 1B    JE SHORT CrakMe-5.004019EE
```

на

```
004019D1   EB 1B    JMP SHORT CrakMe-5.004019EE
```

Запускаем, и — программа работает вне зависимости от текущей даты (рис. 6.5).

Рис. 6.5. Окно сообщения о верной дате

Как видите, здесь все оказалось даже легче, чем поиск пароля в программе. Это объясняется тем, что прочитать системную дату можно очень ограниченным числом способов. Придумать тут что-либо новое довольно сложно.

Еще можно, например, считывать дату создания защищающего файла и сравнивать ее с датой доступа к этому же файлу. Но и этим крекера не остановить.

Со временем вы и сами научитесь видеть, на чем основана защита той или иной программы. Это все приходит с практикой. Чем больше вы будете практиковаться, тем быстрее все освоите.

Инструмент InqSoft Sign Of Misery

Для защиты программ с помощью проверки даты и времени есть специальный программный инструмент InqSoft Sign Of Misery (автор CyberManiac), созданный главным образом для продления "жизни" shareware-программ. Эта программа вобрала в себя большое количество приемов борьбы с ограничениями времени пользования программой, подавления всплывающих окон (NAG-screen), мешающих нормальной работе с программой, изменения и удаления "лишних" файлов и ключей реестра, а также множество других техник. Кроме того, программа содержит удобный и эффективный мастер создания патчей, при помощи которого можно легко создавать пачти и русификаторы к программам. Фактически InqSoft Sign Of Misery позволяет рядовому пользователю, не обладающему какими-либо специальными знаниями, эффективно бороться с защитами типа TimeLimit (ограничение по времени

использования), всплывающими окнами и встроенной в ПО баннерной рекламой. В отличие от других программ подобного рода, программе InqSoft Sign Of Misery не требуется присутствовать в памяти во время запуска и исполнения выбранной программы. Все необходимые функции реализуются при помощи создаваемого пользователем скрипта, который в дальнейшем может быть откомпилирован в обычный исполняемый файл. Конечно, это хорошая программа, но она универсальна, а значит, не дает 100%-ной гарантии того, что она сработает. К тому же, авторы защит постоянно совершенствуют свои произведения, улучшают, изменяют их. Это надо хорошо понимать. Если бы такие программы гарантировали успех на 100%, то никакие крекеры бы не понадобились. К счастью (и для программистов, и для крекеров), это не так. Тем не менее, каждому крекеру полезно иметь эту программу у себя на компьютере.

Фиксированное количество запусков программы

Следующий способ защиты программ — создание Trial-программ, т. е. программ, запускаемых фиксированное число раз.

Защитный механизм таких программ реализуется с некоторыми отличиями:

- программа записывает количество запусков в реестр при установке на компьютер пользователя (при запуске инсталлятора);
- программа записывает количество запусков в реестр при первом запуске;
- программа записывает количество запусков в ключевой файл при первом запуске;

> **Примечание**
>
> *Ключевой файл* — это специальный файл программы, который она старается "спрятать" в недрах Windows или, если файл хранится в той же папке, что и программа, делает в него записи в зашифрованном виде.

- программа записывает количество запусков в ключевой файл при установке на компьютер пользователя (при запуске инсталлятора);
- программа запускается и работает только определенное число дней.

Это наиболее распространенные способы. Конечно, каждый автор может придумать свой, но конечный результат все равно будет одним из перечисленных. Просто программе негде больше хранить счетчик запусков.

Последний пункт надо пояснить. Это еще один вид реализация защиты "с проверкой даты" (TimeLimit). Дело в том, что проверять число прошедших дней можно разными способами.

- Записать дату первого запуска программы в реестр или ключевой файл, а потом сверять ее с текущей. Это самый простой, но и абсолютно неэффективный способ. Для взлома такой программы достаточно просто отследить такую запись и подправить ее текущей датой.
- Сверять текущую дату с датой создания самóго запускаемого файла. Этот способ посложнее, он до сих пор применяется в некоторых программах. Встречаются небольшие вариации данного способа, а именно — считывание даты ключевого файла или DLL-библиотеки, которые помещаются в папку WINDOWS или WINDOWS\SYSTEM. Все это тоже очень легко найти и исправить.

Для того чтобы затруднить поиск такой защиты, рекомендуется считывать дату файла с помощью какой-нибудь нестандартной функции и делать это не сразу. Лучше считывать дату файла именно тогда, когда выполняется считывание множества парамеров других файлов (например, из папки WINDOWS), чтобы замусорить и усложнить, таким образом, поиск защиты.

Давайте посмотрим на примере, как можно обмануть подобную защиту. Программа CrakMe-6 рассчитана на 10 запусков, о чем она и предупреждает нас, отображая количество оставшихся разрешенных запусков (рис. 6.6).

Рис. 6.6. Главное окно программы

Давайте проанализируем программу на предмет паковки или шифрования.

Примечание

Снова подчеркиваю: для того чтобы убедиться, что программа ничем не запакована и не зашифрована, необходимо использовать несколько разных анализаторов, да и то это не даст 100%-ной гарантии.

Итак, запускаем программу-анализатор PEiD и смотрим, что она нам сообщит (рис. 6.7).

Итак, мы видим, что программа написана на Borland C++ 1999. Значит, программа ничем не запакована и не зашифрована.

Рис. 6.7. Результаты анализа программы, выполненного программой PEiD

Раз программа показывает нам количество запусков и это количество уменьшается при каждом запуске программы, то тут вывод напрашивается сам собой: это защита типа "запуск фиксированное число раз". Причем у программы есть счетчик, который она уменьшает при каждом запуске. Счетчик может находиться или в ключевом файле, или в реестре. Давайте проверим сначала реестр как самое распространенное решение.

Для осуществления такой проверки у нас теперь есть два варианта: очень простой и простой. Первый вариант — это использование специальных мониторов для отслеживания изменений в реестре. Второй вариант — использование отладчика для поиска в программе такой защиты. Хотя второй вариант ненамного сложнее и является наиболее правильным, мы для начала остановимся на первом, т. к. с отладчиком вы уже работали, а с мониторами еще не пробовали. Поэтому я покажу вам, как пользоваться специальными программами, называемыми *мониторами*. Наиболее популярны следующие программные мониторы:

- Filemon for Windows NT/2000/9x — монитор файлов;
- Regmon for Windows NT/9x — монитор реестра;
- RegSnap: Registry analyzing tool — монитор изменений реестра и ключевых файлов.

Крекеры часто используют их для отслеживания ключей программ при защите программ с фиксированным числом запусков.

Итак, давайте пойдем по простому пути. Запускаем программу RegSnap (рис. 6.8).

Нажатием кнопки **New Snap** (Сделать снимок) откроем окно, позволяющее сохранить снимок — записать его в специальный файл (рис. 6.9).

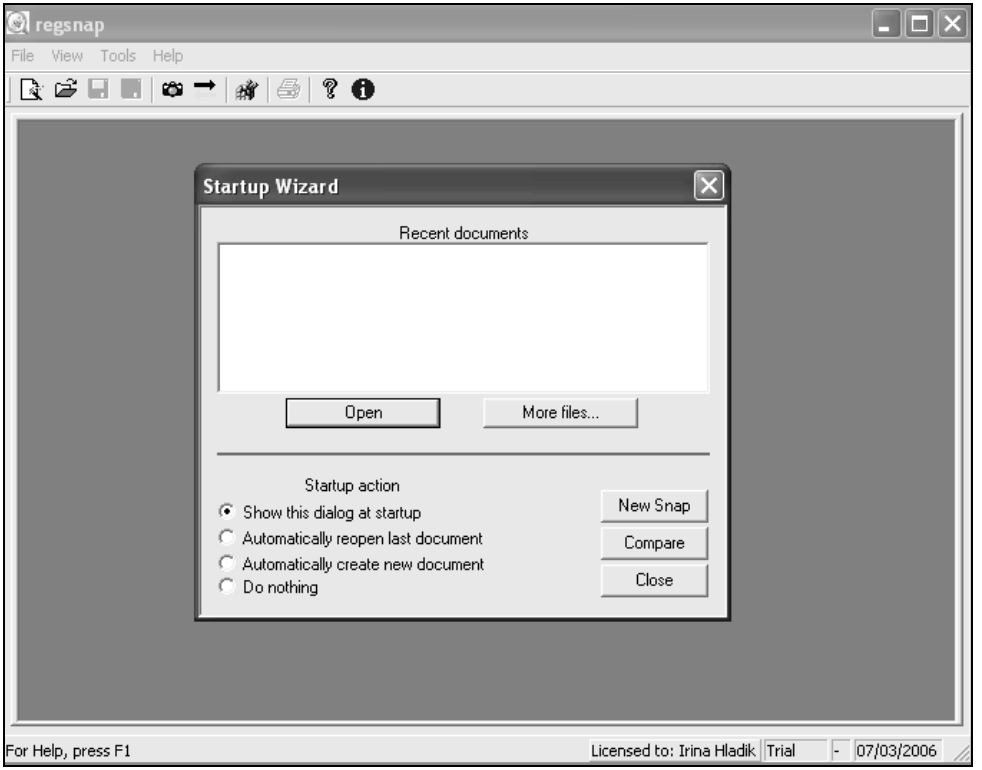

Рис. 6.8. Главное окно программы RegSnap

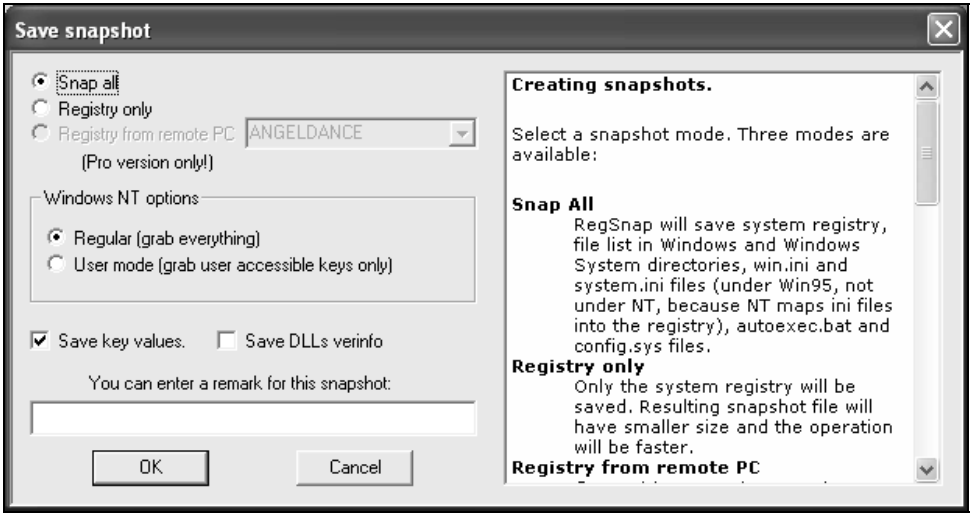

Рис. 6.9. Создание снимка в программе RegSnap

Примечание

Снимок — копия состояния всех ключей реестра и файлов библиотек.

Тут нам надо подумать, что мы будем отслеживать. Есть два варианта:

- только реестр;
- и реестр, и файлы библиотек Windows.

Так как мы решили для начала проверить только реестр, выбираем переключатель **Registry only** (Только реестр) и нажимаем кнопку **OK**.

Теперь необходимо дождаться, пока программа сделает снимок (программа сообщит нам, сколько записей реестра было скопировано).

Запускаем программу CrakMe-6 снова. Программа покажет, что счетчик уменьшился еще на единицу. Затем мы опять запускаем RegSnap и делаем еще один точно такой же снимок. После того как программа сделает второй снимок, мы сравним два снимка. Для этого нажимаем кнопку **Compare** (Сравнить).

Программа сравнит наши два снимка и выведет результат на экран. Нас интересует момент изменения реестра. Переходим к измененным ключам (modified keys) и видим очень интересный ключик:

