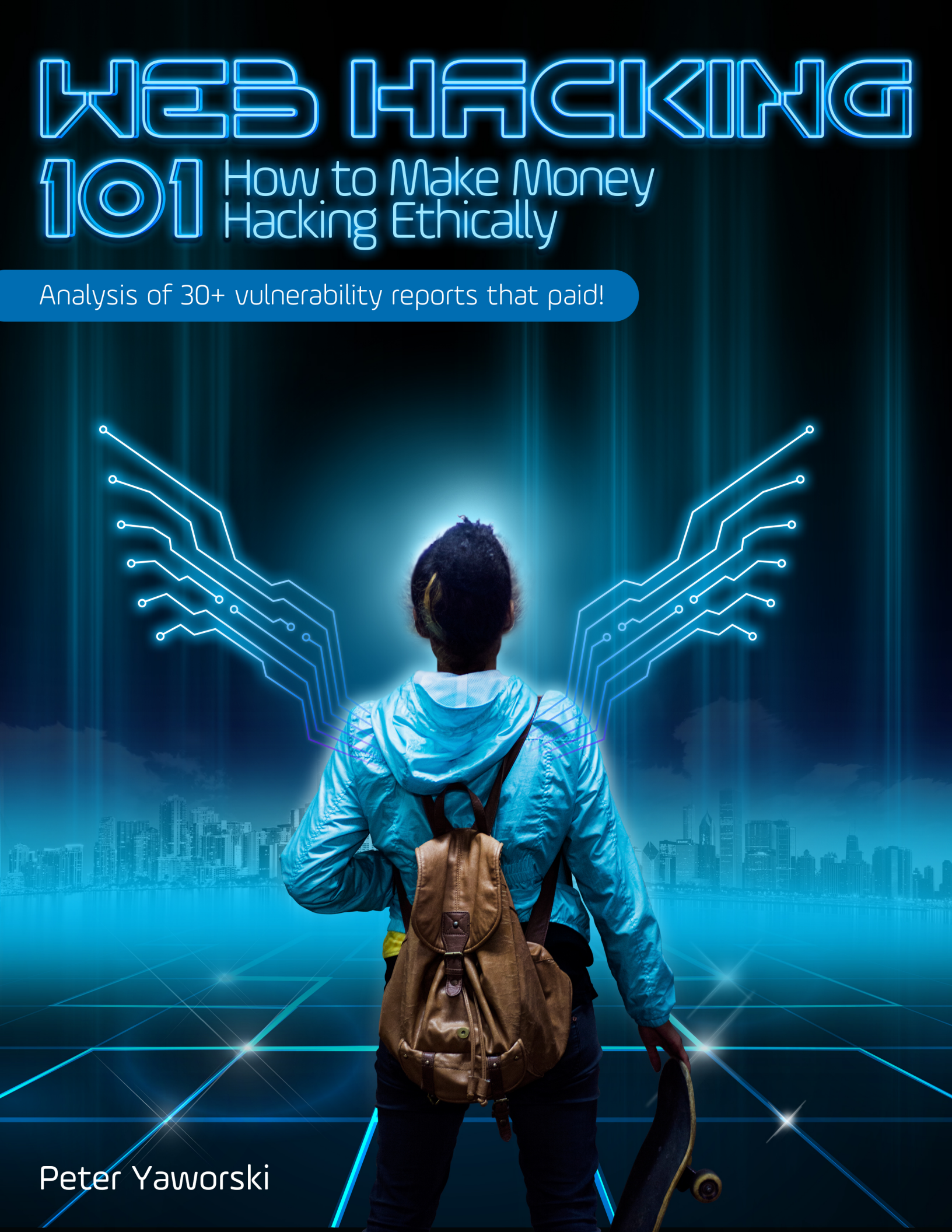
WEB HACKING 101

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 2018-11-30

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

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 Parmy 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 finsihed 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 user 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 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

Report Link: https://hackerone.com/reports/101962[1]

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

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[2]

[1] https://hackerone.com/reports/101962
[2] https://hackerone.com/reports/103772

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 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[3]

Date Reported: January 20, 2016

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)

[3] https://hackerone.com/reports/111968

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[1]

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:

[1] https://hackerone.com/reports/105953

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

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

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

Report Link: blog.mert.ninja/twitter-hpp-vulnerability[2]