```
Modified keys
HKEY_LOCAL_MACHINE\SOFTWARE\Crack7\Crack7
   Old value: DWORD: 8 (0x8)
   New value: DWORD: 7 (0x7)
```

Программа изменила значение ключа, т. е. просто уменьшила значение счетчика на единицу.

Давайте попробуем заменить в реестре это значение другим. Запускаем редактор реестра, переходим к этому разделу и вместо значения "7" запишем "10".

Запустим программу CrakMe-6 еще раз. Что мы видим? Программа опять готова к 10 запускам! Теперь для того, чтобы заставить программу запускаться бесконечное число раз, необходимо всего лишь сделать для нашей программы небольшой загрузчик, который будет:

1. Удалять или изменять значение реестра.
2. Запускать нашу программу CrakMe-6.

Теперь давайте рассмотрим способ обхода такой защиты с помощью отладчика OllyDbg (если что-то будет непонятно, пишите, я помогу).

Начнем эксперимент с того, что удалим ветвь реестра HKEY_LOCAL_MACHINE\SOFTWARE\Crack7\Crack7.

Как обычно, нужно постараться узнать, на каком языке написана программа и не зашифрована ли она.

Примечание

Я еще раз хочу подчеркнуть: ни в коем случае не полагайтесь на результаты анализа, выполненного только одной программой-анализатором.

В качестве альтернативы PEiD я покажу результаты анализа программы, выполненного с помощью такой прекрасной утилиты, как PE Tools (автор NEOx). Эта утилита позволяет делать множество вещей, которые разбиты по группам:

- Task Viewer:
  - дампы процессов:
    - Dump Full;
    - Dump Partial;
    - Dump Region;
  - возможность делать дампы .NET CLR процессов;
  - автоматическое снятие защиты Anti Dump Protection;
  - изменение приоритета процесса;
  - завершение работы процесса;
  - загрузка процесса в PE Editor и PE Sniffer;
  - Generic OEP Finder;
- PE Sniffer:
  - определение типа компилятора/упаковщика;
  - возможность обновления базы сигнатур;
  - возможность сканирования папок;
- PE Rebuilder:
  - оптимизация PE-файла;
  - изменение базового адреса PE-файла;
- PE Editor:
  - редактирование DOS-заголовка;
  - поддержка нового формата PE+ (64-разрядного);
  - корректирование CRC;
  - просмотр и редактирование таблиц импорта/экспорта.

В данном случае нам понадобится PE Sniffer. Но в дальнейшем вам пригодятся и остальные части утилиты.

Запускаем утилиту PE Tools, с помощью мыши перетаскиваем исследуемый файл в главное окно утилиты и смотрим на результат (рис. 6.10).

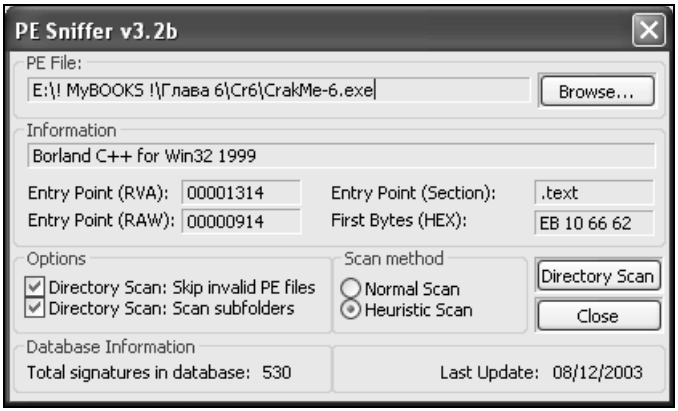

Рис. 6.10. Результаты анализа программы, выполненного утилитой PE Sniffer

Как мы видим, программа не зашифрована, не запакована и написана на языке Borland C++ for Win32 1999. Что же, очень хорошо. Запускаем отладчик OllyDBg и загружаем в него исследуемую программу (CrakMe-6).

Теперь давайте подумаем. Предыдущее исследование показало, что программа изменяет значение в реестре. Значит, надо установить брекпойнт на функцию доступа к реестру. Теперь осталось выбрать функцию, на вызов которой мы его установим (кто не знает или сомневается, загляните в справочник по функциям Builder C++). Функций для работы с реестром в Builder C++ не так уж много, все они находятся в классе `TRegistry`. Перечислим основные:

- `OpenKey(System::AnsiString,bool);`
- `ReadInteger(System::AnsiString);`
- `ReadString(System::AnsiString);`

Можно, конечно, установить брекпойнты на все функции, но это дело вкуса. Я предпочитаю поступать следующим образом. В окне OllyDbg я нажимаю комбинацию клавиш <Ctrl>+<N> и получаю список всех используемых в программе функций. Их очень много, но, к счастью, нужны только функции для работы с реестром. Для быстрого поиска просто набираю на клавиатуре "Regi" и получаю список, показанный на рис. 6.11.

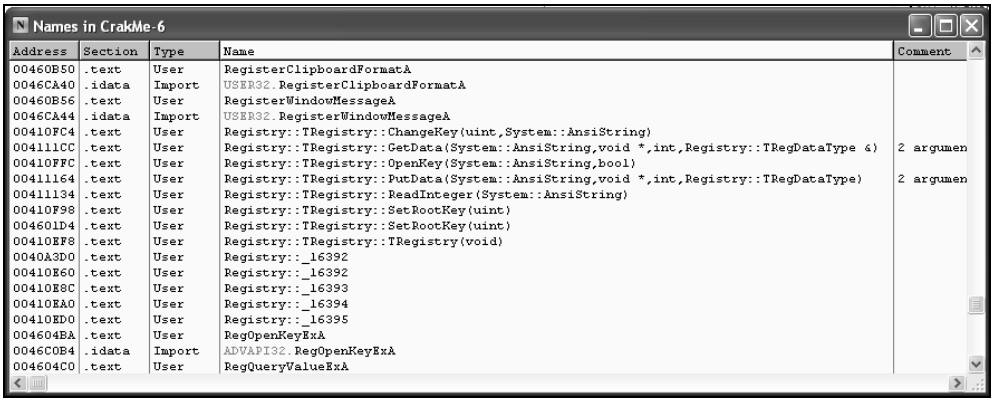

Рис. 6.11. Поиск функций в программе с помощью OllyDbg

Среди этих функций нет функции считывания строк из реестра `ReadString`, но зато мы видим функцию считывания из реестра целых значений:

```
Registry::TRegistry::ReadInteger(System::AnsiString)
```

Из этого следует, что наша исследуемая программа считывает из реестра числовые значения. Устанавливаем на вызов этой функции брекпойнт и запускаем программу. При срабатывании брекпойнта мы попали в функцию (рис. 6.12).

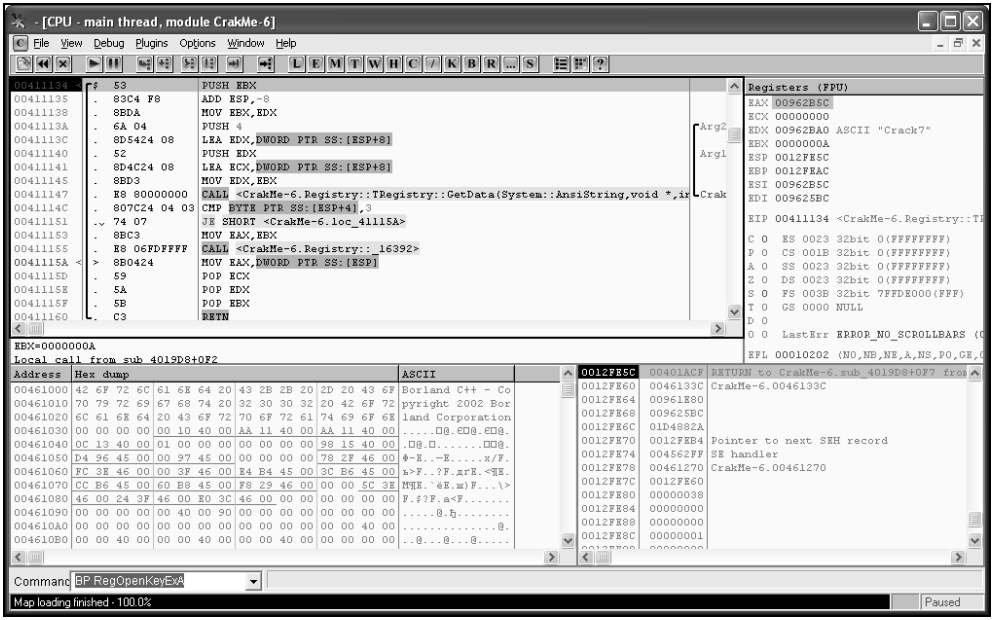

Рис. 6.12. Вид окна OllyDbg при срабатывании брекпойнта

Поскольку мы находимся внутри функции, из нее надо вернуться в то место, откуда она была вызвана. Нажимаем комбинацию клавиш <Ctrl>+<F9>, чтобы дойти до конца функции. Нажатие клавиши <F8> возвратит нас в место, откуда была вызвана функция чтения числовых значений реестра (рис. 6.13).

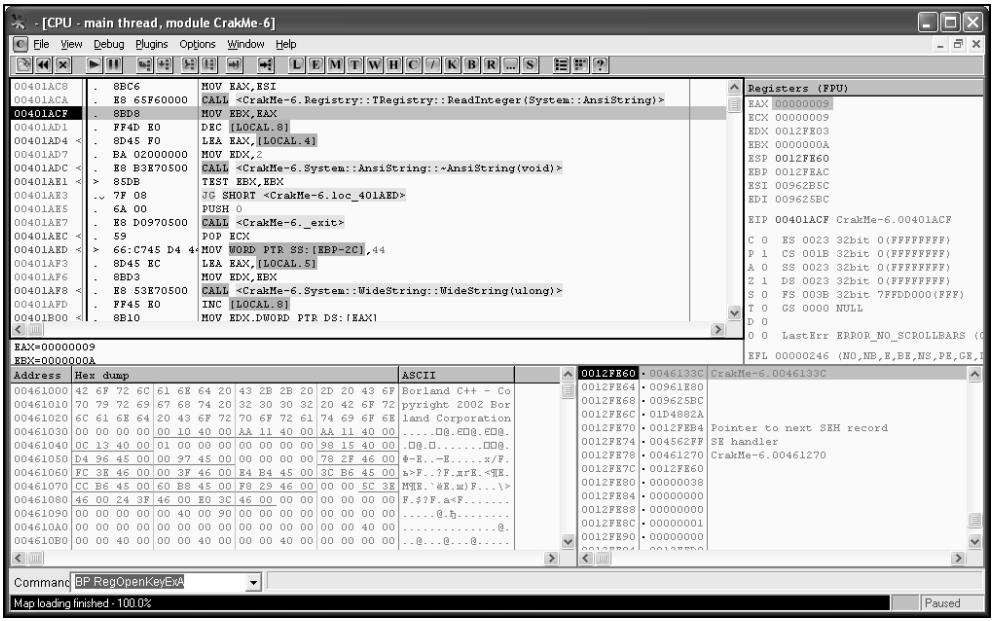

Рис. 6.13. Место вызова функции, на который установлен брекпойнт

Здесь мы видим, что функция вернула прочитанное из реестра значение в регистре EAX (см. рис. 6.13).

Давайте запомним этот адрес: 00401ACFh. Удалим все брекпойнты, установленные нами ранее, поставим брекпойнт только на этот адрес и перезапустим программу. Мы видим, что значение в регистре EAX уменьшилось на единицу. Значит, в этом месте происходит считывание данных из реестра. Как только значение счетчика уменьшится до нуля, программа откажется запускаться. Раз так, то где-то дальше в программе должно выполняться сравнение. Чуть ниже находим строку сравнения:

```
jg    short <CrakMe-6.loc_401AED>
```

Для того чтобы обойти эту проверку, можно заменить в ней команду `jg` командой безусловного перехода `jmp`, и тогда наша программа будет работать все время. Для этого, выделив с помощью мыши команду

```
jg    short <CrakMe-6.loc_401AED>
```

нажмем клавишу <Пробел> и введем

```
jmp  short <CrakMe-6.loc_401AED>
```

Затем в контекстном меню выберем команду **Copy to executable > All modifications** и затем команду **Copy all**. Появится окно (рис. 6.14), при закрытии которого отладчик спросит нас, нужно ли сохранять сделанные изменения. При утвердительном ответе нам предложат выбрать место, где будет сохранен измененный файл.

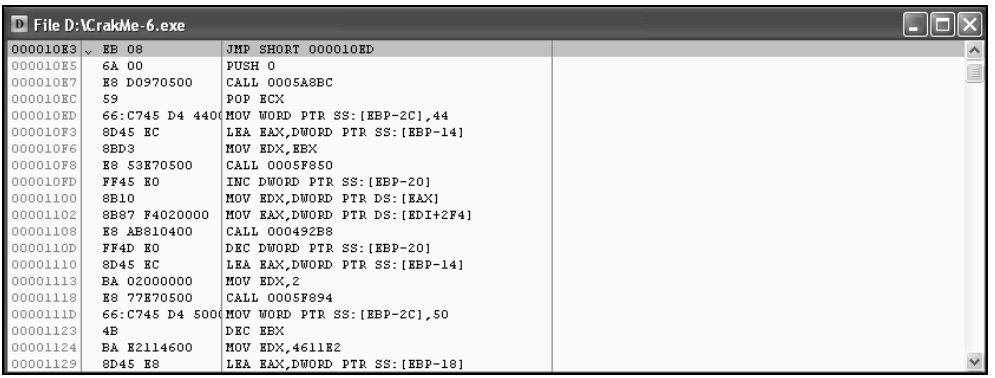

Рис. 6.14. Окно, показывающее наши изменения

Вот и все, программа изменена. Как видите, это ненамного сложнее первого способа, и я, надеюсь, наглядно показал вам, почему крекеры стараются автоматизировать любой процесс, какой только возможно.

Программы, защищенные CRC

Ну, вот мы и добрались до такой интересной темы, как CRC. Сейчас CRC достаточно часто используется в разных областях, но нас интересует только защита программного обеспечения (ПО).

Начнем с определения, т. к. частенько происходит путаница в терминах. Итак, CRC (Cyclic Redundancy Code) — циклический избыточный код. Сейчас очень многие программы защищены таким видом защиты. Это очень правильно, поскольку CRC не только дает дополнительную защиту от крекеров, но также позволяет защитить программы от компьютерных вирусов. Например, если программа была изменена или заражена, то контрольная сумма изменится; на основании этого можно выдать сообщение или вообще завершить работу программы.

Думаю, вы попадали в ситуацию, когда присланный по почте или скачанный из Интернета архив не распаковывается и сообщает о расхождении контрольных сумм. Архиватор, таким образом, подстраховывается, чтобы не подвести пользователя и обеспечить корректную распаковку. Как архиватор определяет целостность файла? Все просто: архиватор сканирует файл по заранее выбранному алгоритму и сравнивает полученную контрольную сумму со значением CRC из файла, и если они различны, выдает сообщение об ошибке. В основном архиваторы используют алгоритм CRC-32, ставший популярным из-за высокого быстродействия и низкой вероятности ложного срабатывания (2^{-32}). Конечно, алгоритмов нахождения контрольных сумм великое множество! Каждый волен придумать такой алгоритм сам, но разработано несколько стандартных алгоритмов CRC, например, CRC, CRC-16, CRC-32, прямая реализация CRC, MD5, SHA и т. д.

Итак, контрольные суммы применяются в следующих целях:

- защита от заражения вирусами;
- защита от изменения кода в памяти или на жестком диске;
- противодействие распаковке (применяется "навесными" протекторами AsProtect, Obsidium, SVKP).

Проверка CRC — очень хороший способ насолить крекеру. Если не верите, попробуйте, не читая дальше, "взломать" программу Crackme_3final (автор nice), которая есть на сопроводительном диске книги. Задача — поменять строку

```
CRC Test №1 DEMO
```

на

```
CRC Test №1 PRO
```

Дам небольшую подсказку: изменить можно константу

```
0040102F   . >CMP EAX, AC34CFDB
```

и вот эти 4 байта:

```
00401038   . 1E           push DS
00401039   . 8022 5F      and [BYTE DS:EDX], 5F
```

Удачи!

Листинг 6.1 содержит полный код данной программы типа CrackMe.

Листинг 6.1. Программа Crackme_3final от nice для проверки возможностей CRC

```
; Пример crc №3
format PE GUI 4.0
; Подключаем файл с макросами
include 'D:\AngelDance\MyFaSM\INCLUDE\win32a.inc'
```

```
.crc1:
      mov       EAX,.crc1
      mov       ECX, 40h    ; 100h/4, т.к. цикл идет с шагом 4
      call      crcCheck
      mov       dword [intZN], EAX   ; Заносим CRC в ячейку памяти
      call      int2str     ; Функция-преобразователь из int в string
                            ; для корректного отображения на экране
      push      EAX
      invoke    MessageBoxA, 0, strZN, str2, 64    ; Вывод сообщения
; Ключевая проверка EAX с CRC
      pop       EAX
      cmp       EAX, 0AFAAF9BAh
      jnz       .go      ; Переход, если CRC не совпадает
      jmp       .next
; 52C46FFB
      db        0FBh, 06Fh, 0C4h, 52h
.next:
      invoke    MessageBoxA, 0, str1, str2, 64      ; Вывод сообщения
      jmp      .goexit
.go:
      invoke    MessageBoxA, 0, CRCError, str2, 64  ; Вывод сообщения
.goexit:
      invoke    ExitProcess, 0                   ; Выход из программы

proc  crcCheck      ; Вычисляем CRC, на входе помещаем
                    ; в EAX указатель на блок данных,
                    ; ECX = длина блока
; Блок проверки контрольной суммы
      mov       EDI, EAX           ; Указатель начала кода
      mov       EBX, 0B239ADAFh    ; Произвольная константа
.test_crc:
      mov       EAX, [EDI]         ; Загружаем 4 байта в EAX
      xor       EAX, EBX           ; Исключающее ИЛИ
      shl       EBX, 1
      xor       EBX, EAX
      add       EDI, 4
      loop      .test_crc          ; Продолжаем, пока ECX<>0
endp
```