Date Reported: August 23, 2015

Bounty Paid: $700

Description:

[2] http://blog.mert.ninja/blog/twitter-hpp-vulnerability

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

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[3]

Date Reported: November 2015

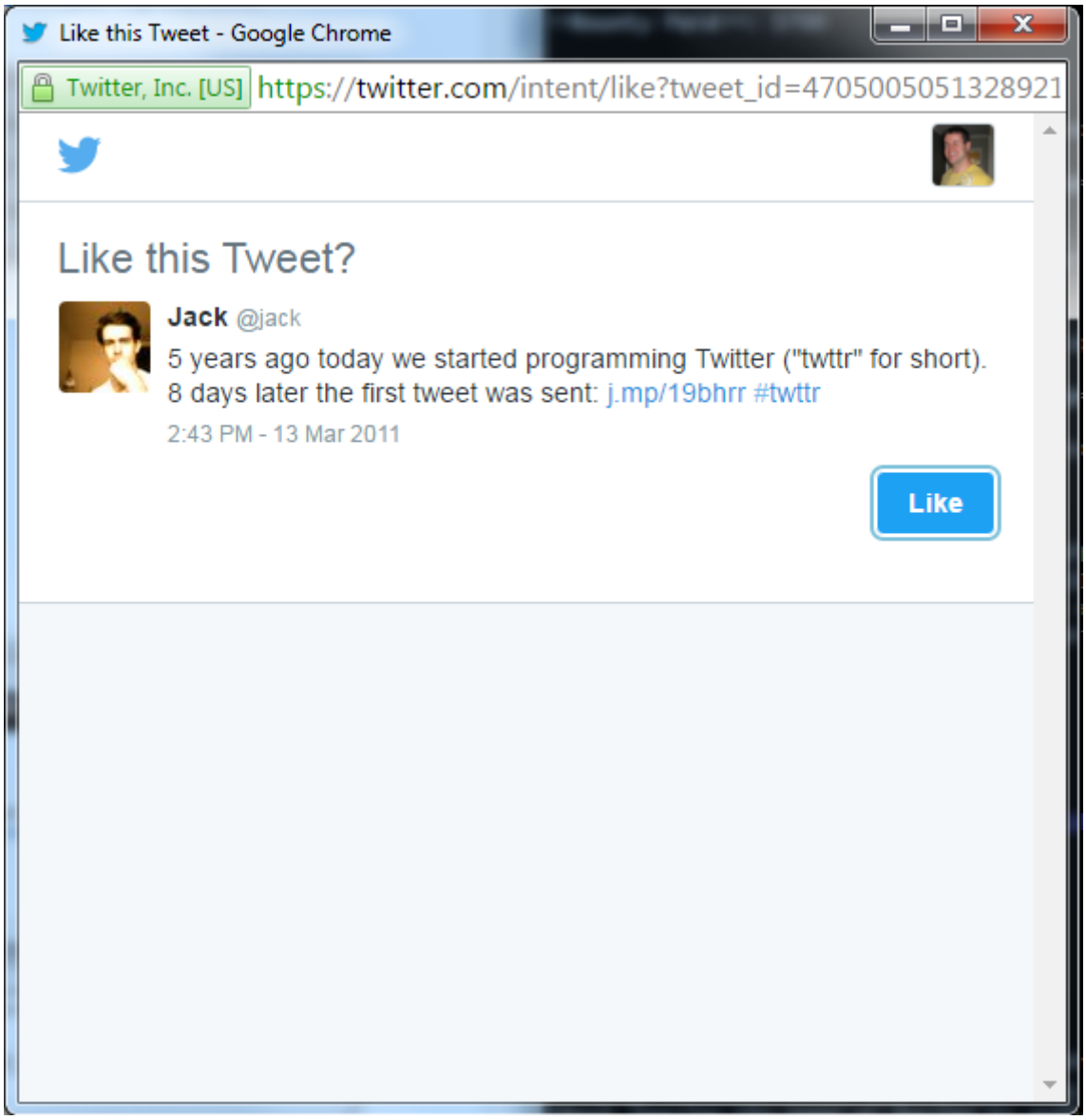
Bounty Paid: Undisclosed

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

[3] https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-intents

new app just for the interaction. Here's an example of what one of these pop-ups looks like:



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=ericrte\
st3" 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=661625230297821184 5&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.