```
proc int2str    ; Процедура переводит число в строку:
                ; 01 -> 31 ... 09 -> 39
                ; 0A -> 41 ... 0F -> 46
      push      EAX
      mov       ESI, strZN
      mov       ECX, 4
.cicl:
      mov       EBX, hexx
      mov       AL, byte [intZN+ECX-1]
      mov       DL, AL
      and       AL, 0f0h
      shr       AL, 4
      xlatb               ; AL=hexx[Al]
      mov       byte [ESI], AL
      inc       ESI
      mov       Al, Dl
      and       AL, 0fh
      xlatb
      mov       byte [ESI], AL
      inc       ESI
      loop      .cicl
      pop       EAX
      ret
endp

hexx     db '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B',
  'C', 'D', 'E', 'F'
intZN    db 04h dup (00h)
strZN    db 09h dup (00h)
str1     db 'CRC Test №1 DEMO', 0
str2     db 'Created by nice', 0
CRCError db 'CRC ERROR! File infected or corrupted!', 0

data import

 library  kernel32, 'KERNEL32.DLL',\
          user32, 'USER32.DLL'

 import kernel32, \
```

```
       ExitProcess, 'ExitProcess'

 import user32, \
       MessageBoxA, 'MessageBoxA'

end data
```

Что, не получилось? Убедились, что CRC — вещь хорошая? Давайте теперь изучим все подробнее и на практике.

Для начала рассмотрим простой вариант. Возьмем строку из пяти байт:

90 90 44 41 13

Высчиляем CRC по формуле:

$CRC = a_1 + a_2 + ... + a_{n-1} + a_n$

Тогда

CRC = 90 + 90 + 44 + 41 + 13 = 278

Для шестнадцатеричной строки 1Ah 12h 84h 07h:

CRC = 1Ah + 12h + 84h + 07h = B7h

Как видите, ничего сложного в этом нет. В таком виде алгоритм хотя и чрезвычайно прост, но имеет массу недостатков: например, мы можем поменять элементы местами и получить тот же самый результат, т. е. не сможем определить, поврежден файл или нет.

Мой опыт показывает, что обычно в программах используют либо популярный алгоритм нахождения контрольной суммы (CRC-32, MD5, SHA), либо простой, придуманный автором программы и использующий логические операции (`xor`, `or`, `and`, `shl`, `not` и т. д.).

Давайте рассмотрим это все на простом примере (листинг 6.2).

Листинг 6.2. Программа без CRC-защиты

```
; Выводим на экран сообщение
      invoke    MessageBoxA, 0, str1, str2, 64
; Завершаем работу программы
      invoke    ExitProcess,0

str1      db 'CRC Test №1. This is DEMO!', 0
str2      db 'Created by AngelDance', 0
```

Это была простая программа без проверки CRC. Затем автор решил подстраховаться и, чтобы программу не взломали, добавил в нее проверку CRC (листинг 6.3).

Листинг 6.3. Программа с CRC-защитой

```
crc:
; Длина проверяемого блока
      mov       ECX, 26h
; Указатель на начало кода 00401000
      mov       EDI, crc
; Произвольная константа
      mov       EBX, 0B239ADAFh
test_crc:
; Данные команды ничего не делают,
; я поставил их специально
; для проведения эксперимента
      nop
      nop
      nop
      nop
; Загружаем 4 байта в EAX
      mov       EAX, [EDI]
; Исключающее ИЛИ
      xor       EAX, EBX
      shl       EBX, 1
      xor       EBX, EAX
      inc       EDI
; Продолжаем, пока ECX<>0
      loop      test_crc
```

Давайте посмотрим наше значение CRC. Берем программу CRC-1 с сопроводительного диска книги. Запускаем отладчик OllyDBg и останавливаемся на адресе 00401028 (это вы уже умеете). Почему именно на этом адресе? А потому, что тут происходит проверка контрольной суммы для 26 байт, как это видно в окне отладчика.

```
00401028 3D AC7C788A   cmp EAX, 8A787CAC
0040102D 75 16         jnz short CRC-1.00401045
```

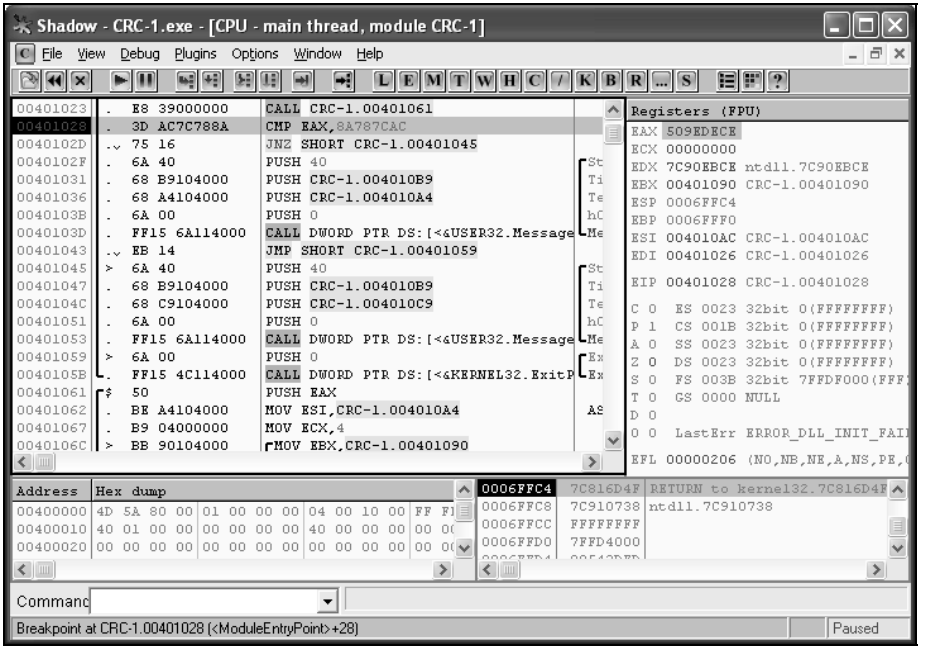

Рис. 6.15. Окно отладчика OllyDBg с нашей программой

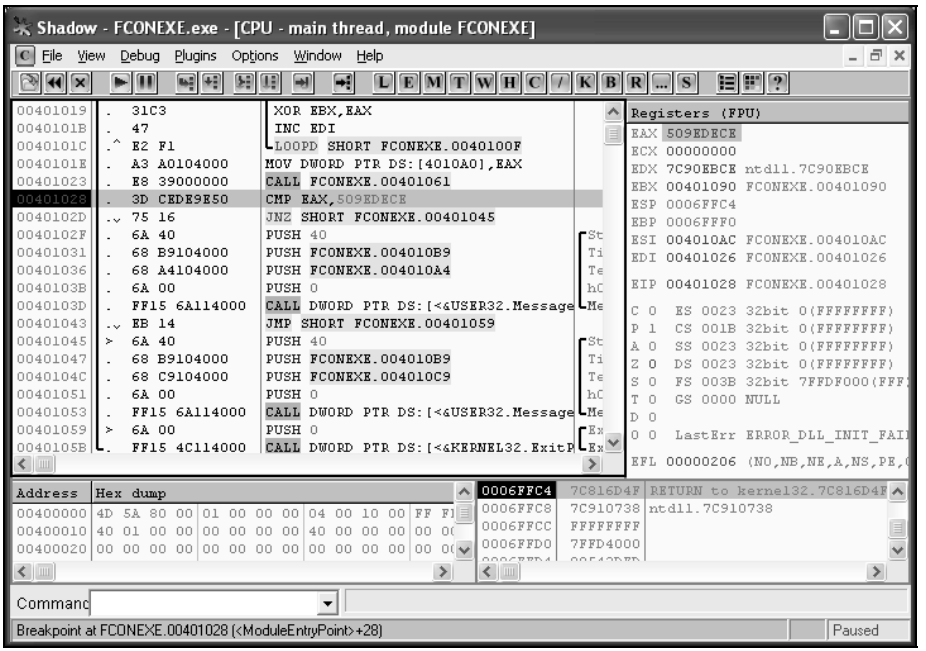

Рис. 6.16. Программа с исправленной контрольной суммой

Теперь посмотрим, какое значение содержит EAX (рис. 6.15).

На рисунке видно, что EAX = 509EDECEh, и если контрольная сумма не совпадает, то выполняется переход на CRC-1.00401045.

Исправив в исходном тексте контрольную сумму и откомпилировав программу заново (см. программу CRC-1.1 на сопроводительном диске книги), мы получим вот такой результат (рис. 6.16).

Как видите, если запустить программу, то проверку на CRC она у нас пройдет.

Поставим эксперимент: "взломаем" программу, изменив байты по адресу 0040100F. Для этого запустим любой шестнадцатеричный редактор и вместо 90 90 запишем 40 48. Ставим брекпойнт на адрес 00401028, нажимаем клавишу <F9> и смотрим: регистр EAX содержит число 06F51F84h.

```
00401028   3D AC7C788A       cmp EAX, 8A787CAC
0040102D   75 16             jnz short crc_1.00401045
```

Нажимаем клавишу <F9> еще раз и получаем сообщение: файл поврежден! Достаточно было исправить 2 байта, и значение CRC изменилось до неузнаваемости! Такое поведение и объясняет популярность подобных защит.

Давайте усложним алгоритм: вынесем функцию вычисления CRC в отдельную процедуру и добавим вывод контрольной сумы на экран для наглядности (см. файл CRC-2.ASM на сопроводительном диске книги):

```
; Указатель на блок данных
mov       EAX, crc1
; Длина блока
mov       ECX, 10h
; Вызов вычислительной функции; возвращает CRC в EAX
call      crcCheck
...
crc1:
str1       db 'CRC Test №1', 0
str2       db 'Created by AngelDance', 0
CRCError   db 'CRC ERROR! File infected or corrupted!', 0
```

Примечание

Функция специально упрощена, нет защиты регистров, алгоритм достаточно простой, при желании вы можете его доработать.

Итак, посчитаем контрольную сумму наших строк. Запускаем программу и получаем сообщение (рис. 6.17).

Рис. 6.17. Вывод контрольной суммы

Подставим эту константу в проверку:

```
      push       EAX
; Вывод сообщения
      invoke     MessageBoxA, 0, strZN, str2, 64
; Ключевая проверка EAX с CRC
      pop        EAX
      cmp        EAX, 0EEB4120Ch
; Переход, если CRC не совпадает
      jnz        go
```

Функция `MessageBox` возвращает в регистре EAX номер нажатой кнопки, поэтому мы восстанавливаем значение после вызова функции (`pop EAX`).

Ну что ж, теперь самое время приступить к взлому. Изменим строку `DEMO` на `PRO`. Для этого исправляем строку в шестнадцатеричном редакторе и сохраняем измененный файл программы.

Запускаем программу и получаем сообщение об ошибке контрольной суммы (рис. 6.18).

Рис. 6.18. Окно сообщения об ошибке контрольной суммы

Думаю, вы уже догадались, что, убрав проверку

```
0040102F  cmp EAX, EEB4120C
00401034  jnz SHORT crc_2.0040104C
```

т. е. заменив в строке

```
00401034  jnz SHORT crc_2.0040104C
```

команду `jnz` на `jmp`

```
00401034  jmp 401036
```

мы можем безнаказанно производить любые изменения в файле.

Более сложный алгоритм проверки CRC

Давайте усложним задачу. Представим, что нам нужно рассчитать контрольную сумму для всего файла. Мы можем создать многочисленные проверки в файле и делать "кусочные" проверки (рис. 6.19).

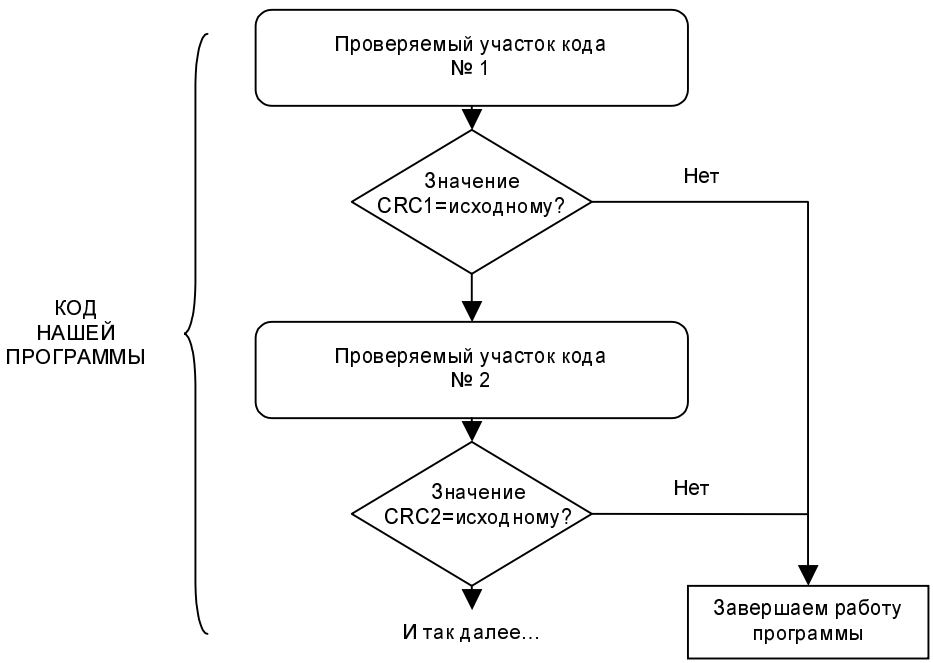

Рис. 6.19. Алгоритм "кусочных" проверок

У такого метода есть как плюсы, так и минусы. Я покажу вам способ расчета CRC для всего файла.

В чем сложность, спросите вы? Дело в том, что если мы рассчитаем контрольную сумму, а потом "забьем" ее как константу:

```
cmp   EAX, 012345678h
jz    .exitProcCrcError
```

то наша проверка всегда будет ложной, т. к. значение CRC (4 байта) изменено и повторный расчет даст другое значение. Получается замкнутый круг: CRC мы не можем вычислить, т. к. при малейшем изменении программы это значение изменится, и вычислить правильное мы не сможем.

А если перенести значение в конец файла? И делать проверку всего файла кроме четырех последних байт?

```
mov    EBX, [lenFile - 4]
cmp    EAX, EBX
```

Это рабочий способ, но, по сути, от первого он ничем не отличается.

Давайте посмотрим на наш код вычисления CRC (немного упрощенный):

```
; загружаем 4 байта в EAX
mov      EAX, [EDI]
; Исключающее ИЛИ
xor      EAX, EBX
 ; Умножаем на 2
shl      EBX, 1
; Исключающее ИЛИ
xor      EBX, EAX
; Увеличиваем счетчик
add      EDI, 4
; Продолжаем, пока ECX<>0
loop     .test_crc
```

Мы передвигаемся по файлу, смещаясь на 4 байта. В математической логике есть такая формула:

A xor $B = C$

Это обратимая операция:

C xor $B = A$

и

A xor $C = B$

Поработав с циклом, вы поймете, что старое значение EAX роли не играет, важны значения EAX (считанное из памяти) и EBX, причем значение EBX потребуется как до умножения, так и после.

Чтобы решить данную проблему, нужно добавить 4 корректирующих байта (рис. 6.20).

Рис. 6.20. Вставка четырех корректирующих байт

Теперь я объясню вам технологию расчета значения четырех корректирующих байт. Для начала вставим пустые (нулевые) байты и запустим программу. Она выдаст значение CRC (у меня оно получилось равным 0AFAAF9BAh).

```
cmp       EAX, 012345678h
; Переход, если CRC не совпадает
jnz       .go
; Этот переход я ввел специально.
; Если не сделать прыжок, то следующие байты
; вызовут исключительную ситуацию
; и в лучшем случае программа завершит свою работу
jmp      .next
; Пока зададим нули
db       0h, 0h, 0h, 0h
.next:
```

Запустим программу в отладчике. Сначала посмотрим, по какому адресу в памяти располагаются наши 4 байта:

```
00401038     0000   add [byte DS:EAX],AL
0040103A     0000   add [byte DS:EAX],AL
```

Перейдем в цикл подсчета CRC (нажимаем в отладчике OllyDBg комбинацию клавиш <Ctrl>+<G> и вводим значение 00401075):

```
00401075  |> 8B07   mov EAX, [dword DS:EDI]
```

Установим брекпойнт на этот адрес с условием (в EDI хранится указатель на считываемые данные):

```
EDI==00401038
```

Для этого нажмем клавиши <Shift>+<F2> и введем условие

```
EDI==401038.
```

Теперь нажимаем клавишу <F9> ("отпускаем" программу); выполнение приостановилось в теле цикла на адресе 00401075h, при этом EDI = 00401038h.

Наша задача — законспектировать значения регистра EBX (почему — я уже сказал) до и после умножения на 2. У меня они были такими:

	EIP	EBX
До:	00401075h	052DE2717h
После:	0040107Dh	0A840E927h

Теперь можно приступать к вычислению искомого 4-байтного значения. Исправляем CRC:

```
cmp       EAX, 0AFAAF9BAh
; Переход, если CRC не совпадает
jnz       .go
```

Перекомпилируем и запускаем программу, устанавливаем брекпойнт, как описано выше, и отпускаем программу с помощью клавиши <F9>. Вот что получилось у меня:

EIP = 00401075h, EBX = 0567C7DB4h

EBX shl 1 = 0ACF8FB68h

Нам нужно получить такое значение EAX, чтобы:

EAX xor (EBX shl 1) = 0A840E927h

Тогда:

$EBX_{\text{старое значение}}$ xor $EBX_{\text{новое значение}}$ = 0A840E927h^0ACF8FB68h = –04B8124Fh

(таким должно быть число в EAX перед последней операцией xor).

Следовательно:

EAX xor 0567C7DB4h = 004B8124Fh

EAX = 004B8124Fh xor 0567C7DB4h = 052C46FFBh — вот искомое число!

Запишем (не забываем, что данные в памяти отображаются в обратном порядке):

```
db     0FBh, 06Fh, 0C4h, 52h     ; 052C46FFBh
```

Мы определили искомые байты. Запускаем программу — CRC совпадает!

Примечание

Этот максимально упрощенный пример содержится в файле CRC_3.ASM на сопроводительном диске книги.

Использование данной техники в защитах значительно усложняет их взлом. А если использовать CRC как переменную, то взлом вообще превращается в головную боль. Удачи!

Классические алгоритмы нахождения CRC и методика их применения в защите ПО

Я уже упоминал некоторые из хеш-функций MD5, SHA, HAVAL и т. п. Это математически обоснованные алгоритмы, активно используемые программистами.

Что же такое хеш-функция? В идеале это необратимая функция, т. е. на вход подаем данные, получаем на выходе значение, а обратное вычисление невозможно.

Функция должна удовлетворять определенным условиям:

- стойкость к коллизиям (два различных набора данных должны иметь различные результаты преобразования);
- необратимость (невозможность вычислить исходные данные по результату преобразования);
- простота реализации;
- высокая скорость.

Это в идеале, на самом деле пока существование необратимой функции не доказано. Всем имеющимся алгоритмам можно противопоставить такой прием, как перебор (брутфорс). Поэтому если вы не уверены в своем алгоритме, то используйте готовые.

Для этого вовсе не обязательно реализовывать такой алгоритм самостоятельно — уже написана масса компонентов, библиотек. Для любого языка программирования практически любой алгоритм можно найти в исходных кодах.

Примечание

На сопроводительном диске книги есть папка CRYPTOHASH, которая содержит примеры алгоритмов для MASM.

Сейчас я покажу еще более простой способ использования хеш-функций для расчета контрольных сумм — с помощью интерфейса Win Crypto API.

Вообще, Crypto API позволяет не только вычислять контрольную сумму, но и зашифровывать/расшифровывать данные, работать с цифровыми подписями и сертификатами (создание, проверка). Поэтому научиться работать с ним будет полезно на будущее. Плюс к тому, удобство в работе — мы можем

использовать различные алгоритмы, передавая необходимую константу как параметр.

Примечание

Для работы с библиотекой следует подключить библиотеку ADVAPI32.DLL.

Давайте рассмотрим пример — программу-калькулятор (листинг 6.4), на котором видно, как реализовать все то, о чем я говорил.

Листинг 6.4. Часть листинга программы-калькулятора с реализацией Crypto API

```
...
; Основная процедура обработки окна
proc DialogProc hWnd, uMsg, wParam, lParam
        cmp     [uMsg], WM_INITDIALOG
              ; uMsg - это сообщение, которое ОС посылает
              ; в зависимости от происходящего в системе
        je      .wminitdialog
        cmp     [uMsg], WM_COMMAND
        je      .wmcommand              ; Нажата кнопка?
        cmp     [uMsg], WM_CLOSE
        je      .wmclose
        xor     EAX, EAX
        ret
;-----------------------------------------------------------------------
  .wminitdialog:
        jmp     .processed
;-----------------------------------------------------------------------
  .wmcommand:
        cmp     [wParam], IDCANCEL     ; Нажата кнопка выхода?
        je      .wmclose
        cmp     [wParam], IDSHA1       ; Расчет по алгоритму SHA1
        mov     [algoritm], CALG_SHA
        je      .SHA1
        cmp     [wParam], IDMD5        ; Расчет по алгоритму MD5
        jne     .processed
        mov     [algoritm], CALG_MD5
;-------------------------------------------------------\\
.SHA1:
```

```
; Считывание введенного текста
        invoke  GetDlgItemText, [hWnd], EDIT_1, txt_buff, 256
        mov     [len], EAX  ; Получаем длину введенного текста
        invoke  CryptAcquireContext, hCrypt, NULL, NULL, PROV_RSA_FULL,
  CRYPT_VERIFY
        cmp     EAX, 0
        jz      .errCrypt   ; Выход, если произошла ошибка.
                            ; Функция CryptCreateHash создает объект
                            ; и вернет указатель на него в hHand
        invoke   CryptCreateHash, [hCrypt], [algoritm], 0, 0, hHand
; Рассчитываем хеш от введенной строки
        invoke   CryptHashData, [hHand], txt_buff, [len], 0
        mov      [len], 520h
; Возвращаем результат и длину хеша
        invoke   CryptGetHashParam, [hHand], 2h, hash_buff, len, 0
        mov      ECX, [len]       ; len = длина полученного хеша
        mov      EAX, hash_buff   ; Указатель на буфер с хешем
        call     int2str     ; Преобразуем значение для вывода на экран
; Освобождаем память, выделенную CryptCreateHash
        invoke   CryptDestroyHash, [hHand]
; Завершаем работу с Crypto API
        invoke   CryptReleaseContext, [hCrypt], 0
; Заносим полученную строку в поле ввода в форме
        invoke   SetDlgItemText, [hWnd], EDIT_2, strZN
.errCrypt:
        jmp     .processed
  .wmclose:
; Выход из программы
        invoke   EndDialog, [hWnd], 0

  .processed:
        mov      EAX, TRUE
  .finish:
        ret
endp
```

Данный пример снабжен подробными комментариями, поэтому дальнейшее обсуждение считаю излишним.

Вот мы и закончили такую интересную тему, как CRC. Как видите, даже проверка CRC не дает 100%-ной гарантии того, что программа не будет взломана. Но это вовсе не значит, что такая защита бесполезна! Главное — как ее применять. Если вы сложите все то, что вы узнали из данной книги, и примените на практике для защиты своей программы, это оттолкнет от взлома вашей программы не только начинающих, но и продвинутых крекеров.

Некоторые менее распространенные способы защиты программ

Я уже рассказал о самых популярных методах защиты программ. Эти методы вы можете применять в своих программах. В этом разделе я расскажу о менее распространенных способах защиты программ. Эти способы тоже применяются на практике, хотя и гораздо реже — они более трудоемки в реализации. Итак, начнем.

Шифрование всей программы или ее части

На первом месте — набирающий все большую популярность метод шифрования. Тут надо сделать пояснение. В отличие от приведенного в предыдущем разделе примера, в котором мы шифровали строки программы, здесь речь идет о шифровании исполняемого кода программы. Главная сложность этого метода заключается в том, что после шифрования программа должна сохранить работоспособность. Но тут надо подойти к делу с умом и ответственностью. Я видал и такое "чудо" программисткой мысли: программа была разбита на две части, одна часть — загрузчик-расшифровщик программы, а вторая часть — сама программа, зашифрованная простым методом XOR. Как вы знаете, это метод простого наложения. Он очень прост — одна и та же функция выполняет и шифрование и расшифровку. Недостаток такого шифрования в том, что программа до ее расшифровки полностью утрачивает работоспособность. Программисты решили избавиться от этого недостатка радикальным способом. Первая часть их программы — расшифровщик — расшифровывает всю программу в папку TEMP, которая находится в папке WINDOWS. После этого расшифровщик запускает оттуда расшифрованную программу, дожидается, когда она закончит свою работу, а затем удаляет ее из папки TEMP. Кто-то может подумать, что это хорошая идея, но так делать НЕ НАДО! Недостаток такого метода очевиден: при работе программы на диске находится полностью расшифрованная программа. Достаточно просто скопировать ее файл, и у вас будет полная рабочая копия программы, с которой вы можете делать все, что захотите. Тот, кто невнимательно читал эту

книгу или пропустил несколько глав, может возразить: "Кто знает, где находится расшифрованная программа?!" Так вот, я еще раз хочу напомнить вам, что существуют специальные программы-мониторы, которые отслеживают все обращения программы к реестру, каким-либо файлам и т. д. Вот они:

- FileMon — утилита, позволяющая вести мониторинг всех операций с файлами; имеет удобный фильтр, может сохранять отчет в файле;
- RegMon — аналог FileMon, только ведется мониторинг всех операций с реестром;
- PortMon — мониторинг работы с портами ввода/вывода;
- TCP-VIEW — монитор соединений по TCP/IP;
- RegistrUtils — набор утилит для контроля реестра; делает копии реестра, позволяет сравнивать копии и просматривать изменения;
- RegSnap — утилита, создающая "снимки" реестра и ключевых файлов и позволяющая сравнивать несколько снимков; имеет удобный интерфейс, выводит результаты (найденные различия) в удобном формате.

Поэтому еще раз хочу предостеречь вас от такого "шифрования". Этот метод если и защитит, то только от тех, кто плохо разбирается в компьютерах. Любой начинающий крекер снимет такую защиту без всякого труда.

Наиболее приемлемый путь при шифровании — сделать следующее. Программа разбивается на несколько кусков. Загрузчик программы загружает первый кусок программы в память и расшифровывает его *в памяти*. После расшифровки управление передается на этот кусок. После того как эта часть программы отработала, загрузчик удаляет ее из памяти, расшифровывает второй кусок программы в памяти, передав на него управление, и т. д. После того как вся программа отработает, загрузчик удаляет ее из памяти. Данный метод, конечно, не лишен недостатков, но он намного действеннее первого. Начинающему крекеру уже будет сложнее взломать такую программу. А если программа еще и спроектирована так, как я рассказывал в предыдущих главах, то у начинающего крекера практически ничего не выйдет.

Чтобы обезопасить программу от более опытного крекера, необходимо предусмотреть в загрузчике антиотладочные и антитрассировочные методы, а также разнообразные нестандартные способы шифрования. Но ни в коем случае не расшифровывайте в памяти всю программу сразу. Крекеру будет достаточно просто снять дамп памяти, т. е. записать копию памяти в файл на диске, немножко подправить, — и ваша программа взломана.

На этом методе и его вариациях работают некоторые упаковщики и протекторы исполняемых файлов. Но это отдельная тема для разговора, и я постараюсь рассказать о ней в будущих книгах.

"Отлов" пошаговой отладки программы

Следующий метод — это отлов пошаговой отладки программы. Самый простой и надежный способ — использовать таймер. Я уже говорил про использование таймера в программе при проверке пароля *(см. главу 5)*, теперь я расскажу, как можно использовать таймер при проверке пошаговой отладки программы.

Дело в том, что при отладке программы крекер анализирует программу вдумчиво и не спеша. Ему некуда торопиться. Вот на этом-то и можно его поймать. Делается это так. При работе программы периодически фиксируем системное время и рассчитываем время работы кусков кода между ними. При нормальной работе процессор выполняет кусок кода программы за 5—10 мс (это, конечно, зависит от размера куска кода). Так вот, если вы обнаружите, что этот же кусок кода процессор выполняет за 2—7 минут, то тут надо задуматься. Но опять хочу предупредить: ни в коем случае не прерывайте работу и не выводите никаких сообщений сразу после того, как определили, что работаете при пошаговой отладке. Самый лучший способ — направить работу программы по альтернативному руслу. Тогда пока крекер поймет, что его надули, пока определит, как это сделано, пройдет еще много времени, и всегда есть шанс, что он отстанет от вашей программы и переключит свое внимание на что-то более легкое.

Проверка на основе динамически изменяющихся параметров

Алгоритмы, использующие подобную проверку, были очень удобны по ряду причин, в том числе:

- параметры каждого компьютера уникальны;
- можно выдавать регистрационные данные (пароль) только на конкретный компьютер, привязанный к имени пользователя.

Сейчас этот метод еще применяется, но уже сдает позиции. Это связано с тем, что современный пользователь постоянно совершенствует свой компьютер, а раз меняется конфигурация компьютера, значит, меняются и его параметры и, как следствие, если программа при запуске проверит регистрационные данные, они будут отличаться. Естественно, программа откажется работать. Пользователю придется снова пройти регистрацию, а программисту — выслушать обвинения по поводу неработоспособности его программы.

Регистрация он-лайн

Суть метода регистрации он-лайн (On-Line) состоит в том, что программа в реальном времени связывается с сайтом разработчиков или с сайтом компании, осуществляющей продажу программы, и передает туда некоторый код пользователя (серийный номер диска или другого оборудования и т. п.), или этот код вычисляется каким-нибудь способом, например: из номера диска вычесть текущую дату, прибавить имя пользователя и т. д. В ответ программе передается регистрационная информация. Этот метод используется для некоторого класса программ, но для всех программ применение его нежелательно. Это связано с рядом причин.

- Никто не может гарантировать, что конкретно передаст программа в Интернет. Думаю, в современном мире мало тех, кого не волнует собственная безопасность. Таким образом, программа может передать все, что угодно: пароли, почту, параметры компьютера и т. д.
- Конечный пользователь может не иметь доступа в Интернет, например, в крупных компаниях, у которых есть свой ИТ-отдел и которые предоставляют пользователям только электронную почту. Это особенно важно для программ, работа которых не связана с Интернетом напрямую. Получается, что зарегистрировать такую программу пользователь не сможет при всем желании.

Есть и другие причины. Таким образом, наиболее очевидно применение этого метода для программ, работа которых связана с Сетью. Для остальных программ применять его нежелательно.

Глава 7

Методы противодействия и способы защиты программ от взлома

В этой главе я расскажу о способах противодействия взлому вашей программы. Я уже говорил в предыдущих главах, как можно защититься от того или иного способа взлома. Теперь я хочу рассказать об этом более подробно.

Скажу сразу, что получить 100%-ные гарантии того, что ваша программа абсолютно защищена, невозможно — это я уже показал на наглядных примерах. То, что придумал один человек, второй обязательно разгадает. Все, что вы можете, — это усложнить задачу крекеру настолько, чтобы он начал жалеть себя, свое время и решил заняться чем-нибудь другим, отказавшись взламывать вашу программу. Как я уже говорил, главная цель защиты — сделать невозможным ее взлом на протяжении определенного времени. Например, если вы написали программу, которая будет ценнной, скажем, в течение одного месяца, то вы должны сделать так, чтобы за этот месяц нельзя было подобрать код или сломать защиту программы. Тут все зависит от реализации самой защиты. Допустим, это защита "запрос пароля", т. е. при запуске программы на экране появляется окно, в котором просят ввести пароль. Если организовать такую защиту грамотно, то ее хватит даже для среднего крекера (надеюсь, вы помните правила, о которых я говорил ранее?). Самый простой и надежный способ защиты от подбора пароля методом их перебора — организовать проверку таким образом, чтобы она занимала довольно много времени. Например, если у вас пароль проверяется за 1 секунду, то за день я смогу проверить *очень* много паролей, но если ваша функция проверки пароля занимает около 2—3 минут, то на перебор паролей потребуется гораздо больше времени. Это только одна из идей, которые я хочу вам порекомендовать.

Как не надо проектировать защиту программ

Если вы написали программу и решили ее защитить, то подумайте еще раз. Вам необходимо сразу решить, от кого вы будете защищать вашу программу. Скорее всего, вы будете защищать вашу программу от крекера, тогда вам надо решить такой вопрос, как какой уровень крекера необходим для того, чтобы снять вашу защиту. Реально оцените свои шансы. Нет никакого смысла тратить на разработку защиты столько же времени, сколько заняла разработка самой программы. От этого программа только станет дороже и снизится вероятность того, что ее будут покупать. Может, дешевле и надежнее будет купить уже готовую защиту? Нет? Вы решили сделать все сами? Ну что же, тогда дам несколько советов. Надеюсь, они помогут вашей программе сопротивляться атакам крекеров.

Итак, вы решили разработать для своей программы защиту, основанную на вводе регистрационного кода. Ввод пароля или регистрационного кода — ответственный этап. Я уже показал, что у крекера есть несколько способов для нахождения пароля:

- установить точку останова (брекпойнт) на функции считывания текста из стандартного элемента ввода;
- "отловить" адрес в памяти, по которому будет записан пароль.

Это стандартный набор действий для взлома таких программ. Для того чтобы затруднить взлом на этом этапе, можно сделать следующее.

- Старайтесь не использовать стандартные обработчики компонентов (кнопок, флажков и т. п.), а организовывайте проверки в цикле сообщений — это усложнит взлом и потребует более тщательного анализа, а начинающих крекеров вовсе отпугнет.

  Как я уже говорил, современные дизассемблеры умеют распознавать стандартные процедуры высокоуровневых языков и API-функции. А если вы их не используете, то взломщику не на что ставить брекпойнты.

- Применяйте нестандартный способ ввода пароля.

  Дело в том, что большинство программистов — народ ленивый, чтобы что-то спросить у пользователя или сообщить ему, они используют стандартные окна Windows. Если вы пишете программы на Delphi или Builder C++, то самый простой путь — написать свой визуальный компонент для ввода регистрационного кода. Он, конечно, должен будет обрабатывать события от клавиатуры, но при этом момент считывания кода нельзя поймать стандартными методами.

- Не храните введенный код в одном месте.

  Разбейте введенный код на три-четыре части сразу при считывании. Если введенный регистрационный номер хранить в одном месте, то, как я показывал ранее, достаточно установить точку останова на адрес в памяти, по которому находится введенный код, и крекер попадет в функцию его проверки.

- Не храните введенный код в текстовом формате ("открытым текстом").

  Я поступаю следующим образом. Создаю несколько переменных типа `AnsiString`. Переписываю введенный код в эти переменные. Потом завожу таймеры со случайным интервалом (таймеров несколько больше, чем переменных). По срабатывании каждый таймер обрабатывает свою часть в своей переменной (шифрует и вычисляет значение). Остальные таймеры выполняют ложную проверку пароля.

- Никогда не анализируйте пароль сразу после его ввода.

  Чем это чревато, я уже показал в предыдущих главах. Лучше всего растянуть проверку во времени со случайным интервалом. Если объединить этот пункт с предыдущим, то после того, как все переменные в таймерах были обработаны, выполняется проверка кода по частям. Например, первая часть проверяется сразу, а остальные — через случайные промежутки времени. Таким образом, у меня проверка растягивается и по времени, и по коду программы.

- Можно объединить несколько советов и сделать, например, компонент, который не только запрашивает код нестандартным способом, но сразу разбивает и шифрует введенный пароль.

  Это очень хороший метод, который обманывает даже более подготовленных крекеров. Но ваш компонент должен обязательно обрабатывать данные! Тогда просто нейтрализовать или отключить его будет недостаточно, это сделает программу неработоспособной.

- Не проверяйте код только в одном месте.

  Это кажется очевидным, но большинство программ делают именно это — проверяют код в одном месте при старте программы. Чем это чревато, я уже показал при анализе программ.

- Не создавайте для проверки функцию или библиотеку.

  Как я уже говорил, написать для проверки кода функцию или библиотеку — мягко говоря, неудачный вариант. Достаточно всего лишь найти и отключить эту проверку, и защита взломана. Если вы разместите код проверки в разных частях программы (сам код, а не ссылку на функцию), то взлом затрудняется.

- Не проверяйте пароль одним алгоритмом

  Это правило — следствие предыдущего. Если проверки выполняются разными методами, то на взлом такой программы уйдет очень много времени.

- Никогда не выполняйте действия, связанные с проверкой, сразу после самой проверки.

  Это стандартная ошибка большинства защит. Мне попадались программы с хитрыми проверками пароля при этом, как только сравниваемый пароль совпадал с образцом, они тут же выдавали сообщение о его правильности. Зачем же создавать нестандартные функции запроса пароля и шифровать его, если всамом конце проверки программа вызывает стандартную функцию вывода сообщения, которую можно отловить с помощью отладчика? Это все равно что хранить пароль в открытом виде — в текстовом файле, расположенном в той же папке, что и программа, — в надежде, что его никто не заметит.

- Применяйте отвлекающие функции проверок.

  В программе, кроме реальных функций проверки кода, должны быть ложные функции проверок. Они должны вызываться сразу после ввода пароля, проводить активные действия с введенным значением, выводить сообщения о некорректности введенного кода, т. е. отвлекать внимание от реальной проверки.

- Не используйте переменную для хранения результатов проверки.

  Это очевидное предупреждение, однако ему очень редко следуют на практике. Не попадайтесь на эту удочку. Достаточно всего лишь изменить значение этой переменной, и защита падет.

- Не используйте переменную для явного ограничения функций незарегистрированной программы.

  Этот совет вытекает из предыдущего. Пускай вам в страшном сне не привидится сделать что-либо похожее на это:

```
if(Register!=true)
{
ShowMessage("Незарегистрированная версия программы");
}
else
{
// Код при верной регистрации
}
```

Для взлома такой защиты достаточно изменить значение переменной в памяти, и программа станет зарегистрированной, а установка брекпойнта на изменение этой переменной приведет нас на место ее проверки.

- Никогда не храните результаты проверки в реестре или на диске.

  Как я уже показал, есть специальные программы-мониторы, благодаря которым найти такую пометку очень просто. Лучше всего поступить так. Создайте файл на диске и храните в нем введенный пользователем пароль в том виде, в котором пользователь его ввел, и при каждом запуске программы проверяйте правильность этого пароля.

- Всегда используйте CRC (проверку контрольной суммы).

  Это очень простое решение, однако на практике применяется очень редко. А ведь это не только поможет программе противостоять взлому, но и защитит ее от вирусов. А ведь такая защита очень проста — при запуске программы рассчитывается контрольная сумма (всего файла или функции) и сравнивается с оригинальной. Однако помните, что при проверке контрольной суммы программа будет сравнивать ее с правильной, а значит, надо использовать и все вышеуказанные методы.

- Применяйте шифрование программ и данных.

  Если вы не можете сами написать программу-шифровальщик, то возьмите готовую, сейчас их можно найти великое множество. Тут главное, чтобы шифровальщик был или неизвестен (малоизвестен), или обладал той устойчивостью к взлому, которая вам необходима.

- Используйте "отлов" пошаговой отладки программы.

  Про это я уже писал, но хочу еще раз повторить: если где-нибудь в вашей программе организована проверка времени, затрачиваемого на выполнение фрагмента программы, и при выявлении недопустимо большого значения этого времени программа выполняется по альтернативному пути, то найти и сломать такую защиту не так-то просто.

- Не определяйте дату и время стандартными способами.

  Стандартные способы приведены в *главе 6*. Добавлю лишь, что если вам необходимо считывать дату или время, то придумайте что-нибудь оригинальное, например, определение даты через Интернет.

- Не стоит хранить что-либо секретное в файлах или реестре.

  Я уже писал про хранение результатов проверок в файлах и реестре. Почему-то некоторые программисты считают это хорошей идеей. Снова повторю: работа с файлами или реестром может быть детально отслежена программами-мониторами.

- Не записывайте никакие текстовые строки в программе в их реальном виде.

  Я уже показал, как легко взламываются программы, в которых присутствует сообщение вида "Unregistered version" или "Password is False". Идеальное решение — написать компонент-контейнер, в котором будут храниться строковые переменные, используемые в вашей программе.

Как "обмануть" отладчики и дизассемблеры

Способов обмана отладчиков и дизассемблеров много. Каждый профессиональный программист, хорошенько подумав, может придумать такие способы. Сейчас самым надежным и перспективным способом считается использование виртуальной машины (VM). Идея тут заключается в том, что в саму программу, которую необходимо защитить, внедряются специальные команды, которые могут обрабатываться только специальным интерпретатором — *виртуальной машиной*. Очевидно, что никакой отладчик или дизассемблер этих команд не знает и поэтому не сможет дизассемблировать программу. Если к тому же такие команды являются постоянно изменяющимися, то снятие такой защиты становится очень трудоемким делом, посильным далеко для не каждого профессионального крекера.

Я расскажу вам о наиболее распространенных, и менее сложных в реализации способах обмана отладчиков и дизассемблеров. Некоторые из них немного устарели, другие с успехом применяются и по сей день.

Для начала давайте определим, что помешает дизассемблеру:

- плохо читаемый листинг;
- непонятные вызовы функций, например, `call [ebp+022h]`;
- зашифрованные куски кода (даже с помощью простой операции XOR);
- зашифрованные строки в ресурсах.

Теперь давайте посмотрим, что помешает отладчику:

- плохо читаемый листинг;
- зашифрованные строки в ресурсах (с оговорками), а также (поскольку программа может определить, что ее выполнение ведется в отладчике):
- изменение работы программы в случае ее выполнения в отладчике;
- определение отладчика до запуска программы с последующим завершением или эмуляцией ошибки (например, ошибки базы данных).

Вот на основе этого мы и рассмотрим некоторые примеры.

Примечание

Приведенные далее приемы можно и нужно использовать в своих программах. Как я уже говорил, это очень затруднит работу крекера, а начинающих крекеров и вовсе остановит.

Для начала, как самое простое, рассмотрим использование стандартных функций.

Функция *IsDebuggerPresent*

Если отладчика в системе нет, то функция вернет в регистре EAX значение 0FFFFFFFFh. То есть в каком-либо месте программы (а лучше в нескольких) мы вызываем данную функцию. Далее мы проверяем регистр EAX, и если он содержит значение, отличное от 0FFFFFFFFh, значит, в системе установлен отладчик. Мы делаем себе пометку, но сразу ничего не предпринимаем (про это я уже говорил).

Далее у нас есть два пути.

- Запускается таймер со случайным интервалом, при срабатывании которого программа завершается.
- Программа продолжает выполняться, но работает по альтернативному алгоритму (в котором, например, не предусмотрены сохранение информации, вывод сообщений и т. д.). Конечно, этот второй алгоритм тоже надо продумать очень тщательно, иначе крекер просто сделает реверс программы ("отмотает" программу) до функции проверки.

Еще раз хочу подчеркнуть: надо делать проверку несколько раз, вставляя *функцию* проверки, а не ссылку на функцию! Это надо делать для того, чтобы крекер, найдя только одну функцию, не смог зарегистрировать программу. В листинге 7.1 приведен пример того, как делать *не надо*, а в листинге 7.2 — пример правильного алгоритма. Листинг 7.3 содержит код, определяющий выполнение программы в отладчике.

Листинг 7.1. Пример неправильного вызова функции проверки

```
int Test_debug()
{
...
Код проверки
...
```

```
}

void __fastcall TForm1::ButtonRegClick(TObject *Sender)
{
If(Test_debug()==1)
  {
    MessageBox(0,"Debug Active", "Debug Active", MB_OK);
        ...
    Отладчик обнаружен
        ...
   }
else
   {
        ...
    Отладчик не обнаружен
        ...
   }
```

Листинг 7.2. Пример правильного вызова функции проверки

```
void __fastcall TForm1::ButtonRegClick(TObject *Sender)
{
// Проверка на наличие отладчика

If(debug==true)
   {
    MessageBox(0, "Debug Active", "Debug Active", MB_OK);
        ...
    Отладчик обнаружен
        ...
   }
else
  {
       ...
   Отладчик не обнаружен
       ...
  }

...
```

И так в каждом месте (не менее трех-четырех мест). Чем больше их будет, тем лучше.

Листинг 7.3. Определение выполнения в отладчике

```
bool proc =GetProcAddress(hHndl,"DebugBreak");
  if(proc)
   {
       MessageBox(0,"Debug Active","Debug Active",MB_OK);
   }
  else
   {
    MessageBox(0,"Not Debug ","Not Debug ",MB_OK);
   }
```

Функция *CreateFileA*

В нашей программе делаем попытку открытия файла:

```
CreateFileA('\\.\NTice', ...)
```

или

```
CreateFileA('\\.\Sice', ...)
```

или

```
CreateFileA('\\.\TRW', ...)
```

и т. д.

Далее анализируем результат и принимаем решение о работе программы, например:

```
if (INVALID_HANDLE_VALUE= -1)
        MessageBox(0,"Not Debug ","Not Debug ",MB_OK);
```

Можно отследить или измерить время выполнения кода. Дело в том, что у процессора есть счетчик time stamp (отметка времени), значение которого увеличивается с каждым процессорным циклом. Для чтения значения этого счетчика есть команда

```
rdtsc
```

Она возвращает значение time stamp в регистрах EAX и EDX.

Когда работает отладчик SoftICE, значение счетчика возрастает гораздо быстрее, чем при простом выполнении программы. Например:

```
rdtsc
sub EAX, EBX
```

```
sbb EDX, ECX
rdtsc
add EAX, EBX
adc EDX, ECX
rdtsc
sub EAX, EBX
sbb EDX, ECX
```

Если в EDX будет не 0 — значит, идет трассировка.

Также можно использовать API-функцию `GetTickCount`, например:

```
a=GetTickCount();
...
Код
...
if ((GetTickCount()-a)>xxxxxx) {error!}
```

Примечание

`xxxxxx` — значение регистра при простом выполнении программы. Подбирается экспериментально, во время работы программы.

Обман дизассемблера

Следующий пример поможет нам обмануть дизассемблер. При нормальной работе программы выполняется переход на рабочий код, а при работе программы в дизассемблере переход не происходит, и начинает выполняться "мусор", что приводит к неверной работе программы.

Примечание

Вместо "мусора" можно запустить альтернативный режим работы программы.

```
if (pass=="1234")
  {
   asm
   jmp @pro
   db 77
   ; Следующие строки - "мусор"
   ...
   ;
   @pro:
   ; Рабочий код
   end;
   }
```

Примечание

Этот прием, к сожалению, не работает для IDA, т. к. IDA это интерактивный дизассемблер.

Усложнение листинга

Например, вызов функции

```
call MyProtExec
```

можно заменить таким кодом:

```
mov EAX, @MyProtExec
sub EAX, 140
push EAX
; Мусор
...
sub EAX, -39
sub EAX, -101
call EAX
```

Рассмотрим практический пример использования этого способа.

Возьмем программу, в которой есть вызов функции:

```
00401326   > A1 8B104600    mov EAX, [DWORD DS:46108B]
0040132B   . C1E0 02        shl EAX, 2
0040132E   . A3 8F104600    mov [dword DS:46108F], EAX
00401333   . 52             push EDX
00401334   . 6A 00          push 0    ; /pModule = NULL
; \GetModuleHandleA
00401336   . E8 39F20500    call <JMP.&KERNEL32.GetModuleHandleA>
0040133B   . 8BD0           mov EDX, EAX
0040133D   . E8 CA470500    call <CrakMe-6.___CRTL_VCL_Init>
```

Вот во что превратится команда после усложнения листинга:

```
00401326     EB 03          jmp short CrakMe-6.0040132B
00401328     0000           add [byte DS:EAX], AL
0040132A     00C1           add CL, AL
0040132C     E0 02          loopdne short CrakMe-6.00401330
0040132E   . A3 8F104600    mov [dword DS:46108F], EAX
```

Согласитесь, что теперь с первого взгляда тяжело понять, что здесь происходит.

Заключение

Вот мы и рассмотрели некоторые методы противодействия крекеру и его инструментам. Надеюсь, что эти знания помогут вам в ваших начинаниях.

К сожалению, за пределами книги остались такие темы, как ручная распаковка запакованных (зашифрованных) программ, снятие протекторов, написание скриптов для OllyDbg, подробное описание дизассемблера hiew и работа с ним, а также многие другие. Эти темы мы рассмотрим во втором томе книги, если он когда-либо будет написан.

Жду ваших замечаний и пожеланий (**mail@bhv.ru**).

Приложение

Описание компакт-диска

На сопроводительном компакт-диске книги вы найдете некоторые программы, которые я использовал в работе над книгой, разнообразные программы типа crackme, написанные мной для того, чтобы вы могли потренироваться в этом нелегком и увлекательном деле. Эти и многие другие программы вы найдете на моем сайте **www.abyss-soft.narod.ru**.

Содержимое папок компакт-диска:

- \Books — программы типа crackme, которые рассматриваются в книге;
- \Cryptohash — примеры реализации разнообразных алгоритмов проверки;
- \CrackMe — разнообразные программы типа crackme для новичков;
- \XOR — программа для шифрования переменных в ваших программах;
- \Plugins for OllyDBg — описанные в книге плагины для отладчика OllyDbg.

Рекомендую вам для удобства перенести файлы с сопроводительного компакт-диска книги в отдельную папку на жестком диске своего компьютера.

Список литературы*

1. Айрапетян Р. А. Отладчик SoftICE: Подробный справочник. — М.: СОЛОН-Пресс, 2004.
2. Гошко С. В. Энциклопедия по защите от вирусов. — М.: СОЛОН-Пресс, 2004.
3. Зубков С. В. Assembler для DOS, Windows и UNIX. 3-е изд. — М.: ДМК Пресс; СПб.: Питер, 2004.
4. Касперски Крис. Образ мышления — дизассемблер IDA. — М.: СОЛОН-Пресс, 2004.
5. Панов А. С. Assembler: Экспресс-курс. — СПб.: БХВ-Петербург, 2006.
6. Пирогов В. Ю. Assembler для Windows. — М.: Изд-во Молгачева С. В., 2002.
7. Юров В. Assembler: Специальный справочник. — СПб.: Питер, 2000.
8. Юров В. Assembler: Практикум. — СПб.: Питер, 2001.
9. Юров В. Assembler: Учебник для вузов. — СПб.: Питер, 2003.

* Также были использованы ресурсы Интернета.

Предметный указатель